The Python Library Reference

릴리스 *3.14.0rc3*

Guido van Rossum and the Python development team

10월 01, 2025

Contents

1 소개 **3**
1.1 가용성에 대한 참고 사항 . . . 3

2 내장 함수 **7**

3 내장 상수 **35**
3.1 site 모듈에 의해 추가된 상수들 . . . 36

4 내장형 **37**
4.1 논리값 검사 . . . 37
4.2 논리 연산 — and, or, not . . . 37
4.3 비교 . . . 38
4.4 숫자 형 — int, float, complex . . . 38
4.5 Boolean Type - bool . . . 46
4.6 이터레이터 형 . . . 46
4.7 시퀀스 형 — list, tuple, range . . . 47
4.8 Text and Binary Sequence Type Methods Summary . . . 53
4.9 텍스트 시퀀스 형 — str . . . 54
4.10 바이너리 시퀀스 형 — bytes, bytearray, memoryview . . . 67
4.11 집합 형 — set, frozenset . . . 91
4.12 매핑 형 — dict . . . 93
4.13 컨텍스트 관리자 형 . . . 98
4.14 Type Annotation Types — Generic Alias, Union . . . 99
4.15 기타 내장형 . . . 105
4.16 특수 어트리뷰트 . . . 107
4.17 Integer string conversion length limitation . . . 108

5 내장 예외 **111**
5.1 예외 컨텍스트 . . . 111
5.2 내장 예외를 상속받기 . . . 112
5.3 베이스 클래스 . . . 112
5.4 구체적인 예외 . . . 113
5.5 경고 . . . 120
5.6 예외 그룹 . . . 121
5.7 예외 계층 구조 . . . 123

6 텍스트 처리 서비스 **125**
6.1 string — 일반적인 문자열 연산 . . . 125
6.2 string.templatelib — Support for template string literals . . . 136
6.3 re — Regular expression operations . . . 142
6.4 difflib — 델타 계산을 위한 도우미 . . . 164

6.5 textwrap — 텍스트 래핑과 채우기 . . . 177
6.6 unicodedata — 유니코드 데이터베이스 . . . 180
6.7 stringprep — 인터넷 문자열 준비 . . . 183
6.8 readline — GNU readline 인터페이스 . . . 184
6.9 rlcompleter — GNU readline을 위한 완성 함수 . . . 189

7 바이너리 데이터 서비스 191
7.1 struct — 패킹 된 바이너리 데이터로 바이트열을 해석 . . . 191
7.2 codecs — 코덱 레지스트리와 베이스 클래스 . . . 198

8 데이터형 217
8.1 datetime — Basic date and time types . . . 217
8.2 zoneinfo — IANA 시간대 지원 . . . 256
8.3 calendar — 일반 달력 관련 함수 . . . 261
8.4 collections — 컨테이너 데이터형 . . . 269
8.5 collections.abc — 컨테이너의 추상 베이스 클래스 . . . 287
8.6 heapq — 힙 큐 알고리즘 . . . 293
8.7 bisect — 배열 이진 분할 알고리즘 . . . 297
8.8 array — 효율적인 숫자 배열 . . . 301
8.9 weakref — 약한 참조 . . . 304
8.10 types — 동적 형 생성과 내장형 이름 . . . 311
8.11 copy — 얕은 복사와 깊은 복사 연산 . . . 317
8.12 pprint — 예쁜 데이터 인쇄기 . . . 319
8.13 reprlib — 대안 repr() 구현 . . . 324
8.14 enum — 열거형 지원 . . . 327
8.15 graphlib — 그래프와 유사한 구조에 작동하는 기능 . . . 343

9 숫자와 수학 모듈 347
9.1 numbers — 숫자 추상 베이스 클래스 . . . 347
9.2 math — 수학 함수 . . . 350
9.3 cmath — 복소수를 위한 수학 함수 . . . 360
9.4 decimal — Decimal fixed-point and floating-point arithmetic . . . 364
9.5 fractions — 유리수 . . . 393
9.6 random — 의사 난수 생성 . . . 397
9.7 statistics — 수학 통계 함수 . . . 406

10 함수형 프로그래밍 모듈 423
10.1 itertools — Functions creating iterators for efficient looping . . . 423
10.2 functools — 고차 함수와 콜러블 객체에 대한 연산 . . . 440
10.3 operator — 함수로서의 표준 연산자 . . . 451

11 파일과 디렉터리 액세스 459
11.1 pathlib — Object-oriented filesystem paths . . . 459
11.2 os.path — 일반적인 경로명 조작 . . . 486
11.3 stat — stat() 결과 해석하기 . . . 493
11.4 filecmp — 파일과 디렉터리 비교 . . . 499
11.5 tempfile — 임시 파일과 디렉터리 생성 . . . 501
11.6 glob — 유닉스 스타일 경로명 패턴 확장 . . . 506
11.7 fnmatch — 유닉스 파일명 패턴 일치 . . . 509
11.8 linecache — 텍스트 줄에 대한 무작위 액세스 . . . 510
11.9 shutil — 고수준 파일 연산 . . . 511

12 데이터 지속성 523
12.1 pickle — 파이썬 객체 직렬화 . . . 523
12.2 copyreg — pickle 지원 함수 등록 . . . 540
12.3 shelve — 파이썬 객체 지속성 . . . 540
12.4 marshal — 내부 파이썬 객체 직렬화 . . . 543
12.5 dbm — 유닉스 "데이터베이스" 인터페이스 . . . 545
12.6 sqlite3 — DB-API 2.0 interface for SQLite databases . . . 551

13 데이터 압축 및 보관 **583**
13.1 The compression package . . . 583
13.2 compression.zstd — Compression compatible with the Zstandard format . . . 583
13.3 zlib — **gzip** 과 호환되는 압축 . . . 594
13.4 gzip — **gzip** 파일 지원 . . . 598
13.5 bz2 — **bzip2** 압축 지원 . . . 601
13.6 lzma — LZMA 알고리즘을 사용한 압축 . . . 606
13.7 zipfile — ZIP 아카이브 작업 . . . 611
13.8 tarfile — tar 아카이브 파일 읽기와 쓰기 . . . 623

14 파일 형식 **641**
14.1 csv — CSV 파일 읽기와 쓰기 . . . 641
14.2 configparser — 구성 파일 구문 분석기 . . . 649
14.3 tomllib — Parse TOML files . . . 667
14.4 netrc — netrc 파일 처리 . . . 669
14.5 plistlib — 애플 .plist 파일 생성과 구문 분석 . . . 670

15 암호화 서비스 **673**
15.1 hashlib — 보안 해시와 메시지 요약 . . . 673
15.2 hmac — 메시지 인증을 위한 키 해싱 . . . 685
15.3 secrets — 비밀 관리를 위한 안전한 난수 생성 . . . 686

16 일반 운영 체제 서비스 **689**
16.1 os — Miscellaneous operating system interfaces . . . 689
16.2 io — 스트림 작업을 위한 핵심 도구 . . . 757
16.3 time — 시간 액세스와 변환 . . . 771
16.4 logging — 파이썬 로깅 시설 . . . 784
16.5 logging.config — 로깅 구성 . . . 802
16.6 logging.handlers — 로깅 처리기 . . . 814
16.7 platform — 하부 플랫폼의 식별 데이터에 대한 액세스 . . . 828
16.8 errno — 표준 errno 시스템 기호 . . . 833
16.9 ctypes — A foreign function library for Python . . . 841

17 Command-line interface libraries **881**
17.1 argparse — Parser for command-line options, arguments and subcommands . . . 881
17.2 optparse — 명령 줄 옵션용 구문 분석기 . . . 926
17.3 getpass — 이식성 있는 암호 입력 . . . 953
17.4 fileinput — 여러 입력 스트림에서 줄을 이터레이트 하기 . . . 954
17.5 curses — 문자 셀 디스플레이를 위한 터미널 처리 . . . 957
17.6 curses.textpad — curses 프로그램을 위한 텍스트 입력 위젯 . . . 982
17.7 curses.ascii — ASCII 문자용 유틸리티 . . . 984
17.8 curses.panel — curses 용 패널 스택 확장 . . . 987
17.9 cmd — 줄 지향 명령 인터프리터 지원 . . . 988

18 동시 실행 **995**
18.1 threading — 스레드 기반 병렬 처리 . . . 995
18.2 multiprocessing — Process-based parallelism . . . 1012
18.3 multiprocessing.shared_memory — 프로세스 간 직접 액세스를 위한 공유 메모리 . . . 1057
18.4 concurrent 패키지 . . . 1063
18.5 concurrent.futures — 병렬 작업 실행하기 . . . 1063
18.6 concurrent.interpreters — Multiple interpreters in the same process . . . 1072
18.7 subprocess — 서브 프로세스 관리 . . . 1077
18.8 sched — 이벤트 스케줄러 . . . 1096
18.9 queue — 동기화된 큐 클래스 . . . 1098
18.10 contextvars — 컨텍스트 변수 . . . 1101
18.11 _thread — 저수준 스레드 API . . . 1105

19 네트워킹과 프로세스 간 통신 **1109**
19.1 asyncio — 비동기 I/O . . . 1109

19.2 socket — Low-level networking interface . . . 1212
19.3 ssl — TLS/SSL wrapper for socket objects . . . 1241
19.4 select — I/O 완료 대기 . . . 1277
19.5 selectors — 고수준 I/O 다중화 . . . 1284
19.6 signal — 비동기 이벤트에 대한 처리기 설정 . . . 1288
19.7 mmap — 메모리 맵 파일 지원 . . . 1297

20 인터넷 데이터 처리 1303
20.1 email — 전자 메일과 MIME 처리 패키지 . . . 1303
20.2 json — JSON 인코더와 디코더 . . . 1360
20.3 mailbox — 다양한 형식의 사서함 조작하기 . . . 1370
20.4 mimetypes — 파일명을 MIME 유형에 매핑 . . . 1389
20.5 base64 — Base16, Base32, Base64, Base85 데이터 인코딩 . . . 1394
20.6 binascii — 바이너리와 ASCII 간의 변환 . . . 1398
20.7 quopri — MIME quoted-printable 데이터 인코딩과 디코딩 . . . 1400

21 구조화된 마크업 처리 도구 1401
21.1 html — 하이퍼텍스트 마크업 언어 지원 . . . 1401
21.2 html.parser — 간단한 HTML과 XHTML 구문 분석기 . . . 1402
21.3 html.entities — HTML 일반 엔티티의 정의 . . . 1406
21.4 XML 처리 모듈 . . . 1407
21.5 xml.etree.ElementTree — ElementTree XML API . . . 1408
21.6 xml.dom — 문서 객체 모델 API . . . 1426
21.7 xml.dom.minidom — 최소 DOM 구현 . . . 1436
21.8 xml.dom.pulldom — 부분 DOM 트리 구축 지원 . . . 1440
21.9 xml.sax — SAX2 구문 분석기 지원 . . . 1442
21.10 xml.sax.handler — SAX 처리기의 베이스 클래스 . . . 1444
21.11 xml.sax.saxutils — SAX 유틸리티 . . . 1449
21.12 xml.sax.xmlreader — XML 구문 분석기 인터페이스 . . . 1450
21.13 xml.parsers.expat — Expat을 사용한 빠른 XML 구문 분석 . . . 1454

22 인터넷 프로토콜과 지원 1465
22.1 webbrowser — 편리한 웹 브라우저 제어기 . . . 1465
22.2 wsgiref — WSGI 유틸리티와 참조 구현 . . . 1468
22.3 urllib — URL 처리 모듈 . . . 1478
22.4 urllib.request — URL을 열기 위한 확장 가능한 라이브러리 . . . 1478
22.5 urllib.response — urllib가 사용하는 응답 클래스 . . . 1495
22.6 urllib.parse — URL을 구성 요소로 구문 분석 . . . 1496
22.7 urllib.error — urllib.request에 의해 발생하는 예외 클래스 . . . 1504
22.8 urllib.robotparser — robots.txt 구문 분석기 . . . 1505
22.9 http — HTTP 모듈 . . . 1506
22.10 http.client — HTTP 프로토콜 클라이언트 . . . 1510
22.11 ftplib — FTP 프로토콜 클라이언트 . . . 1517
22.12 poplib — POP3 프로토콜 클라이언트 . . . 1524
22.13 imaplib — IMAP4 프로토콜 클라이언트 . . . 1527
22.14 smtplib — SMTP 프로토콜 클라이언트 . . . 1534
22.15 uuid — UUID objects according to **RFC 9562** . . . 1541
22.16 socketserver — 네트워크 서버를 위한 프레임워크 . . . 1547
22.17 http.server — HTTP 서버 . . . 1556
22.18 http.cookies — HTTP 상태 관리 . . . 1563
22.19 http.cookiejar — HTTP 클라이언트를 위한 쿠키 처리 . . . 1567
22.20 xmlrpc — XMLRPC 서버와 클라이언트 모듈 . . . 1575
22.21 xmlrpc.client — XML-RPC 클라이언트 액세스 . . . 1576
22.22 xmlrpc.server — 기본 XML-RPC 서버 . . . 1583
22.23 ipaddress — IPv4/IPv6 조작 라이브러리 . . . 1589

23 멀티미디어 서비스 1605
23.1 wave — WAV 파일 읽고 쓰기 . . . 1605
23.2 colorsys — 색 체계 간의 변환 . . . 1608

24 국제화 **1609**
24.1 gettext — 다국어 국제화 서비스 . . . 1609
24.2 locale — 국제화 서비스 . . . 1617

25 Graphical user interfaces with Tk **1627**
25.1 tkinter — Tcl/Tk 파이썬 인터페이스 . . . 1627
25.2 tkinter.colorchooser — 색상 선택 대화 상자 . . . 1640
25.3 tkinter.font — Tkinter 글꼴 래퍼 . . . 1640
25.4 Tkinter 대화 상자 . . . 1641
25.5 tkinter.messagebox — Tkinter 메시지 프롬프트 . . . 1644
25.6 tkinter.scrolledtext — 스크롤 되는 Text 위젯 . . . 1647
25.7 tkinter.dnd — 드래그 앤드 드롭 지원 . . . 1647
25.8 tkinter.ttk — Tk 테마 위젯 . . . 1648
25.9 IDLE — 파이썬 편집기와 셸 . . . 1666
25.10 turtle — Turtle graphics . . . 1678

26 개발 도구 **1717**
26.1 typing — Support for type hints . . . 1717
26.2 pydoc — 설명서 생성과 온라인 도움말 시스템 . . . 1773
26.3 파이썬 개발 모드 . . . 1774
26.4 doctest — 대화형 파이썬 예제 테스트 . . . 1777
26.5 unittest — 단위 테스트 프레임워크 . . . 1800
26.6 unittest.mock — 모의 객체 라이브러리 . . . 1831
26.7 unittest.mock — 시작하기 . . . 1873
26.8 test — 파이썬 용 회귀 테스트 패키지 . . . 1893
26.9 test.support — 파이썬 테스트 스위트용 유틸리티 . . . 1896
26.10 test.support.socket_helper — 소켓 테스트용 유틸리티 . . . 1905
26.11 test.support.script_helper — 파이썬 실행 테스트용 유틸리티 . . . 1906
26.12 test.support.bytecode_helper — 올바른 바이트 코드 생성 테스트를 위한 지원 도구 . 1907
26.13 test.support.threading_helper — threading 테스트용 유틸리티 . . . 1907
26.14 test.support.os_helper — os 테스트용 유틸리티 . . . 1908
26.15 test.support.import_helper — 임포트 테스트용 유틸리티 . . . 1910
26.16 test.support.warnings_helper — warnings 테스트용 유틸리티 . . . 1912

27 디버깅과 프로파일링 **1915**
27.1 감사 이벤트 표 . . . 1915
27.2 bdb — 디버거 프레임워크 . . . 1919
27.3 faulthandler — 파이썬 트레이스백 덤프 . . . 1925
27.4 pdb — 파이썬 디버거 . . . 1928
27.5 파이썬 프로파일러 . . . 1938
27.6 timeit — 작은 코드 조각의 실행 시간 측정 . . . 1947
27.7 trace — 파이썬 문장 실행 추적 . . . 1952
27.8 tracemalloc — 메모리 할당 추적 . . . 1954

28 소프트웨어 패키징 및 배포 **1965**
28.1 ensurepip — pip 설치 프로그램 부트스트랩 . . . 1965
28.2 venv — 가상 환경 생성 . . . 1967
28.3 zipapp — 실행 가능한 파이썬 zip 아카이브 관리 . . . 1976

29 파이썬 실행시간 서비스 **1981**
29.1 sys — 시스템 특정 파라미터와 함수 . . . 1981
29.2 sys.monitoring — Execution event monitoring . . . 2008
29.3 sysconfig — 파이썬의 구성 정보에 접근하기 . . . 2013
29.4 builtins — 내장 객체 . . . 2019
29.5 __main__ — 최상위 코드 환경 . . . 2020
29.6 warnings — 경고 제어 . . . 2025
29.7 dataclasses — 데이터 클래스 . . . 2033
29.8 contextlib — with 문 컨텍스트를 위한 유틸리티 . . . 2044
29.9 abc — 추상 베이스 클래스 . . . 2058

29.10 atexit — 종료 처리기 . . . 2063
29.11 traceback — 스택 트레이스백 인쇄와 조회 . . . 2065
29.12 __future__ — 퓨처 문 정의 . . . 2075
29.13 gc — 가비지 수거기 인터페이스 . . . 2077
29.14 inspect — 라이브 객체 검사 . . . 2081
29.15 annotationlib — Functionality for introspecting annotations . . . 2100
29.16 site — 사이트별 구성 훅 . . . 2108

30 사용자 정의 파이썬 인터프리터 2113
30.1 code — 인터프리터 베이스 클래스 . . . 2113
30.2 codeop — 파이썬 코드 컴파일 . . . 2115

31 모듈 임포트 하기 2117
31.1 zipimport — Zip 저장소에서 모듈 임포트 . . . 2117
31.2 pkgutil — 패키지 확장 유틸리티 . . . 2119
31.3 modulefinder — 스크립트에서 사용되는 모듈 찾기 . . . 2122
31.4 runpy — 파이썬 모듈 찾기와 실행 . . . 2123
31.5 importlib — import의 구현 . . . 2125
31.6 importlib.resources – Package resource reading, opening and access . . . 2145
31.7 importlib.resources.abc – Abstract base classes for resources . . . 2148
31.8 importlib.metadata – 패키지 메타데이터 액세스 . . . 2150
31.9 The initialization of the sys.path module search path . . . 2158

32 파이썬 언어 서비스 2161
32.1 ast — Abstract syntax trees . . . 2161
32.2 symtable — 컴파일러 심볼 테이블 액세스 . . . 2200
32.3 token — 파이썬 구문 분석 트리에 사용되는 상수 . . . 2205
32.4 keyword — 파이썬 키워드 검사 . . . 2210
32.5 tokenize — 파이썬 소스를 위한 토크나이저 . . . 2210
32.6 tabnanny — 모호한 들여쓰기 감지 . . . 2214
32.7 pyclbr — 파이썬 모듈 브라우저 지원 . . . 2215
32.8 py_compile — 파이썬 소스 파일 컴파일 . . . 2217
32.9 compileall — 파이썬 라이브러리 바이트 컴파일하기 . . . 2218
32.10 dis — Disassembler for Python bytecode . . . 2222
32.11 pickletools — 피클 개발자를 위한 도구 . . . 2245

33 MS 윈도우 특정 서비스 2247
33.1 msvcrt — MS VC++ 런타임의 유용한 루틴 . . . 2247
33.2 winreg — 윈도우 레지스트리 액세스 . . . 2250
33.3 winsound — 윈도우용 소리 재생 인터페이스 . . . 2258

34 Unix-specific services 2263
34.1 shlex — 간단한 어휘 분석 . . . 2263
34.2 posix — 가장 일반적인 POSIX 시스템 호출 . . . 2268
34.3 pwd — 암호 데이터베이스 . . . 2269
34.4 grp — 그룹 데이터베이스 . . . 2270
34.5 termios — POSIX 스타일 tty 제어 . . . 2271
34.6 tty — 터미널 제어 함수 . . . 2272
34.7 pty — 의사 터미널 유틸리티 . . . 2273
34.8 fcntl — fcntl과 ioctl 시스템 호출 . . . 2275
34.9 resource — 자원 사용 정보 . . . 2278
34.10 syslog — 유닉스 syslog 라이브러리 루틴 . . . 2282

35 Modules command-line interface (CLI) 2287

36 대체된 모듈 2289
36.1 getopt — 명령 줄 옵션용 C 스타일 구문 분석기 . . . 2289

37 Removed Modules 2293

37.1 aifc — AIFF와 AIFC 파일 읽고 쓰기 . . . 2293
37.2 asynchat — 비동기 소켓 명령/응답 처리기 . . . 2293
37.3 asyncore — 비동기 소켓 처리기 . . . 2293
37.4 audioop — 원시 오디오 데이터 조작 . . . 2294
37.5 cgi — 공통 게이트웨이 인터페이스 지원 . . . 2294
37.6 cgitb — CGI 스크립트를 위한 트레이스백 관리자 . . . 2294
37.7 chunk — IFF 청크된 데이터 읽기 . . . 2294
37.8 crypt — 유닉스 비밀번호 확인 함수 . . . 2294
37.9 distutils — 파이썬 모듈 빌드와 설치 . . . 2294
37.10 imghdr — 이미지 유형 판단 . . . 2295
37.11 imp — 임포트 내부 액세스 . . . 2295
37.12 mailcap — Mailcap 파일 처리 . . . 2295
37.13 msilib — 마이크로소프트 설치 파일 읽고 쓰기 . . . 2295
37.14 nis — Sun의 NIS(옐로 페이지)에 대한 인터페이스 . . . 2295
37.15 nntplib — NNTP 프로토콜 클라이언트 . . . 2295
37.16 ossaudiodev — OSS 호환 오디오 장치 액세스 . . . 2296
37.17 pipes — 셸 파이프라인에 대한 인터페이스 . . . 2296
37.18 smtpd — SMTP 서버 . . . 2296
37.19 sndhdr — 음향 파일 유형 판단 . . . 2296
37.20 spwd — 섀도 암호 데이터베이스 . . . 2296
37.21 sunau — Sun AU 파일 읽고 쓰기 . . . 2296
37.22 telnetlib — 텔넷 클라이언트 . . . 2297
37.23 uu — uuencode 파일 인코딩과 디코딩 . . . 2297
37.24 xdrlib — XDR 데이터 인코딩과 디코딩 . . . 2297

38 Security Considerations **2299**

A 용어집 **2301**

B 이 설명서에 관하여 **2319**
B.1 파이썬 설명서의 공헌자들 . . . 2319

C 역사와 라이센스 **2321**
C.1 소프트웨어의 역사 . . . 2321
C.2 파이썬에 액세스하거나 사용하기 위한 이용 약관 . . . 2322
C.3 포함된 소프트웨어에 대한 라이센스 및 승인 . . . 2325

D 저작권 **2341**

Bibliography **2343**

Python 모듈 목록 **2345**

색인 **2349**

reference-index 는 파이썬 언어의 정확한 문법과 의미를 설명하고 있지만, 이 라이브러리 레퍼런스 설명서는 파이썬과 함께 배포되는 표준 라이브러리를 설명합니다. 또한, 파이썬 배포판에 일반적으로 포함되어있는 선택적 구성 요소 중 일부를 설명합니다.

파이썬의 표준 라이브러리는 매우 광범위하며, 아래 나열된 긴 목차에 표시된 대로 다양한 기능을 제공합니다. 라이브러리에는 일상적인 프로그래밍에서 발생하는 많은 문제에 대한 표준적인 해결책을 제공하는 파이썬으로 작성된 모듈뿐만 아니라, 파일 I/O와 같은 시스템 기능에 액세스하는 (C로 작성된) 내장 모듈들이 포함됩니다 (이 모듈들이 없다면 파이썬 프로그래머가 액세스할 방법은 없습니다). 이 모듈 중 일부는 플랫폼 관련 사항을 플랫폼 중립적인 API들로 추상화시킴으로써, 파이썬 프로그램의 이식성을 권장하고 개선하도록 명시적으로 설계되었습니다.

윈도우 플랫폼용 파이썬 설치 프로그램은 일반적으로 전체 표준 라이브러리를 포함하며 종종 많은 추가 구성 요소도 포함합니다. 유닉스와 같은 운영체제의 경우, 파이썬은 일반적으로 패키지 모음으로 제공되기 때문에, 운영 체제와 함께 제공되는 패키지 도구를 사용하여 선택적 구성 요소의 일부 또는 전부를 구해야 할 수 있습니다.

표준 라이브러리 외에도, 수십만 가지 컴포넌트(개별 프로그램과 모듈부터 패키지 및 전체 응용 프로그램 개발 프레임워크까지)로 구성된 활발한 모음이 있는데, 파이썬 패키지 색인 에서 얻을 수 있습니다.

CHAPTER 1

소개

"파이썬 라이브러리" 에는 여러 가지 구성 요소가 포함되어 있습니다.

여기에는 일반적으로 숫자 및 리스트와 같이 언어의 "핵심" 부분으로 간주하는 데이터형이 포함됩니다. 이러한 형의 경우, 파이썬 언어 핵심은 리터럴의 형식을 정의하고 그 의미에 몇 가지 제약을 가하지만, 의미를 완전히 정의하지는 않습니다. (반면에, 언어 핵심은 연산자의 철자법과 우선순위와 같은 문법적 속성을 정의합니다.)

라이브러리는 또한 내장 함수와 예외를 포함합니다 — `import` 문을 쓰지 않고도 모든 파이썬 코드에서 사용할 수 있는 객체들입니다. 이들 중 일부는 언어 핵심에 의해 정의되지만, 핵심 의미에 필수적인 것은 아니며 여기에서 설명합니다.

그러나 라이브러리 대부분은 모듈 컬렉션으로 구성됩니다. 이 컬렉션을 나누는 데는 여러 가지 방법이 있습니다. 일부 모듈은 C로 작성되고 파이썬 인터프리터에 내장되어 있습니다; 다른 것은 파이썬으로 작성되고 소스 형식으로 임포트 됩니다. 일부 모듈은 스택 추적 인쇄와 같이 파이썬에 매우 특정한 인터페이스를 제공합니다; 일부는 특정 하드웨어에 대한 액세스와 같이 운영 체제에 특정한 인터페이스를 제공합니다; 다른 것은 월드 와이드 웹과 같은 응용 프로그램 영역에 특정한 인터페이스를 제공합니다. 일부 모듈은 파이썬의 모든 버전과 이식에서 사용할 수 있습니다; 다른 것은 하위 시스템이 지원하거나 요구할 때만 사용할 수 있습니다; 그러나 다른 것들은 파이썬이 컴파일되고 설치될 때 특정 설정 옵션이 선택되었을 때만 사용할 수 있습니다.

이 설명서는 "안쪽에서부터 밖으로" 구성되어 있습니다. 먼저 내장 함수, 데이터형 및 예외, 마지막으로 관련 모듈의 장으로 그룹화된 모듈들을 설명합니다.

즉, 처음부터 이 설명서를 읽고, 지루할 때 다음 장으로 건너뛰면, 파이썬 라이브러리가 지원하는 사용 가능한 모듈과 응용 프로그램 영역에 대한 적당한 개요를 얻게 됩니다. 물론 소설처럼 읽을 필요는 없습니다. (설명서 앞에 있는) 목차를 검색하거나, (뒤에 있는) 색인에서 특정 함수, 모듈 또는 용어를 찾을 수도 있습니다. 그리고 마지막으로, 무작위 주제에 대해 배우는 것을 즐긴다면, 임의의 페이지 번호 (모듈 *random* 참조)를 선택하고 한두 섹션을 읽으면 됩니다. 이 설명서의 섹션을 읽는 순서와 관계없이, 내장 함수 장에서 시작하는 것이 도움이 되는데, 설명서의 나머지 부분은 이 내용에 익숙하다고 가정하기 때문입니다.

쇼를 시작합시다!

1.1 가용성에 대한 참고 사항

- "가용성: 유닉스" 참고 사항은 이 기능이 유닉스 시스템에서 일반적으로 발견된다는 것을 뜻합니다. 특정 운영 체제에 이 기능이 존재하는지에 관한 어떠한 주장도 하지 않습니다.
- 별도로 언급되지 않은 경우, "가용성: 유닉스" 를 주장하는 모든 기능은 유닉스 코어를 기반으로 하는 맥 OS, iOS 및 안드로이드에서 지원됩니다.

- If an availability note contains both a minimum Kernel version and a minimum libc version, then both conditions must hold. For example a feature with note *Availability: Linux >= 3.17 with glibc >= 2.27* requires both Linux 3.17 or newer and glibc 2.27 or newer.

1.1.1 웹어셈블리 플랫폼

The WebAssembly platforms `wasm32-emscripten` (Emscripten) and `wasm32-wasi` (WASI) provide a subset of POSIX APIs. WebAssembly runtimes and browsers are sandboxed and have limited access to the host and external resources. Any Python standard library module that uses processes, threading, networking, signals, or other forms of inter-process communication (IPC), is either not available or may not work as on other Unix-like systems. File I/O, file system, and Unix permission-related functions are restricted, too. Emscripten does not permit blocking I/O. Other blocking operations like `sleep()` block the browser event loop.

The properties and behavior of Python on WebAssembly platforms depend on the Emscripten-SDK or WASI-SDK version, WASM runtimes (browser, NodeJS, wasmtime), and Python build time flags. WebAssembly, Emscripten, and WASI are evolving standards; some features like networking may be supported in the future.

For Python in the browser, users should consider Pyodide or PyScript. PyScript is built on top of Pyodide, which itself is built on top of CPython and Emscripten. Pyodide provides access to browsers' JavaScript and DOM APIs as well as limited networking capabilities with JavaScript' s `XMLHttpRequest` and `Fetch` APIs.

- Process-related APIs are not available or always fail with an error. That includes APIs that spawn new processes (`fork()`, `execve()`), wait for processes (`waitpid()`), send signals (`kill()`), or otherwise interact with processes. The `subprocess` is importable but does not work.
- The `socket` module is available, but is limited and behaves differently from other platforms. On Emscripten, sockets are always non-blocking and require additional JavaScript code and helpers on the server to proxy TCP through WebSockets; see Emscripten Networking for more information. WASI snapshot preview 1 only permits sockets from an existing file descriptor.
- Some functions are stubs that either don' t do anything and always return hardcoded values.
- Functions related to file descriptors, file permissions, file ownership, and links are limited and don' t support some operations. For example, WASI does not permit symlinks with absolute file names.

1.1.2 모바일 플랫폼

Android and iOS are, in most respects, POSIX operating systems. File I/O, socket handling, and threading all behave as they would on any POSIX operating system. However, there are several major differences:

- Mobile platforms can only use Python in "embedded" mode. There is no Python REPL, and no ability to use separate executables such as **`python`** or **`pip`**. To add Python code to your mobile app, you must use the Python embedding API. For more details, see using-android and using-ios.
- Subprocesses:
  - On Android, creating subprocesses is possible but officially unsupported. In particular, Android does not support any part of the System V IPC API, so `multiprocessing` is not available.
  - An iOS app cannot use any form of subprocessing, multiprocessing, or inter-process communication. If an iOS app attempts to create a subprocess, the process creating the subprocess will either lock up, or crash. An iOS app has no visibility of other applications that are running, nor any ability to communicate with other running applications, outside of the iOS-specific APIs that exist for this purpose.
- Mobile apps have limited access to modify system resources (such as the system clock). These resources will often be *readable*, but attempts to modify those resources will usually fail.
- Console input and output:
  - On Android, the native `stdout` and `stderr` are not connected to anything, so Python installs its own streams which redirect messages to the system log. These can be seen under the tags `python.stdout` and `python.stderr` respectively.
  - iOS apps have a limited concept of console output. `stdout` and `stderr` *exist*, and content written to `stdout` and `stderr` will be visible in logs when running in Xcode, but this content *won' t* be recorded

in the system log. If a user who has installed your app provides their app logs as a diagnostic aid, they will not include any detail written to `stdout` or `stderr`.

– Mobile apps have no usable `stdin` at all. While apps can display an on-screen keyboard, this is a software feature, not something that is attached to `stdin`.

  As a result, Python modules that involve console manipulation (such as *curses* and *readline*) are not available on mobile platforms.

CHAPTER 2

내장 함수

파이썬 인터프리터에는 항상 사용할 수 있는 많은 함수와 형이 내장되어 있습니다. 여기에서 알파벳 순으로 나열합니다.

내장 함수

A
abs()
aiter()
all()
anext()
any()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

abs(*number*, / (*Positional-only parameter separator* (*PEP 570*)))

Return the absolute value of a number. The argument may be an integer, a floating-point number, or an object implementing `__abs__()`. If the argument is a complex number, its magnitude is returned.

aiter(*async_iterable*, /)

Return an *asynchronous iterator* for an *asynchronous iterable*. Equivalent to calling `x.__aiter__()`.

Note: Unlike `iter()`, `aiter()` has no 2-argument variant.

Added in version 3.10.

all(*iterable*, /)

iterable 의 모든 요소가 참이면 (또는 iterable 이 비어있으면) `True` 를 돌려줍니다. 다음과 동등합니다:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

awaitable anext(*async_iterator*, /)

awaitable anext(*async_iterator, default, /*)

When awaited, return the next item from the given *asynchronous iterator*, or *default* if given and the iterator is exhausted.

This is the async variant of the `next()` builtin, and behaves similarly.

This calls the `__anext__()` method of *async_iterator*, returning an *awaitable*. Awaiting this returns the next value of the iterator. If *default* is given, it is returned if the iterator is exhausted, otherwise `StopAsyncIteration` is raised.

Added in version 3.10.

any(*iterable, /*)

iterable 의 요소 중 어느 하나라도 참이면 `True` 를 돌려줍니다. iterable 이 비어 있으면 `False` 를 돌려줍니다. 다음과 동등합니다:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

ascii(*object, /*)

As `repr()`, return a string containing a printable representation of an object, but escape the non-ASCII characters in the string returned by `repr()` using `\x`, `\u`, or `\U` escapes. This generates a string similar to that returned by `repr()` in Python 2.

bin(*integer, /*)

Convert an integer number to a binary string prefixed with "0b". The result is a valid Python expression. If *integer* is not a Python `int` object, it has to define an `__index__()` method that returns an integer. Some examples:

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

If the prefix "0b" is desired or not, you can use either of the following ways.

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

자세한 내용은 `format()` 을 보세요.

class bool(*object=False, /*)

Return a Boolean value, i.e. one of `True` or `False`. The argument is converted using the standard *truth testing procedure*. If the argument is false or omitted, this returns `False`; otherwise, it returns `True`. The `bool` class is a subclass of `int` (see *숫자 형 — int, float, complex*). It cannot be subclassed further. Its only instances are `False` and `True` (see *Boolean Type - bool*).

버전 3.7 에서 변경: The parameter is now positional-only.

breakpoint(**args, **kws*)

This function drops you into the debugger at the call site. Specifically, it calls `sys.breakpointhook()`, passing `args` and `kws` straight through. By default, `sys.breakpointhook()` calls `pdb.set_trace()` expecting no arguments. In this case, it is purely a convenience function so you don't have to explicitly import `pdb` or type as much code to enter the debugger. However, `sys.breakpointhook()` can be set to some other function and `breakpoint()` will automatically call that, allowing you to drop into the debugger of choice. If `sys.breakpointhook()` is not accessible, this function will raise `RuntimeError`.

By default, the behavior of `breakpoint()` can be changed with the `PYTHONBREAKPOINT` environment variable. See `sys.breakpointhook()` for usage details.

Note that this is not guaranteed if `sys.breakpointhook()` has been replaced.

`breakpointhook`을 인자로 감사 이벤트(*auditing event*) `builtins.breakpoint`를 발생시킵니다.

Added in version 3.7.

class bytearray(*source=b''*)

class bytearray(*source, encoding, errors='strict'*)

새로운 바이트 배열을 돌려줍니다. `bytearray` 클래스는 0 <= x < 256 범위에 있는 정수의 가변 시퀀스입니다. `bytes` 형이 가진 대부분의 메서드뿐만 아니라 (바이트열 과 바이트 배열 연산 를 보세요), 가변 시퀀스 형 에 기술된 가변 시퀀스의 일반적인 메서드 대부분을 갖고 있습니다.

선택적 *source* 매개변수는 몇 가지 다른 방법으로 배열을 초기화하는 데 사용할 수 있습니다:

- 문자열 이면, 반드시 *encoding* 매개변수도 제공해야 합니다 (그리고 선택적으로 *errors* 도); 그러면 `bytearray()` 는 `str.encode()` 를 사용하여 문자열을 바이트로 변환합니다.
- 정수 면, 배열은 그 크기를 갖고, 널 바이트로 초기화됩니다.
- 버퍼 인터페이스를 제공하는 객체면, 객체의 읽기 전용 버퍼가 바이트 배열을 초기화하는 데 사용됩니다.
- 이터러블 이면, 범위 `0 <= x < 256` 의 정수를 제공하는 이터러블이어야 하고, 그 값들이 배열의 초기 내용물로 사용됩니다.

인자가 없으면 크기 0의 배열이 만들어집니다.

바이너리 시퀀스 형 — *bytes, bytearray, memoryview*와 바이트 배열 객체 도 보세요.

class bytes(*source=b''*)

class bytes(*source, encoding, errors='strict'*)

Return a new "bytes" object which is an immutable sequence of integers in the range `0 <= x < 256`. `bytes` is an immutable version of `bytearray` – it has the same non-mutating methods and the same indexing and slicing behavior.

따라서 생성자 인자는 `bytearray()` 와 같이 해석됩니다.

바이트열 객체는 리터럴을 사용하여 만들 수도 있습니다 (strings 를 보세요).

바이너리 시퀀스 형 — *bytes, bytearray, memoryview*, 바이트열 객체 및 바이트열 과 바이트 배열 연산 도 보세요.

callable(*object, /*)

Return `True` if the *object* argument appears callable, `False` if not. If this returns `True`, it is still possible that a call fails, but if it is `False`, calling *object* will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a `__call__()` method.

Added in version 3.2: 이 함수는 파이썬 3.0에서 먼저 제거된 다음 파이썬 3.2에서 다시 도입했습니다.

chr(*codepoint, /*)

Return the string representing a character with the specified Unicode code point. For example, `chr(97)` returns the string `'a'`, while `chr(8364)` returns the string `'€'`. This is the inverse of `ord()`.

The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16). `ValueError` will be raised if it is outside that range.

@classmethod

메서드를 클래스 메서드로 변환합니다.

A class method receives the class as an implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C:
    @classmethod
    def f(cls, arg1, arg2): ...
```

`@classmethod` 형식은 함수 데코레이터 입니다 – 자세한 내용은 function를 보세요.

클래스 메서드는 클래스 (`C.f()` 처럼) 또는 인스턴스 (`C().f()` 처럼) 를 통해 호출할 수 있습니다. 인스턴스는 클래스만 참조하고 무시됩니다. 파생 클래스에 대해 클래스 메서드가 호출되면, 파생 클래스 객체가 묵시적인 첫 번째 인자로 전달됩니다.

클래스 메서드는 C++ 또는 자바의 정적 메서드와 다릅니다. 그것들을 원하면, 이 섹션의 `staticmethod()`를 보세요. 클래스 메서드에 대한 더 자세한 정보는, types을 참고하세요.

버전 3.9에서 변경: 클래스 메서드는 이제 `property()`와 같은 다른 디스크립터를 래핑 할 수 있습니다.

버전 3.10에서 변경: Class methods now inherit the method attributes (`__module__`, `__name__`, `__qualname__`, `__doc__` and `__annotations__`) and have a new `__wrapped__` attribute.

Deprecated since version 3.11, removed in version 3.13: Class methods can no longer wrap other *descriptors* such as `property()`.

compile(*source, filename, mode, flags=0, dont_inherit=False, optimize=-1*)

source 를 코드 또는 AST 객체로 컴파일합니다. 코드 객체는 `exec()` 또는 `eval()` 로 실행할 수 있습니다. *source* 는 일반 문자열, 바이트열 또는 AST 객체 일 수 있습니다. AST 객체로 작업하는 방법에 대한 정보는 `ast` 모듈 문서를 참조하세요.

filename 인자는 코드를 읽은 파일을 제공해야 합니다; 파일에서 읽지 않으면 인식 가능한 값을 전달합니다 (`'<string>'` 이 일반적으로 사용됩니다).

mode 인자는 컴파일해야 하는 코드 종류를 지정합니다; *source* 가 문장의 시퀀스로 구성되어 있다면 `exec`, 단일 표현식으로 구성되어 있다면 `'eval'`, 단일 대화형 문장으로 구성되면 `'single'` 이 될 수 있습니다 (마지막의 경우 `None` 이외의 값으로 구해지는 표현식 문은 인쇄됩니다).

선택적 인자 *flags* 와 *dont_inherit* 는 어떤 컴파일러 옵션이 활성화되어야 하고 어떤 퓨처 기능이 허락되어야 하는지 제어합니다. 둘 다 제공되지 않는 경우 (또는 둘 다 0의 경우), 코드는 `compile()` 을 호출하는 코드에 적용되고 있는 것과 같은 플래그로 컴파일됩니다. *flags* 인자가 주어지고, *dont_inherit* 가 없으면 (또는 0) 원래 사용될 것에 더해 *flags* 인자로 지정된 컴파일러 옵션과 퓨처 문이 사용됩니다. *dont_inherit* 가 0이 아닌 정수면 *flags* 인자가 사용됩니다 – 둘러싼 코드의 플래그(퓨처 기능과 컴파일러 옵션)는 무시됩니다.

컴파일러 옵션과 퓨처 문은 여러 개의 옵션을 지정하기 위해 비트 OR 될 수 있는 비트에 의해 지정됩니다. 주어진 퓨처 기능을 지정하는 데 필요한 비트 필드는 `__future__` 모듈의 `_Feature` 인스턴스에서 `compiler_flag` 어트리뷰트로 찾을 수 있습니다. 컴파일러 플래그는 `PyCF_` 접두사로 `ast` 모듈에서 찾을 수 있습니다.

인자 *optimize* 는 컴파일러의 최적화 수준을 지정합니다; 기본값 `-1` 은 `-O` 옵션에 의해 주어진 인터프리터의 최적화 수준을 선택합니다. 명시적 수준은 `0` (최적화 없음, `__debug__` 이 참입니다), `1` (assert 가 제거됩니다, `__debug__` 이 거짓입니다) 또는 `2` 다 (독스트링도 제거됩니다).

이 함수는 컴파일된 소스가 올바르지 않으면 `SyntaxError` 를 일으키고, 소스에 널 바이트가 들어있는 경우 `ValueError` 를 일으킵니다.

파이썬 코드를 AST 표현으로 파싱하려면, `ast.parse()` 를 보세요.

`source`와 `filename`을 인자로 감사 이벤트(*auditing event*) `compile`을 발생시킵니다. 이 이벤트는 묵시적 컴파일로 인해 발생할 수도 있습니다.

> **참고**
>
> `'single'` 또는 `'eval'` mode로 여러 줄 코드를 가진 문자열을 컴파일할 때, 적어도 하나의 개행 문자로 입력을 끝내야 합니다. 이것은 `code` 모듈에서 문장이 불완전한지 완전한지를 쉽게 탐지하게 하기 위함입니다.

> **경고**
>
> 파이썬의 AST 컴파일러에서 스택 깊이 제한으로 인해, AST 객체로 컴파일할 때 충분히 크고 복잡한 문자열로 파이썬 인터프리터가 크래시를 일으키도록 만들 수 있습니다.

버전 3.2에서 변경: Allowed use of Windows and Mac newlines. Also, input in `'exec'` mode does not have to end in a newline anymore. Added the *optimize* parameter.

버전 3.5에서 변경: 이전에는, *source* 에서 널 바이트가 발견될 때 `TypeError` 가 발생했습니다.

Added in version 3.8: 이제 `ast.PyCF_ALLOW_TOP_LEVEL_AWAIT`를 flags로 전달하여 최상위 수준 `await`, `async for` 및 `async with`를 지원할 수 있습니다.

class complex(*number=0, /*)

class complex(*string, /*)

class complex(*real=0, imag=0*)

Convert a single string or number to a complex number, or create a complex number from real and imaginary parts.

Examples:

```
>>> complex('+1.23')
(1.23+0j)
>>> complex('-4.5j')
-4.5j
>>> complex('-1.23+4.5j')
(-1.23+4.5j)
>>> complex('\t( -1.23+4.5J )\n')
(-1.23+4.5j)
>>> complex('-Infinity+NaNj')
(-inf+nanj)
>>> complex(1.23)
(1.23+0j)
>>> complex(imag=-4.5)
-4.5j
>>> complex(-1.23, 4.5)
(-1.23+4.5j)
```

If the argument is a string, it must contain either a real part (in the same format as for `float()`) or an imaginary part (in the same format but with a `'j'` or `'J'` suffix), or both real and imaginary parts (the sign of the imaginary part is mandatory in this case). The string can optionally be surrounded by whitespaces and the round parentheses `'('` and `')'`, which are ignored. The string must not contain whitespace between `'+'`, `'-'`, the `'j'` or `'J'` suffix, and the decimal number. For example, `complex('1+2j')` is fine, but `complex('1 + 2j')` raises `ValueError`. More precisely, the input must conform to the `complexvalue` production rule in the following grammar, after parentheses and leading and trailing whitespace characters are removed:

```
complexvalue: floatvalue |
              floatvalue ("j" | "J") |
              floatvalue sign absfloatvalue ("j" | "J")
```

If the argument is a number, the constructor serves as a numeric conversion like `int` and `float`. For a general Python object `x`, `complex(x)` delegates to `x.__complex__()`. If `__complex__()` is not defined then it falls back to `__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

If two arguments are provided or keyword arguments are used, each argument may be any numeric type (including complex). If both arguments are real numbers, return a complex number with the real component *real* and the imaginary component *imag*. If both arguments are complex numbers, return a complex number with the real component `real.real-imag.imag` and the imaginary component `real.imag+imag.real`. If one of arguments is a real number, only its real component is used in the above expressions.

See also `complex.from_number()` which only accepts a single numeric argument.

If all arguments are omitted, returns `0j`.

복소수 형은 숫자 형 — *int, float, complex* 에서 설명합니다.

버전 3.6에서 변경: 코드 리터럴 처럼 숫자를 밑줄로 그룹화할 수 있습니다.

버전 3.8에서 변경: Falls back to `__index__()` if `__complex__()` and `__float__()` are not defined.

버전 3.14부터 폐지됨: Passing a complex number as the *real* or *imag* argument is now deprecated; it should only be passed as a single positional argument.

delattr(*object, name, /*)

This is a relative of `setattr()`. The arguments are an object and a string. The string must be the name of one of the object' s attributes. The function deletes the named attribute, provided the object allows it. For example, `delattr(x, 'foobar')` is equivalent to `del x.foobar`. *name* need not be a Python identifier (see `setattr()`).

class dict(***kwargs*)

class dict(*mapping, /, **kwargs*)

class dict(*iterable, /, **kwargs*)

새 딕셔너리를 만듭니다. `dict` 객체는 딕셔너리 클래스입니다. 이 클래스에 대한 설명서는 `dict` 및 매핑 형 — *dict* 을 보세요.

다른 컨테이너의 경우 `list`, `set` 및 `tuple` 클래스와 `collections` 모듈을 보세요.

dir()

dir(*object, /*)

인자가 없으면, 현재 지역 스코프에 있는 이름들의 리스트를 돌려줍니다. 인자가 있으면, 해당 객체에 유효한 어트리뷰트들의 리스트를 돌려주려고 시도합니다.

If the object has a method named `__dir__()`, this method will be called and must return the list of attributes. This allows objects that implement a custom `__getattr__()` or `__getattribute__()` function to customize the way `dir()` reports their attributes.

If the object does not provide `__dir__()`, the function tries its best to gather information from the object' s `__dict__` attribute, if defined, and from its type object. The resulting list is not necessarily complete and may be inaccurate when the object has a custom `__getattr__()`.

기본 `dir()` 메커니즘은 다른 형의 객체에 대해서 다르게 동작하는데, 완전한 정보보다는 가장 적절한 정보를 만들려고 시도하기 때문입니다:

- 객체가 모듈 객체면, 리스트에는 모듈 어트리뷰트의 이름이 포함됩니다.
- 객체가 형 또는 클래스 객체면, 리스트에는 그것의 어트리뷰트 이름과 베이스의 어트리뷰트 이름들이 재귀적으로 포함됩니다.
- 그 밖의 경우, 리스트에는 객체의 어트리뷰트 이름, 해당 클래스의 어트리뷰트 이름 및 해당 클래스의 베이스 클래스들의 어트리뷰트 이름을 재귀적으로 포함합니다.

결과 리스트는 알파벳 순으로 정렬됩니다. 예를 들어:

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
>>> s = Shape()
>>> dir(s)
['area', 'location', 'perimeter']
```

> **참고**
>
> `dir()` 은 주로 대화형 프롬프트에서의 사용 편의를 위해 제공되기 때문에, 엄격하거나 일관되게 정의된 이름 집합을 제공하기보다 흥미로운 이름 집합을 제공하려고 시도하며, 상세한 동작은 배포마다 변경될 수 있습니다. 예를 들어, 인자가 클래스면 메타 클래스 어트리뷰트는 결과 리스트에 없습니다.

divmod(*a*, *b*, /)

Take two (non-complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division. With mixed operand types, the rules for binary arithmetic operators apply. For integers, the result is the same as `(a // b, a % b)`. For floating-point numbers the result is `(q, a % b)`, where *q* is usually `math.floor(a / b)` but may be 1 less than that. In any case `q * b + a % b` is very close to *a*, if `a % b` is non-zero it has the same sign as *b*, and `0 <= abs(a % b) < abs(b)`.

enumerate(*iterable*, *start=0*)

열거 객체를 돌려줍니다. *iterable* 은 시퀀스, 이터레이터 또는 이터레이션을 지원하는 다른 객체여야 합니다. `enumerate()` 에 의해 반환된 이터레이터의 `__next__()` 메서드는 카운트 (기본값 0을 갖는 *start* 부터)와 *iterable* 을 이터레이션 해서 얻어지는 값을 포함하는 튜플을 돌려줍니다.

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

다음과 동등합니다:

```
def enumerate(iterable, start=0):
    n = start
    for elem in iterable:
        yield n, elem
        n += 1
```

eval(*source*, /, *globals=None*, *locals=None*)

매개변수

- **source** (*str* | code object) – A Python expression.
- **globals** (*dict* | None) – The global namespace (default: None).
- **locals** (*mapping* | None) – The local namespace (default: None).

반환

The result of the evaluated expression.

Raises

Syntax errors are reported as exceptions.

> **경고**
>
> This function executes arbitrary code. Calling it with user-supplied input may lead to security vulnerabilities.

The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* mappings as global and local namespace. If the *globals* dictionary is present and does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key before *expression* is parsed. That way you can control what builtins are available to the executed code by inserting your own `__builtins__` dictionary into *globals* before passing it to `eval()`. If the *locals* mapping is omitted it defaults to the *globals* dictionary. If both mappings are omitted, the expression is executed with the *globals* and *locals* in the environment where `eval()` is called. Note, *eval*() will only have access to the *nested scopes* (non-locals) in the enclosing environment if they are already referenced in the scope that is calling `eval()` (e.g. via a `nonlocal` statement).

Example:

```
>>> x = 1
>>> eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as those created by `compile()`). In this case, pass a code object instead of a string. If the code object has been compiled with `'exec'` as the *mode* argument, `eval()`'s return value will be `None`.

Hints: dynamic execution of statements is supported by the `exec()` function. The `globals()` and `locals()` functions return the current global and local dictionary, respectively, which may be useful to pass around for use by `eval()` or `exec()`.

If the given source is a string, then leading and trailing spaces and tabs are stripped.

리터럴 만 포함 된 표현식의 값을 안전하게 구할 수 있는 함수 `ast.literal_eval()` 를 보세요.

코드 객체를 인자로 감사 이벤트(*auditing event*) `exec`를 발생시킵니다. 코드 컴파일 이벤트도 발생할 수 있습니다.

버전 3.13에서 변경: The *globals* and *locals* arguments can now be passed as keywords.

버전 3.13에서 변경: The semantics of the default *locals* namespace have been adjusted as described for the `locals()` builtin.

exec(*source*, /, *globals=None*, *locals=None*, * (*Keyword-only parameters separator* (*PEP 3102*)), *closure=None*)

> **경고**
>
> This function executes arbitrary code. Calling it with user-supplied input may lead to security vulnerabilities.

This function supports dynamic execution of Python code. *source* must be either a string or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs).[1] If it is a code object, it is simply executed. In all cases, the code that' s executed is expected to be valid as file input (see the section file-input in the Reference Manual). Be aware that the `nonlocal`, `yield`, and `return` statements may not be used outside of function definitions even within the context of code passed to the `exec()` function. The return value is `None`.

In all cases, if the optional parts are omitted, the code is executed in the current scope. If only *globals* is provided, it must be a dictionary (and not a subclass of dictionary), which will be used for both the global and the local variables. If *globals* and *locals* are given, they are used for the global and local variables, respectively. If provided, *locals* can be any mapping object. Remember that at the module level, globals and locals are the same dictionary.

> **참고**

[1] 파서는 유닉스 스타일의 줄 종료 규칙만 받아들이는 것에 주의하세요. 파일에서 코드를 읽는 경우, 줄 넘김 변환 모드를 사용해서 윈도우나 맥 스타일 줄 넘김을 변환해야 합니다.

> When `exec` gets two separate objects as *globals* and *locals*, the code will be executed as if it were embedded in a class definition. This means functions and classes defined in the executed code will not be able to access variables assigned at the top level (as the "top level" variables are treated as class variables in a class definition).

globals 딕셔너리가 `__builtins__` 를 키로 하는 값을 갖고 있지 않으면, 그 키로 내장 모듈 `builtins` 에 대한 참조가 삽입됩니다. 이런 식으로 `exec()` 에 전달하기 전에 *globals* 에 여러분 자신의 `__builtins__` 딕셔너리를 삽입함으로써, 실행되는 코드에 어떤 내장 객체들이 제공될지를 제어할 수 있습니다.

The *closure* argument specifies a closure–a tuple of cellvars. It' s only valid when the *object* is a code object containing *free (closure) variables*. The length of the tuple must exactly match the length of the code object' s `co_freevars` attribute.

코드 객체를 인자로 감사 이벤트(*auditing event*) `exec`를 발생시킵니다. 코드 컴파일 이벤트도 발생할 수 있습니다.

> **참고**
>
> The built-in functions `globals()` and `locals()` return the current global and local namespace, respectively, which may be useful to pass around for use as the second and third argument to `exec()`.

> **참고**
>
> The default *locals* act as described for function `locals()` below. Pass an explicit *locals* dictionary if you need to see effects of the code on *locals* after function `exec()` returns.

버전 3.11에서 변경: Added the *closure* parameter.

버전 3.13에서 변경: The *globals* and *locals* arguments can now be passed as keywords.

버전 3.13에서 변경: The semantics of the default *locals* namespace have been adjusted as described for the `locals()` builtin.

filter(*function, iterable, /*)

Construct an iterator from those elements of *iterable* for which *function* is true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *function* is `None`, the identity function is assumed, that is, all elements of *iterable* that are false are removed.

`filter(function, iterable)` 는 function이 `None` 이 아닐 때 제너레이터 표현식 `(item for item in iterable if function(item))` 과, `None` 일 때 `(item for item in iterable if item)` 와 동등함에 유의하세요.

See `itertools.filterfalse()` for the complementary function that returns elements of *iterable* for which *function* is false.

class float(*number=0.0, /*)

class float(*string, /*)

Return a floating-point number constructed from a number or a string.

Examples:

```
>>> float('+1.23')
1.23
>>> float('   -12345\n')
-12345.0
>>> float('1e-003')
0.001
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf
```

If the argument is a string, it should contain a decimal number, optionally preceded by a sign, and optionally embedded in whitespace. The optional sign may be `'+'` or `'-'`; a `'+'` sign has no effect on the value produced. The argument may also be a string representing a NaN (not-a-number), or positive or negative infinity. More precisely, the input must conform to the *floatvalue* production rule in the following grammar, after leading and trailing whitespace characters are removed:

```
sign:           "+" | "-"
infinity:       "Infinity" | "inf"
nan:            "nan"
digit:          <a Unicode decimal digit, i.e. characters in Unicode general category Nd>
digitpart:      digit (["_"] digit)*
number:         [digitpart] "." digitpart | digitpart ["."]
exponent:       ("e" | "E") [sign] digitpart
floatnumber:    number [exponent]
absfloatvalue:  floatnumber | infinity | nan
floatvalue:     [sign] absfloatvalue
```

Case is not significant, so, for example, "inf", "Inf", "INFINITY", and "iNfINity" are all acceptable spellings for positive infinity.

Otherwise, if the argument is an integer or a floating-point number, a floating-point number with the same value (within Python' s floating-point precision) is returned. If the argument is outside the range of a Python float, an `OverflowError` will be raised.

For a general Python object `x`, `float(x)` delegates to `x.__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

See also `float.from_number()` which only accepts a numeric argument.

인자가 주어지지 않으면, `0.0` 을 돌려줍니다.

float 형은 숫자 형 — *int, float, complex* 에 설명되어 있습니다.

버전 3.6에서 변경: 코드 리터럴 처럼 숫자를 밑줄로 그룹화할 수 있습니다.

버전 3.7에서 변경: The parameter is now positional-only.

버전 3.8에서 변경: Falls back to `__index__()` if `__float__()` is not defined.

format(*value, format_spec=", /*)

Convert a *value* to a "formatted" representation, as controlled by *format_spec*. The interpretation of *format_spec* will depend on the type of the *value* argument; however, there is a standard formatting syntax that is used by most built-in types: 포맷 명세 미니 언어.

기본 *format_spec* 은 빈 문자열이며 일반적으로 `str(value)` 를 호출하는 것과 같은 효과를 줍니다.

A call to `format(value, format_spec)` is translated to `type(value).__format__(value, format_spec)` which bypasses the instance dictionary when searching for the value' s `__format__()` method. A `TypeError` exception is raised if the method search reaches `object` and the *format_spec* is non-empty, or if either the *format_spec* or the return value are not strings.

버전 3.4에서 변경: `object().__format__(format_spec)` 은 *format_spec* 이 빈 문자열이 아닌 경우 `TypeError` 를 일으킵니다.

class frozenset(*iterable=(), /*)

새 `frozenset` 객체를 돌려주는데, 선택적으로 *iterable* 에서 가져온 요소를 포함합니다. `frozenset` 은 내장 클래스입니다. 이 클래스에 대한 설명서는 `frozenset` 과 집합 형 — *set, frozenset*을 보세요.

다른 컨테이너의 경우 `set`, `list`, `tuple` 및 `dict` 클래스와 `collections` 모듈을 보세요.

getattr(*object, name, /*)

getattr(*object, name, default, /*)

Return the value of the named attribute of *object*. *name* must be a string. If the string is the name of one of the object' s attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is equivalent to `x.foobar`. If the named attribute does not exist, *default* is returned if provided, otherwise `AttributeError` is raised. *name* need not be a Python identifier (see `setattr()`).

> **참고**
>
> Since private name mangling happens at compilation time, one must manually mangle a private attribute' s (attributes with two leading underscores) name in order to retrieve it with `getattr()`.

globals()

Return the dictionary implementing the current module namespace. For code within functions, this is set when the function is defined and remains the same regardless of where the function is called.

hasattr(*object, name, /*)

인자는 객체와 문자열입니다. 문자열이 객체의 속성 중 하나의 이름이면 결과는 `True` 이고, 그렇지 않으면 `False` 가 됩니다. (이것은 `getattr(object, name)` 을 호출하고 `AttributeError` 를 발생시키는지를 보는 식으로 구현됩니다.)

hash(*object, /*)

객체의 해시값을 돌려줍니다 (해시가 있는 경우). 해시값은 정수다. 딕셔너리 조회 중에 딕셔너리 키를 빨리 비교하는 데 사용됩니다. 같다고 비교되는 숫자 값은 같은 해시값을 갖습니다 (1과 1.0의 경우와 같이 형이 다른 경우조차도 그렇습니다).

> **참고**
>
> For objects with custom `__hash__()` methods, note that `hash()` truncates the return value based on the bit width of the host machine.

help()

help(*request*)

내장 도움말 시스템을 호출합니다. (이 함수는 대화형 사용을 위한 것입니다.) 인자가 제공되지 않으면, 인터프리터 콘솔에서 대화형 도움말 시스템이 시작됩니다. 인자가 문자열이면 문자열은 모듈, 함수, 클래스, 메서드, 키워드 또는 설명서 주제의 이름으로 조회되고, 도움말 페이지가 콘솔에 인쇄됩니다. 인자가 다른 종류의 객체면, 객체에 대한 도움말 페이지가 만들어집니다.

Note that if a slash(/) appears in the parameter list of a function when invoking `help()`, it means that the parameters prior to the slash are positional-only. For more info, see the FAQ entry on positional-only parameters.

이 함수는 `site` 모듈에 의해 내장 이름 공간에 추가됩니다.

버전 3.4에서 변경: `pydoc` 과 `inspect` 의 변경 사항은 콜러블의 시그니처가 이제 더 포괄적이고 일관성이 있음을 의미합니다.

hex(*integer, /*)

Convert an integer number to a lowercase hexadecimal string prefixed with "0x". If *integer* is not a Python `int` object, it has to define an `__index__()` method that returns an integer. Some examples:

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

정수를 대문자 또는 소문자 16진수로, 접두사가 있거나 없는 형태로 변환하려면 다음 방법의 하나를 사용할 수 있습니다:

```
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
>>> f'{255:#x}', f'{255:x}', f'{255:X}'
('0xff', 'ff', 'FF')
```

자세한 내용은 `format()`을 보세요.

16진수 문자열을 진수 16을 사용해서 정수로 변환하려면 `int()` 도 보세요.

> **참고**
>
> float에 대한 16진수 문자열 표현을 얻으려면, `float.hex()` 메서드를 사용하세요.

id(*object, /*)

객체의 "아이덴티티"를 돌려준다. 이것은 객체의 수명 동안 유일하고 바뀌지 않음이 보장되는 정수입니다. 수명이 겹치지 않는 두 개의 객체는 같은 `id()` 값을 가질 수 있습니다.

CPython 구현 상세: This is the address of the object in memory.

`id` 인자로 감사 이벤트(*auditing event*) `builtins.id`를 발생시킵니다.

input()

input(*prompt, /*)

prompt 인자가 있으면, 끝에 개행 문자를 붙이지 않고 표준 출력에 씁니다. 그런 다음 함수는 입력에서 한 줄을 읽고, 문자열로 변환해서 (줄 끝의 줄 바꿈 문자를 제거한다) 돌려줍니다. EOF를 읽으면 `EOFError` 를 일으킵니다. 예:

```
>>> s = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

`readline` 모듈이 로드되었다면, `input()` 은 그것을 사용하여 정교한 줄 편집과 히스토리 기능을 제공합니다.

입력을 읽기 전에 `prompt` 인자로 감사 이벤트(*auditing event*) `builtins.input`을 발생시킵니다.

Raises an *auditing event* `builtins.input/result` with the result after successfully reading input.

class int(*number=0, /*)

class int(*string, /, base=10*)

Return an integer object constructed from a number or a string, or return `0` if no arguments are given.

Examples:

```
>>> int(123.45)
123
>>> int('123')
123
>>> int('   -12_345\n')
-12345
>>> int('FACE', 16)
64206
>>> int('0xface', 0)
64206
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> int('01110011', base=2)
115
```

If the argument defines `__int__()`, `int(x)` returns `x.__int__()`. If the argument defines `__index__()`, it returns `x.__index__()`. For floating-point numbers, this truncates towards zero.

If the argument is not a number or if *base* is given, then it must be a string, `bytes`, or `bytearray` instance representing an integer in radix *base*. Optionally, the string can be preceded by `+` or `-` (with no space in between), have leading zeros, be surrounded by whitespace, and have single underscores interspersed between digits.

A base-n integer string contains digits, each representing a value from 0 to n-1. The values 0–9 can be represented by any Unicode decimal digit. The values 10–35 can be represented by `a` to `z` (or `A` to `Z`). The default *base* is 10. The allowed bases are 0 and 2–36. Base-2, -8, and -16 strings can be optionally prefixed with `0b`/`0B`, `0o`/`0O`, or `0x`/`0X`, as with integer literals in code. For base 0, the string is interpreted in a similar way to an integer literal in code, in that the actual base is 2, 8, 10, or 16 as determined by the prefix. Base 0 also disallows leading zeros: `int('010', 0)` is not legal, while `int('010')` and `int('010', 8)` are.

정수 형은 숫자 형 — *int, float, complex* 에 설명되어 있습니다.

버전 3.4에서 변경: *base* 가 `int` 의 인스턴스가 아니고 *base* 객체가 `base.__index__` 메서드를 가지면, 그 진수로 쓸 정수를 얻기 위해 그 메서드를 호출합니다. 예전 버전에서는 `base.__index__` 대신에 `base.__int__` 가 사용되었습니다.

버전 3.6에서 변경: 코드 리터럴 처럼 숫자를 밑줄로 그룹화할 수 있습니다.

버전 3.7에서 변경: The first parameter is now positional-only.

버전 3.8에서 변경: Falls back to `__index__()` if `__int__()` is not defined.

버전 3.11에서 변경: `int` string inputs and string representations can be limited to help avoid denial of service attacks. A `ValueError` is raised when the limit is exceeded while converting a string to an `int` or when converting an `int` into a string would exceed the limit. See the *integer string conversion length limitation* documentation.

버전 3.14에서 변경: `int()` no longer delegates to the `__trunc__()` method.

isinstance(*object, classinfo, /*)

Return `True` if the *object* argument is an instance of the *classinfo* argument, or of a (direct, indirect, or *virtual*) subclass thereof. If *object* is not an object of the given type, the function always returns `False`. If *classinfo* is a tuple of type objects (or recursively, other such tuples) or a *Union Type* of multiple types, return `True` if *object* is an instance of any of the types. If *classinfo* is not a type or tuple of types and such tuples, a `TypeError` exception is raised. `TypeError` may not be raised for an invalid type if an earlier check succeeds.

버전 3.10에서 변경: *classinfo* can be a *Union Type*.

issubclass(*class, classinfo, /*)

Return `True` if *class* is a subclass (direct, indirect, or *virtual*) of *classinfo*. A class is considered a subclass of itself. *classinfo* may be a tuple of class objects (or recursively, other such tuples) or a *Union Type*, in which case return `True` if *class* is a subclass of any entry in *classinfo*. In any other case, a `TypeError` exception is raised.

버전 3.10에서 변경: *classinfo* can be a *Union Type*.

iter(*iterable, /*)

iter(*callable, sentinel, /*)

Return an *iterator* object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, the single argument must be a collection object which supports the *iterable* protocol (the `__iter__()` method), or it must support the sequence protocol (the `__getitem__()` method with integer arguments starting at `0`). If it does not support either of those protocols, `TypeError` is raised. If the second argument, *sentinel*, is given, then the first argument must be a callable object. The iterator created in this case will call *callable* with no arguments for each call to its `__next__()`

method; if the value returned is equal to *sentinel*, `StopIteration` will be raised, otherwise the value will be returned.

이터레이터 형 도 보세요.

두 번째 형태의 `iter()` 의 한가지 유용한 응용은 블록 리더를 만드는 것입니다. 예를 들어, 바이너리 데이터베이스 파일에서 파일의 끝까지 고정 폭 블록 읽기입니다:

```
from functools import partial
with open('mydata.db', 'rb') as f:
    for block in iter(partial(f.read, 64), b''):
        process_block(block)
```

len(*object, /*)

객체의 길이 (항목 수)를 돌려줍니다. 인자는 시퀀스 (문자열, 바이트열, 튜플, 리스트 또는 range 같은) 또는 컬렉션 (딕셔너리, 집합 또는 불변 집합 같은) 일 수 있습니다.

`len`은 `range(2 ** 100)`와 같이 `sys.maxsize`보다 긴 길이에서 `OverflowError`를 발생시킵니다.

class list(*iterable=(), /*)

함수이기보다, 리스트 와 시퀀스 형 — *list, tuple, range* 에 문서화 된 것처럼, `list` 는 실제로는 가변 시퀀스 형입니다.

locals()

Return a mapping object representing the current local symbol table, with variable names as the keys, and their currently bound references as the values.

At module scope, as well as when using `exec()` or `eval()` with a single namespace, this function returns the same namespace as `globals()`.

At class scope, it returns the namespace that will be passed to the metaclass constructor.

When using `exec()` or `eval()` with separate local and global arguments, it returns the local namespace passed in to the function call.

In all of the above cases, each call to `locals()` in a given frame of execution will return the *same* mapping object. Changes made through the mapping object returned from `locals()` will be visible as assigned, reassigned, or deleted local variables, and assigning, reassigning, or deleting local variables will immediately affect the contents of the returned mapping object.

In an *optimized scope* (including functions, generators, and coroutines), each call to `locals()` instead returns a fresh dictionary containing the current bindings of the function' s local variables and any nonlocal cell references. In this case, name binding changes made via the returned dict are *not* written back to the corresponding local variables or nonlocal cell references, and assigning, reassigning, or deleting local variables and nonlocal cell references does *not* affect the contents of previously returned dictionaries.

Calling `locals()` as part of a comprehension in a function, generator, or coroutine is equivalent to calling it in the containing scope, except that the comprehension' s initialised iteration variables will be included. In other scopes, it behaves as if the comprehension were running as a nested function.

Calling `locals()` as part of a generator expression is equivalent to calling it in a nested generator function.

버전 3.12에서 변경: The behaviour of `locals()` in a comprehension has been updated as described in **PEP 709**.

버전 3.13에서 변경: As part of **PEP 667**, the semantics of mutating the mapping objects returned from this function are now defined. The behavior in *optimized scopes* is now as described above. Aside from being defined, the behaviour in other scopes remains unchanged from previous versions.

map(*function, iterable, /, *iterables, strict=False*)

Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterables* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. If *strict* is `True` and one of the iterables is exhausted before the others, a `ValueError` is raised. For cases where the function inputs are already arranged into argument tuples, see `itertools.starmap()`.

버전 3.14에서 변경: Added the *strict* parameter.

max(*iterable*, *, *key=None*)

max(*iterable*, *, *default*, *key=None*)

max(*arg1*, *arg2*, **args*, *key=None*)

iterable 에서 가장 큰 항목이나 두 개 이상의 인자 중 가장 큰 것을 돌려줍니다.

하나의 위치 인자가 제공되면, 그것은 이터러블 이어야 합니다. iterable에서 가장 큰 항목을 돌려줍니다. 두 개 이상의 위치 인자가 제공되면, 위치 인자 중 가장 큰 것을 돌려줍니다.

선택적 키워드-전용 인자가 두 개 있습니다. *key* 인자는 `list.sort()` 에 사용되는 것처럼 단일 인자 순서 함수를 지정합니다. *default* 인자는 제공된 iterable이 비어있는 경우 돌려줄 객체를 지정합니다. iterable이 비어 있고 *default* 가 제공되지 않으면 `ValueError` 가 발생합니다.

여러 항목이 최댓값이면, 함수는 처음 만난 항목을 돌려줍니다. 이것은 `sorted(iterable, key=keyfunc, reverse=True)[0]` 와 `heapq.nlargest(1, iterable, key=keyfunc)` 같은 다른 정렬 안정성 보존 도구와 일관성을 유지합니다.

버전 3.4에서 변경: Added the *default* keyword-only parameter.

버전 3.8에서 변경: *key*는 `None` 일 수 있습니다.

class memoryview(*object*)

지정된 인자로부터 만들어진 "메모리 뷰" 객체를 돌려줍니다. 자세한 정보는 메모리 뷰 를 보세요.

min(*iterable*, *, *key=None*)

min(*iterable*, *, *default*, *key=None*)

min(*arg1*, *arg2*, **args*, *key=None*)

iterable 에서 가장 작은 항목이나 두 개 이상의 인자 중 가장 작은 것을 돌려줍니다.

하나의 위치 인자가 제공되면, 그것은 이터러블 이어야 합니다. iterable에서 가장 작은 항목을 돌려줍니다. 두 개 이상의 위치 인자가 제공되면, 위치 인자 중 가장 작은 것을 돌려줍니다.

선택적 키워드-전용 인자가 두 개 있습니다. *key* 인자는 `list.sort()` 에 사용되는 것처럼 단일 인자 순서 함수를 지정합니다. *default* 인자는 제공된 iterable이 비어있는 경우 돌려줄 객체를 지정합니다. iterable이 비어 있고 *default* 가 제공되지 않으면 `ValueError` 가 발생합니다.

여러 항목이 최솟값이면, 함수는 처음 만난 항목을 돌려줍니다. 이것은 `sorted(iterable, key=keyfunc)[0]` 와 `heapq.nsmallest(1, iterable, key=keyfunc)` 같은 다른 정렬 안정성 보존 도구와 일관성을 유지합니다.

버전 3.4에서 변경: Added the *default* keyword-only parameter.

버전 3.8에서 변경: *key*는 `None` 일 수 있습니다.

next(*iterator*, /)

next(*iterator*, *default*, /)

Retrieve the next item from the *iterator* by calling its `__next__()` method. If *default* is given, it is returned if the iterator is exhausted, otherwise `StopIteration` is raised.

class object

This is the ultimate base class of all other classes. It has methods that are common to all instances of Python classes. When the constructor is called, it returns a new featureless object. The constructor does not accept any arguments.

> **참고**
>
> `object` instances do *not* have `__dict__` attributes, so you can' t assign arbitrary attributes to an instance of `object`.

oct(*integer, /*)

Convert an integer number to an octal string prefixed with "0o". The result is a valid Python expression. If *integer* is not a Python `int` object, it has to define an `__index__()` method that returns an integer. For example:

```
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'
```

If you want to convert an integer number to an octal string either with the prefix "0o" or not, you can use either of the following ways.

```
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
>>> f'{10:#o}', f'{10:o}'
('0o12', '12')
```

자세한 내용은 `format()` 을 보세요.

open(*file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None*)

file 을 열고 해당 파일 객체 를 돌려줍니다. 파일을 열 수 없으면, `OSError` 가 발생합니다. 이 함수를 사용하는 방법에 대한 더 많은 예제는 tut-files를 참조하십시오.

file is a *path-like object* giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed unless *closefd* is set to `False`.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to `'r'` which means open for reading in text mode. Other common values are `'w'` for writing (truncating the file if it already exists), `'x'` for exclusive creation, and `'a'` for appending (which on *some* Unix systems, means that *all* writes append to the end of the file regardless of the current seek position). In text mode, if *encoding* is not specified the encoding used is platform-dependent: `locale.getencoding()` is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave *encoding* unspecified.) The available modes are:

문자	의미
`'r'`	읽기용으로 엽니다 (기본값)
`'w'`	쓰기용으로 엽니다, 파일을 먼저 자릅니다.
`'x'`	독점적인 파일 만들기용으로 엽니다, 이미 존재하는 경우에는 실패합니다.
`'a'`	open for writing, appending to the end of file if it exists
`'b'`	바이너리 모드
`'t'`	텍스트 모드 (기본값)
`'+'`	갱신(읽기 및 쓰기)용으로 엽니다

The default mode is `'r'` (open for reading text, a synonym of `'rt'`). Modes `'w+'` and `'w+b'` open and truncate the file. Modes `'r+'` and `'r+b'` open the file with no truncation.

개요 에서 언급했듯이, 파이썬은 바이너리와 텍스트 I/O를 구별합니다. 바이너리 모드 (*mode* 인자에 `'b'` 를 포함합니다)로 열린 파일은 내용을 디코딩 없이 `bytes` 객체로 돌려줍니다. 텍스트 모드 (기본값, 또는 *mode* 인자에 `'t'` 가 포함될 때)에서는, 파일의 내용이 `str`로 반환되는데, 바이트열이 플랫폼 의존적인 인코딩이나 주어진 *encoding* 을 사용해서 먼저 디코드 됩니다.

> **참고**
>
> 파이썬은 하위 운영 체제의 텍스트 파일 개념에 의존하지 않습니다. 모든 처리는 파이썬 자체에 의해 수행되므로 플랫폼에 독립적입니다.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable when writing in text mode), and an integer > 1 to indicate the size in bytes of a fixed-size chunk buffer. Note that specifying a buffer size this way applies for binary buffered I/O, but `TextIOWrapper` (i.e., files opened with `mode='r+'`) would have another buffering. To disable buffering in `TextIOWrapper`, consider using the `write_through` flag for `io.TextIOWrapper.reconfigure()`. When no *buffering* argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is `max(min(blocksize, 8 MiB), DEFAULT_BUFFER_SIZE)` when the device block size is available. On most systems, the buffer will typically be 128 kilobytes long.
- "대화형" 텍스트 파일 (`isatty()` 가 `True` 를 돌려주는 파일)은 줄 버퍼링을 사용합니다. 다른 텍스트 파일은 바이너리 파일에 대해 위에서 설명한 정책을 사용합니다.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent (whatever `locale.getencoding()` returns), but any *text encoding* supported by Python can be used. See the `codecs` module for the list of supported encodings.

errors 는 인코딩 및 디코딩 에러를 처리하는 방법을 지정하는 선택적 문자열입니다. 바이너리 모드에서는 사용할 수 없습니다. 다양한 표준 에러 처리기가 제공됩니다 (에러 처리기 에 나열됩니다). 하지만, `codecs.register_error()` 로 등록된 에러 처리기 이름 역시 사용할 수 있습니다. 표준 이름은 다음과 같습니다:

- `'strict'` 는 인코딩 에러가 있는 경우 `ValueError` 예외를 발생시킵니다. 기본값 `None` 은 같은 효과를 냅니다.
- `'ignore'` 는 에러를 무시합니다. 인코딩 에러를 무시하면 데이터가 손실될 수 있음에 주의하세요.
- `'replace'` 는 잘못된 데이터가 있는 자리에 대체 마커(`'?'` 와 같은)를 삽입합니다.
- `'surrogateescape'` will represent any incorrect bytes as low surrogate code units ranging from U+DC80 to U+DCFF. These surrogate code units will then be turned back into the same bytes when the `surrogateescape` error handler is used when writing data. This is useful for processing files in an unknown encoding.
- `'xmlcharrefreplace'` is only supported when writing to a file. Characters not supported by the encoding are replaced with the appropriate XML character reference `&#nnn;`.
- `'backslashreplace'` 는 잘못된 데이터를 파이썬의 역 슬래시 이스케이프 시퀀스로 대체합니다.
- `'namereplace'` (역시 파일에 쓸 때만 지원됩니다)는 지원되지 않는 문자를 `\N{...}` 이스케이프 시퀀스로 대체합니다.

newline determines how to parse newline characters from the stream. It can be `None`, `''`, `'\n'`, `'\r'`, and `'\r\n'`. It works as follows:

- 스트림에서 입력을 읽을 때, *newline* 이 `None` 이면, 유니버설 줄 넘김 모드가 활성화됩니다. 입력에 있는 줄은 `'\n'`, `'\r'` 또는 `'\r\n'` 로 끝날 수 있으며, 호출자에게 돌려주기 전에 모두 `'\n'` 로 변환됩니다. 그것이 `''` 이면, 유니버설 줄 넘김 모드가 활성화되지만, 줄 끝은 변환되지 않은 채로 호출자에게 반환됩니다. 다른 유효한 값이면, 입력 줄은 주어진 문자열로만 끝나며, 줄 끝은 변환되지 않은 채로 호출자에게 돌려줍니다.
- 스트림에 출력을 쓸 때, *newline* 이 `None` 이면, 모든 `'\n'` 문자는 시스템 기본 줄 구분자인 `os.linesep` 로 변환됩니다. *newline* 이 `''` 또는 `'\n'` 이면, 변환이 이루어지지 않습니다. *newline* 이 다른 유효한 값이면, 쓰이는 모든 `'\n'` 문자는 주어진 문자열로 변환됩니다.

If *closefd* is `False` and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given *closefd* must be `True` (the default); otherwise, an error will be raised.

콜러블을 *opener* 로 전달하여 커스텀 오프너를 사용할 수 있습니다. 파일 객체를 위한 하위 파일 디스크립터는 *opener* 를 (*file*, *flags*) 로 호출해서 얻습니다. *opener* 는 열린 파일 디스크립터를 반환해야 합니다 (*opener* 에 `os.open` 을 전달하는 것은 `None` 을 전달하는 것과 비슷한 기능을 수행하게 됩니다).

새로 만들어진 파일은 상속 불가능 합니다.

다음 예는 주어진 디렉터리에 상대적인 파일을 열기 위해 `os.open()` 함수의 *dir_fd* 매개변수를 사용합니다:

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spamspam.txt', file=f)
...
>>> os.close(dir_fd)  # don't leak a file descriptor
```

`open()` 함수에 의해 반환된 파일 객체 의 형은 모드에 의존합니다. `open()` 이 텍스트 모드('w', 'r', 'wt', 'rt', 등)로 파일을 여는 데 사용되면, `io.TextIOBase` 의 서브 클래스를 돌려줍니다 (구체적으로 `io.TextIOWrapper`). 버퍼링과 함께 바이너리 모드로 파일을 여는 데 사용되는 경우, 반환되는 클래스는 `io.BufferedIOBase` 의 서브 클래스입니다. 정확한 클래스는 다양합니다: 읽기 바이너리 모드에서는, `io.BufferedReader` 를 돌려줍니다; 쓰기 바이너리와 덧붙이기 바이너리 모드에서는, `io.BufferedWriter` 를 돌려주고, 읽기/쓰기 모드에서는, `io.BufferedRandom` 을 돌려줍니다. 버퍼링을 끄면, 날 스트림, `io.RawIOBase` 의 서브 클래스, `io.FileIO`, 을 돌려줍니다.

See also the file handling modules, such as `fileinput`, `io` (where `open()` is declared), `os`, `os.path`, `tempfile`, and `shutil`.

Raises an *auditing event* `open` with arguments `path`, `mode`, `flags`.

`mode`와 `flags` 인자는 원래 호출에서 수정되거나 추론되었을 수 있습니다.

버전 3.3에서 변경:

- *opener* 매개변수가 추가되었습니다.
- `'x'` 모드가 추가되었습니다.
- `IOError` 를 일으켜왔습니다. 이제는 `OSError` 의 별칭입니다.
- 독점적 파일 만들기 모드('x')로 여는 파일이 이미 존재하면, 이제 `FileExistsError` 를 일으킵니다.

버전 3.4에서 변경:

- 파일은 이제 상속 불가능합니다.

버전 3.5에서 변경:

- 시스템 호출이 인터럽트 되고 시그널 처리기가 예외를 발생시키지 않으면, 이 함수는 이제 `InterruptedError` 예외를 일으키는 대신 시스템 호출을 재시도합니다 (이유는 **PEP 475** 를 보세요).
- `'namereplace'` 오류 처리기가 추가되었습니다.

버전 3.6에서 변경:

- `os.PathLike` 를 구현하는 객체를 받아들이도록 지원이 추가되었습니다.
- 윈도우에서, 콘솔 버퍼를 열면 `io.FileIO` 가 아닌 `io.RawIOBase` 의 서브 클래스가 반환될 수 있습니다.

버전 3.11에서 변경: The `'U'` mode has been removed.

ord(*character*, /)

Return the ordinal value of a character.

If the argument is a one-character string, return the Unicode code point of that character. For example, `ord('a')` returns the integer `97` and `ord('€')` (Euro sign) returns `8364`. This is the inverse of `chr()`.

If the argument is a `bytes` or `bytearray` object of length 1, return its single byte value. For example, `ord(b'a')` returns the integer `97`.

pow(*base*, *exp*, *mod=None*)

base 의 *exp* 거듭제곱을 돌려줍니다; *mod* 가 있는 경우, *base* 의 *exp* 거듭제곱의 모듈로 *mod* 를 돌려줍니다 (`pow(base, exp) % mod` 보다 더 빠르게 계산됩니다). 두 개의 인자 형식인 `pow(base, exp)` 는 거듭제곱 연산자를 사용하는 것과 동등합니다: `base**exp`.

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For `int` operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `pow(10, 2)` returns `100`, but `pow(10, -2)` returns `0.01`. For a negative base of type `int` or `float` and a non-integral exponent, a complex result is delivered. For example, `pow(-9, 0.5)` returns a value close to `3j`. Whereas, for a negative base of type `int` or `float` with an integral exponent, a float result is delivered. For example, `pow(-9, 2.0)` returns `81.0`.

`int` 피연산자 *base* 및 *exp* 의 경우, *mod* 가 있으면, *mod* 도 정수 형이어야 하고 *mod* 는 0 이 아니어야 합니다. *mod* 가 있고 *exp* 가 음수면, *base* 는 *mod* 와 서로 소 (relatively prime) 여야 합니다. 이 경우, `pow(inv_base, -exp, mod)` 가 반환되며, 여기서 *inv_base* 는 *base* 모듈로 *mod* 의 역입니다.

다음은 `38` 모듈로 `97` 의 역을 계산하는 예입니다:

```
>>> pow(38, -1, mod=97)
23
>>> 23 * 38 % 97 == 1
True
```

버전 3.8 에서 변경: `int` 피연산자의 경우, `pow` 의 3 인자 형식은 이제 두 번째 인자가 음수가 되는 것을 허용하여, 모듈러 역수를 계산할 수 있게 합니다.

버전 3.8 에서 변경: 키워드 인자를 허용합니다. 이전에는, 위치 인자만 지원되었습니다.

print(**objects*, *sep=' '*, *end='\n'*, *file=None*, *flush=False*)

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file*, and *flush*, if present, must be given as keyword arguments.

모든 비 키워드 인자는 `str()` 이 하듯이 문자열로 변환된 후 스트림에 쓰이는데, *sep* 로 구분되고 *end* 를 뒤에 붙입니다. *sep* 과 *end* 는 모두 문자열이어야 합니다; `None` 일 수도 있는데, 기본값을 사용한다는 뜻입니다. *objects* 가 주어지지 않으면 `print()` 는 *end* 만 씁니다.

file 인자는 `write(string)` 메서드를 가진 객체여야 합니다; 존재하지 않거나 `None` 이면, `sys.stdout` 이 사용됩니다. 인쇄된 인자는 텍스트 문자열로 변환되기 때문에, `print()` 는 바이너리 모드 파일 객체와 함께 사용할 수 없습니다. 이를 위해서는. 대신 `file.write(...)` 를 사용합니다.

Output buffering is usually determined by *file*. However, if *flush* is true, the stream is forcibly flushed.

버전 3.3 에서 변경: *flush* 키워드 인자가 추가되었습니다.

class property(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

프로퍼티 어트리뷰트를 돌려줍니다.

fget 은 어트리뷰트 값을 얻는 함수입니다. *fset* 은 어트리뷰트 값을 설정하는 함수입니다. *fdel* 은 어트리뷰트 값을 삭제하는 함수입니다. 그리고 *doc* 은 어트리뷰트의 독스트링을 만듭니다.

전형적인 사용은 관리되는 어트리뷰트 `x` 를 정의하는 것입니다:

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

If *c* is an instance of *C*, `c.x` will invoke the getter, `c.x = value` will invoke the setter, and `del c.x` the deleter.

주어진 경우, *doc* 은 프로퍼티 어트리뷰트의 독스트링이 됩니다. 그렇지 않으면, *fget* 의 독스트링(있는 경우)이 복사됩니다. 이렇게 하면 `property()` 를 데코레이터 로 사용하여 읽기 전용 프로퍼티를 쉽게 만들 수 있습니다:

```
class Parrot:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

The `@property` decorator turns the `voltage()` method into a "getter" for a read-only attribute with the same name, and it sets the docstring for *voltage* to "Get the current voltage."

@getter

@setter

@deleter

A property object has `getter`, `setter`, and `deleter` methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

이 코드는 첫 번째 예제와 정확히 동등합니다. 추가적인 함수들에 원래 프로퍼티(이 경우 `x`)와 같은 이름을 사용해야 합니다.

반환된 프로퍼티 객체는 생성자 인자에 해당하는 `fget`, `fset` 및 `fdel` 어트리뷰트를 가집니다.

버전 3.5에서 변경: 이제 프로퍼티 개체의 독스트링이 쓰기 가능합니다.

__name__

Attribute holding the name of the property. The name of the property can be changed at runtime.

Added in version 3.13.

class range(*stop*, /)

class range(*start*, *stop*, *step=1*, /)

함수라기보다, `range` 는 실제로는 범위 와 시퀀스 형 — *list, tuple, range* 에 설명된 대로 불변 시퀀스 형입니다.

repr(*object*, /)

Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to `eval()`; otherwise, the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a `__repr__()` method. If `sys.displayhook()` is not accessible, this function will raise `RuntimeError`.

This class has a custom representation that can be evaluated:

```
class Person:
   def __init__(self, name, age):
      self.name = name
      self.age = age

   def __repr__(self):
      return f"Person('{self.name}', {self.age})"
```

reversed(*object*, /)

Return a reverse *iterator*. The argument must be an object which has a `__reversed__()` method or supports the sequence protocol (the `__len__()` method and the `__getitem__()` method with integer arguments starting at `0`).

round(*number*, *ndigits=None*)

number 를 소수점 다음에 *ndigits* 정밀도로 반올림한 값을 돌려줍니다. *ndigits* 가 생략되거나 `None` 이면, 입력에 가장 가까운 정수를 돌려줍니다.

For the built-in types supporting `round()`, values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done toward the even choice (so, for example, both `round(0.5)` and `round(-0.5)` are `0`, and `round(1.5)` is `2`). Any integer value is valid for *ndigits* (positive, zero, or negative). The return value is an integer if *ndigits* is omitted or `None`. Otherwise, the return value has the same type as *number*.

일반적인 파이썬 객체 `number` 의 경우, `round` 는 `number.__round__` 에 위임합니다.

> **참고**
>
> float에 대한 `round()` 의 동작은 예상과 다를 수 있습니다: 예를 들어, `round(2.675, 2)` 는 `2.68` 대신에 `2.67` 을 제공합니다. 이것은 버그가 아닙니다: 대부분의 십진 소수가 float로 정확히 표현될 수 없다는 사실로부터 오는 결과입니다. 자세한 정보는 tut-fp-issues 를 보세요.

class set(*iterable*=(), /)

새 `set` 객체를 돌려줍니다. 선택적으로 *iterable* 에서 가져온 요소를 갖습니다. `set` 은 내장 클래스입니다. 이 클래스에 대한 설명서는 `set` 및 집합 형 — *set, frozenset* 을 보세요.

다른 컨테이너의 경우 내장 `frozenset`, `list`, `tuple` 및 `dict` 클래스와 `collections` 모듈을 보세요.

setattr(*object*, *name*, *value*, /)

This is the counterpart of `getattr()`. The arguments are an object, a string, and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, `setattr(x, 'foobar', 123)` is equivalent to `x.foobar = 123`.

name need not be a Python identifier as defined in identifiers unless the object chooses to enforce that, for example in a custom `__getattribute__()` or via `__slots__`. An attribute whose name is not an identifier will not be accessible using the dot notation, but is accessible through `getattr()` etc..

> **참고**
>
> Since private name mangling happens at compilation time, one must manually mangle a private attribute's (attributes with two leading underscores) name in order to set it with `setattr()`.

class slice(*stop, /*)

class slice(*start, stop, step=None, /*)

Return a *slice* object representing the set of indices specified by `range(start, stop, step)`. The *start* and *step* arguments default to `None`.

Slice objects have read-only data attributes `start`, `stop`, and `step` which merely return the argument values (or their default). They have no other explicit functionality; however, they are used by NumPy and other third-party packages.

start

stop

step

Slice objects are also generated when extended indexing syntax is used. For example: `a[start:stop:step]` or `a[start:stop, i]`. See `itertools.islice()` for an alternate version that returns an *iterator*.

버전 3.12에서 변경: Slice objects are now *hashable* (provided `start`, `stop`, and `step` are hashable).

sorted(*iterable, /, *, key=None, reverse=False*)

iterable 의 항목들로 새 정렬된 리스트를 돌려줍니다.

키워드 인자로만 지정해야 하는 두 개의 선택적 인자가 있습니다.

key 는 하나의 인자를 받는 함수를 지정하는데, *iterable* 의 각 요소들로부터 비교 키를 추출하는 데 사용됩니다 (예를 들어, `key = str.lower`). 기본값은 `None` 입니다 (요소를 직접 비교합니다).

reverse 는 논리값입니다. `True` 로 설정되면, 각 비교가 뒤집힌 것처럼 리스트 요소들이 정렬됩니다.

예전 스타일의 *cmp* 함수를 *key* 함수로 변환하려면 `functools.cmp_to_key()` 를 사용하세요.

내장 `sorted()` 함수는 안정적(stable)임이 보장됩니다. 정렬은 같다고 비교되는 요소의 상대적 순서를 변경하지 않으면 안정적입니다 — 이는 여러 번 정렬할 때 유용합니다 (예를 들어, 부서별로 정렬한 후에 급여 등급별로 정렬하기).

The sort algorithm uses only < comparisons between items. While defining an `__lt__()` method will suffice for sorting, **PEP 8** recommends that all six rich comparisons be implemented. This will help avoid bugs when using the same data with other ordering tools such as `max()` that rely on a different underlying method. Implementing all six comparisons also helps avoid confusion for mixed type comparisons which can call reflected the `__gt__()` method.

정렬 예제와 간단한 정렬 자습서는 sortinghowto 를 보세요.

@staticmethod

메서드를 정적 메서드로 변환합니다.

정적 메서드는 묵시적인 첫 번째 인자를 받지 않습니다. 정적 메서드를 선언하려면, 이 관용구를 사용하세요:

```
class C:
    @staticmethod
    def f(arg1, arg2, argN): ...
```

`@staticmethod` 형식은 함수 데코레이터 입니다 – 자세한 내용은 function를 보세요.

A static method can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). Moreover, the static method *descriptor* is also callable, so it can be used in the class definition (such as `f()`).

Static methods in Python are similar to those found in Java or C++. Also, see `classmethod()` for a variant that is useful for creating alternate class constructors.

모든 데코레이터와 마찬가지로, `staticmethod` 를 정규 함수로 호출하여 그 결과로 어떤 일을 할 수도 있습니다. 이것은 클래스 바디에서 함수에 대한 참조가 필요하고 인스턴스 메서드로 자동 변환되는 것을 피하고자 할 때 필요합니다. 이 경우 다음 관용구를 사용하세요:

```
def regular_function():
    ...

class C:
    method = staticmethod(regular_function)
```

정적 메서드에 대한 더 자세한 정보는, types을 참조하세요.

버전 3.10에서 변경: Static methods now inherit the method attributes (`__module__`, `__name__`, `__qualname__`, `__doc__` and `__annotations__`), have a new `__wrapped__` attribute, and are now callable as regular functions.

class str(*, *encoding='utf-8'*, *errors='strict'*)

class str(*object*)

class str(*object*, *encoding*, *errors='strict'*)

class str(*object*, *, *errors*)

object 의 `str` 버전을 돌려줍니다. 자세한 내용은 `str()` 을 보세요.

`str` 은 내장 문자열 클래스 입니다. 문자열에 대한 일반적인 정보는 텍스트 시퀀스 형 — *str* 를 보세요.

sum(*iterable*, */*, *start=0*)

start 및 *iterable* 의 항목들을 왼쪽에서 오른쪽으로 합하고 합계를 돌려줍니다. *iterable* 의 항목은 일반적으로 숫자며 시작 값은 문자열이 될 수 없습니다.

For some use cases, there are good alternatives to `sum()`. The preferred, fast way to concatenate a sequence of strings is by calling `''.join(sequence)`. To add floating-point values with extended precision, see `math.fsum()`. To concatenate a series of iterables, consider using `itertools.chain()`.

버전 3.8에서 변경: *start* 매개 변수는 키워드 인자로만 지정될 수 있습니다.

버전 3.12에서 변경: Summation of floats switched to an algorithm that gives higher accuracy and better commutativity on most builds.

버전 3.14에서 변경: Added specialization for summation of complexes, using same algorithm as for summation of floats.

class super

class super(*type*, *object_or_type=None*, */*)

메서드 호출을 *type* 의 부모나 형제 클래스에 위임하는 프락시 객체를 돌려줍니다. 이는 클래스에서 재정의된 상속 된 메서드를 액세스할 때 유용합니다.

The *object_or_type* determines the *method resolution order* to be searched. The search starts from the class right after the *type*.

For example, if `__mro__` of *object_or_type* is `D -> B -> C -> A -> object` and the value of *type* is `B`, then `super()` searches `C -> A -> object`.

The `__mro__` attribute of the class corresponding to *object_or_type* lists the method resolution search order used by both `getattr()` and `super()`. The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

두 번째 인자가 생략되면, 반환되는 슈퍼 객체는 연결되지 않았습니다(unbound). 두 번째 인자가 객체면, `isinstance(obj, type)` 는 참이어야 합니다. 두 번째 인자가 형이면, `issubclass(type2, type)` 는 참이어야 합니다 (이것은 클래스 메서드에 유용합니다).

When called directly within an ordinary method of a class, both arguments may be omitted ("zero-argument `super()`"). In this case, *type* will be the enclosing class, and *obj* will be the first argument of the immediately

enclosing function (typically `self`). (This means that zero-argument `super()` will not work as expected within nested functions, including generator expressions, which implicitly create nested functions.)

super 에는 두 가지 일반적인 사용 사례가 있습니다. 단일 상속 클래스 계층 구조에서는, *super* 를 사용하여 명시적으로 이름을 지정하지 않고 부모 클래스를 참조할 수 있으므로, 코드를 더 유지 관리하기 쉽게 만들 수 있습니다. 이 사용은 다른 프로그래밍 언어에서 *super* 를 쓰는 것과 매우 유사합니다.

The second use case is to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement "diamond diagrams" where multiple base classes implement the same method. Good design dictates that such implementations have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

두 경우 모두, 일반적인 슈퍼 클래스 호출은 이런 식입니다:

```
class C(B):
    def method(self, arg):
        super().method(arg)     # This does the same thing as:
                                # super(C, self).method(arg)
```

메서드 조회 외에, *super()* 는 어트리뷰트 조회에도 작동합니다. 한가지 가능한 사용 사례는 부모나 형제 클래스에 있는 디스크립터를 호출하는 것입니다.

Note that *super()* is implemented as part of the binding process for explicit dotted attribute lookups such as `super().__getitem__(name)`. It does so by implementing its own `__getattribute__()` method for searching classes in a predictable order that supports cooperative multiple inheritance. Accordingly, *super()* is undefined for implicit lookups using statements or operators such as `super()[name]`.

또한, 인자가 없는 형식을 제외하고는, *super()* 는 메서드 내부에서만 사용하도록 제한되지 않는다는 점에 유의하세요. 두 개의 인자 형식은 인자를 정확하게 지정하고 적절한 참조를 만듭니다. 인자가 없는 형식은 클래스 정의 내에서만 작동하는데, 컴파일러가 정의되고 있는 클래스를 올바르게 가져오고 일반 메서드에서 현재 인스턴스에 액세스하는 데 필요한 세부 정보를 채우기 때문입니다.

super() 를 사용하여 협력적 클래스를 설계하는 방법에 대한 실용적인 제안은 super() 사용 안내 를 보세요.

버전 3.14에서 변경: *super* objects are now *pickleable* and *copyable*.

class tuple(*iterable*=(), /)

함수이기보다, *tuple* 은 실제로 튜플 과 시퀀스 형 — *list, tuple, range* 에 문서화 된 것처럼 불변 시퀀스 형입니다.

class type(*object*, /)

class type(*name*, *bases*, *dict*, /, ***kwargs*)

With one argument, return the type of an *object*. The return value is a type object and generally the same object as returned by `object.__class__`.

객체의 형을 검사하는 데는 *isinstance()* 내장 함수가 권장되는데, 서브 클래스를 고려하기 때문입니다.

With three arguments, return a new type object. This is essentially a dynamic form of the `class` statement. The *name* string is the class name and becomes the `__name__` attribute. The *bases* tuple contains the base classes and becomes the `__bases__` attribute; if empty, *object*, the ultimate base of all classes, is added. The *dict* dictionary contains attribute and method definitions for the class body; it may be copied or wrapped before becoming the `__dict__` attribute. The following two statements create identical `type` objects:

```
>>> class X:
...     a = 1
...
>>> X = type('X', (), dict(a=1))
```

See also:

- Documentation on attributes and methods on classes.
- 형 객체

세 인자 형식에 제공된 키워드 인자는 클래스 정의의 (*metaclass* 를 제외한) 키워드와 같은 방식으로 적절한 메타 클래스 장치(일반적으로 `__init_subclass__()`)에 전달됩니다.

class-customization 도 보세요.

버전 3.6에서 변경: Subclasses of `type` which don't override `type.__new__` may no longer use the one-argument form to get the type of an object.

vars()

vars(*object, /*)

Return the `__dict__` attribute for a module, class, instance, or any other object with a `__dict__` attribute.

Objects such as modules and instances have an updateable `__dict__` attribute; however, other objects may have write restrictions on their `__dict__` attributes (for example, classes use a `types.MappingProxyType` to prevent direct dictionary updates).

Without an argument, `vars()` acts like `locals()`.

객체가 지정되었지만 `__dict__` 어트리뷰트가 없으면 `TypeError` 예외가 발생합니다 (예를 들어, 해당 클래스가 `__slots__` 어트리뷰트를 정의하면).

버전 3.13에서 변경: The result of calling this function without an argument has been updated as described for the `locals()` builtin.

zip(**iterables, strict=False*)

Iterate over several iterables in parallel, producing tuples with an item from each one.

Example:

```
>>> for item in zip([1, 2, 3], ['sugar', 'spice', 'everything nice']):
...     print(item)
...
(1, 'sugar')
(2, 'spice')
(3, 'everything nice')
```

More formally: `zip()` returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument iterables.

Another way to think of `zip()` is that it turns rows into columns, and columns into rows. This is similar to transposing a matrix.

`zip()` is lazy: The elements won't be processed until the iterable is iterated on, e.g. by a `for` loop or by wrapping in a `list`.

One thing to consider is that the iterables passed to `zip()` could have different lengths; sometimes by design, and sometimes because of a bug in the code that prepared these iterables. Python offers three different approaches to dealing with this issue:

- By default, `zip()` stops when the shortest iterable is exhausted. It will ignore the remaining items in the longer iterables, cutting off the result to the length of the shortest iterable:

```
>>> list(zip(range(3), ['fee', 'fi', 'fo', 'fum']))
[(0, 'fee'), (1, 'fi'), (2, 'fo')]
```

- `zip()` is often used in cases where the iterables are assumed to be of equal length. In such cases, it's recommended to use the `strict=True` option. Its output is the same as regular `zip()`:

```
>>> list(zip(('a', 'b', 'c'), (1, 2, 3), strict=True))
[('a', 1), ('b', 2), ('c', 3)]
```

Unlike the default behavior, it raises a `ValueError` if one iterable is exhausted before the others:

```
>>> for item in zip(range(3), ['fee', 'fi', 'fo', 'fum'], strict=True):
...     print(item)
...
(0, 'fee')
(1, 'fi')
(2, 'fo')
Traceback (most recent call last):
  ...
ValueError: zip() argument 2 is longer than argument 1
```

Without the `strict=True` argument, any bug that results in iterables of different lengths will be silenced, possibly manifesting as a hard-to-find bug in another part of the program.

- Shorter iterables can be padded with a constant value to make all the iterables have the same length. This is done by *itertools.zip_longest()*.

Edge cases: With a single iterable argument, *zip()* returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.

Tips and tricks:

- The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using `zip(*[iter(s)]*n, strict=True)`. This repeats the *same* iterator `n` times so that each output tuple has the result of `n` calls to the iterator. This has the effect of dividing the input into n-length chunks.
- *zip()* 을 `*` 연산자와 함께 쓰면 리스트를 unzip 할 수 있습니다:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> list(zip(x, y))
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True
```

버전 3.10에서 변경: Added the `strict` argument.

__import__(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)

> **참고**
>
> 이것은 *importlib.import_module()* 과 달리 일상적인 파이썬 프로그래밍에서는 필요하지 않은 고급 함수입니다.

이 함수는 `import` 문에 의해 호출됩니다. `import` 문의 의미를 변경하기 위해 대체할 수 있습니다 (*builtins* 모듈을 임포트하고 `builtins.__import__` 에 대입합니다). 그러나 그렇게 하지 말 것을 **강하게** 권고하는데, 보통 같은 목적을 달성하는데 임포트 훅(**PEP 302** 를 보세요)을 사용하는 것이 더 간단하고 기본 임포트 구현이 사용될 것이라고 가정하는 코드들과 문제를 일으키지 않기 때문입니다. *__import__()* 의 직접 사용 역시 피하고 *importlib.import_module()* 을 사용할 것을 권합니다.

The function imports the module *name*, potentially using the given *globals* and *locals* to determine how to interpret the name in a package context. The *fromlist* gives the names of objects or submodules that should be imported from the module given by *name*. The standard implementation does not use its *locals* argument at all and uses its *globals* only to determine the package context of the `import` statement.

level 은 절대 또는 상대 임포트를 사용할지를 지정합니다. `0` (기본값)은 오직 절대 임포트를 수행한다는 것을 의미합니다. 양수 값 *level* 은 *__import__()* 를 호출하는 모듈 디렉터리에 상대적으로 검색할 상위 디렉터리들의 개수를 가리킵니다 (자세한 내용은 **PEP 328**을 보세요).

name 변수가 `package.module` 형식일 때, 일반적으로 *name* 에 의해 명명된 모듈이 아니라, 최상위 패키지(첫 번째 점까지의 이름)가 반환됩니다. 그러나 비어 있지 않은 *fromlist* 인자가 주어지면 *name* 에 의해 명명된 모듈이 반환됩니다.

예를 들어, 문장 `import spam` 은 다음 코드를 닮은 바이트 코드를 생성합니다:

```
spam = __import__('spam', globals(), locals(), [], 0)
```

문장 `import spam.ham` 은 이런 호출로 이어집니다:

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

여기에서 *__import__()* 가 최상위 모듈을 돌려주는 것에 주목하세요. 이것이 `import` 문에 의해 이름에 연결되는 객체이기 때문입니다.

반면에, 문장 `from spam.ham import eggs, sausage as saus` 는 이런 결과를 줍니다:

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], 0)
eggs = _temp.eggs
saus = _temp.sausage
```

여기서 `spam.ham` 모듈이 *__import__()* 에서 반환됩니다. 이 객체로부터, 임포트할 이름들을 가져온 후 해당 이름들로 대입됩니다.

단순히 이름으로 모듈을 임포트 하기 원한다면 (잠재적으로 패키지 내에서), *importlib.import_module()* 을 사용하세요.

버전 3.3에서 변경: 음수 *level* 은 더 지원되지 않습니다 (기본값도 0으로 변경합니다).

버전 3.9에서 변경: 명령 줄 옵션 `-E` 나 `-I` 를 사용 중일 때, 환경 변수 `PYTHONCASEOK` 는 이제 무시됩니다.

CHAPTER 3

내장 상수

작은 개수의 상수가 내장 이름 공간에 있습니다. 그것들은:

False

bool 형의 거짓 값. `False` 에 대입할 수 없고 *SyntaxError* 를 일으킵니다.

True

bool 형의 참값. `True` 에 대입할 수 없고 *SyntaxError* 를 일으킵니다.

None

기본 인자가 함수에 전달되지 않을 때처럼, 값의 부재를 나타내는 데 자주 사용되는 객체. `None` 으로의 대입은 허락되지 않고 *SyntaxError* 를 일으킵니다. `None`은 *NoneType* 형의 유일한 인스턴스입니다.

NotImplemented

연산이 다른 형에 대해 구현되지 않았음을 나타내기 위해, 이 항 특수 메서드(예를 들어, `__eq__()`, `__lt__()`, `__add__()`, `__rsub__()` 등)가 돌려줘야 하는 특별한 값; 같은 목적으로 증분 이 항 특수 메서드(예를 들어, `__imul__()`, `__iand__()` 등)가 반환할 수 있습니다. 불리언 문맥에서 평가해서는 안됩니다. `NotImplemented`는 *types.NotImplementedType* 형의 유일한 인스턴스입니다.

> **참고**
>
> 이 항 (또는 증분) 메서드가 `NotImplemented` 를 반환하면 인터프리터는 다른 형(또는 연산자에 따라 다른 폴백)에서 뒤집힌 연산을 시도합니다. 모든 시도가 `NotImplemented` 를 반환하면, 인터프리터는 적절한 예외를 발생시킵니다. 부정확하게 `NotImplemented` 를 반환하면 오해의 소지가 있는 에러 메시지가 나오거나 파이썬 코드에 `NotImplemented` 값이 반환됩니다.
>
> 예는 산술 연산 구현을 보세요.

> **조심**
>
> `NotImplementedError` 와 `NotImplemented` 는 바꿔쓸 수 없습니다. 이 상수는 위에서 설명한 대로만 사용해야 합니다; 이 예외의 올바른 사용법에 대한 자세한 내용은 *NotImplementedError*를 참조하세요.

버전 3.9에서 변경: Evaluating `NotImplemented` in a boolean context was deprecated.

버전 3.14에서 변경: Evaluating `NotImplemented` in a boolean context now raises a *TypeError*. It previously evaluated to *True* and emitted a *DeprecationWarning* since Python 3.9.

Ellipsis

The same as the ellipsis literal "`...`", an object frequently used to indicate that something is omitted. Assignment to `Ellipsis` is possible, but assignment to `...` raises a *SyntaxError*. `Ellipsis` is the sole instance of the *types.EllipsisType* type.

__debug__

이 상수는 파이썬이 `-O` 옵션으로 시작되지 않았다면 참이 됩니다. `assert` 문도 볼 필요가 있습니다.

> **참고**
>
> *None*, *False*, *True* 그리고 *__debug__* 은 다시 대입할 수 없습니다 (이것들을 대입하면, 설사 어트리뷰트 이름으로 사용해도, *SyntaxError* 를 일으킵니다). 그래서 이것들은 "진짜" 상수로 간주 될 수 있습니다.

3.1 site 모듈에 의해 추가된 상수들

site 모듈(`-S` 명령행 옵션이 주어진 경우를 제외하고는, 시작할 때 자동으로 임포트 됩니다)은 내장 이름 공간에 여러 상수를 추가합니다. 대화형 인터프리터 셸에 유용하고 프로그램에서 사용해서는 안 됩니다.

quit(*code=None*)

exit(*code=None*)

Objects that when printed, print a message like "Use quit() or Ctrl-D (i.e. EOF) to exit", and when accessed directly in the interactive interpreter or called as functions, raise *SystemExit* with the specified exit code.

help

Object that when printed, prints the message "Type help() for interactive help, or help(object) for help about object.", and when accessed directly in the interactive interpreter, invokes the built-in help system (see *help()*).

copyright

credits

인쇄하거나 호출할 때, 각각 저작권 또는 크레딧 텍스트를 인쇄하는 객체입니다.

license

인쇄될 때 "Type license() to see the full license text" 와 같은 메시지를 인쇄하고, 호출될 때 전체 라이센스 텍스트를 페이지 생성기와 같은 방식(한 번에 한 화면씩)으로 표시하는 객체입니다.

CHAPTER 4

내장형

다음 섹션에서는 인터프리터에 내장된 표준형에 관해 설명합니다.

기본 내장 유형은 숫자, 시퀀스, 매핑, 클래스, 인스턴스 및 예외입니다.

일부 컬렉션 클래스는 가변입니다. 제자리에서 멤버를 추가, 삭제 또는 재배치하고 특정 항목을 반환하지 않는 메서드는 컬렉션 인스턴스 자체를 반환하지 않고 `None` 을 반환합니다.

일부 연산들은 여러 객체 형에서 지원됩니다; 특히 사실상 모든 객체를 동등 비교하고, 논리값을 검사하고, (*repr()* 함수 또는 약간 다른 *str()* 함수를 사용해서) 문자열로 변환할 수 있습니다. 두 번째 함수는 *print()* 함수로 객체를 쓸 때 묵시적으로 사용됩니다.

4.1 논리값 검사

모든 객체는 논리값을 검사할 수 있는데, `if` 또는 `while` 조건 또는 다음에 나오는 논리 연산의 피연산자로 사용될 수 있도록 합니다.

By default, an object is considered true unless its class defines either a `__bool__()` method that returns `False` or a `__len__()` method that returns zero, when called with the object.[1] Here are most of the built-in objects considered false:

- constants defined to be false: `None` and `False`
- 모든 숫자 형들의 영: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- 빈 시퀀스와 컬렉션: `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

논리값을 돌려주는 연산과 내장 함수는 달리 명시하지 않는 한 항상 거짓의 경우 `0` 이나 `False` 를, 참이면 `1` 이나 `True` 를 돌려줍니다. (중요한 예외: 논리 연산 `or` 와 `and` 는 항상 피연산자 중 하나를 돌려줍니다.)

4.2 논리 연산 — and, or, not

이것들은 우선순위에 따라 오름차순으로 정렬된 논리 연산들입니다:

[1] 이 특수 메서드에 대한 추가 정보는 파이썬 레퍼런스 설명서(customization)에서 찾을 수 있습니다.

연산	결과	노트
`x or y`	if *x* is true, then *x*, else *y*	(1)
`x and y`	*x**가 거짓이면 **x*, 그렇지 않으면 *y*	(2)
`not x`	*x* 가 거짓이면 `True`, 그렇지 않으면 `False`	(3)

노트:

(1) 이것은 단락-회로 연산자이므로 첫 번째 인자가 거짓일 때만 두 번째의 값을 구합니다.

(2) 이것은 단락-회로 연산자이므로 첫 번째 인자가 참일 때만 두 번째의 값을 구합니다.

(3) `not` 은 비논리 연산자들보다 낮은 우선순위를 갖습니다. 그래서, `not a == b` 는 `not (a == b)` 로 해석되고, `a == not b` 는 문법 오류입니다.

4.3 비교

파이썬에는 8 가지 비교 연산이 있습니다. 이들 모두는 같은 우선순위를 가집니다 (논리 연산보다는 높습니다). 비교는 임의로 연결될 수 있습니다; 예를 들어 `x < y <= z` 는 *y* 의 값을 한 번만 구한다는 점을 제외하고는 `x < y and y <= z` 와 동등합니다 (하지만 두 경우 모두 `x < y` 가 거짓으로 밝혀지면 *z* 의 값을 구하지 않습니다).

이 표는 비교 연산을 요약합니다:

연산	뜻
`<`	엄격히 작다
`<=`	작거나 같다
`>`	엄격히 크다
`>=`	크거나 같다
`==`	같다
`!=`	같지 않다
`is`	객체 아이덴티티
`is not`	부정된 객체 아이덴티티

서로 다른 숫자 형을 제외하고는 서로 다른 형의 객체들은 같다고 비교되지 않습니다. `==` 연산자는 항상 정의되지만, 일부 객체 형(예를 들어, 클래스 객체)의 경우 `is`와 동등합니다. `<`, `<=`, `>` 및 `>=` 연산자는 의미가 있는 경우에만 정의됩니다; 예를 들어, 인자 중 하나가 복소수이면 *`TypeError`* 예외가 발생합니다.

Non-identical instances of a class normally compare as non-equal unless the class defines the `__eq__()` method.

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines enough of the methods `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` (in general, `__lt__()` and `__eq__()` are sufficient, if you want the conventional meanings of the comparison operators).

`is` 와 `is not` 연산자의 동작은 사용자 정의할 수 없습니다; 또한 임의의 두 객체에 적용할 수 있으며 예외를 발생시키지 않습니다.

Two more operations with the same syntactic priority, `in` and `not in`, are supported by types that are *iterable* or implement the `__contains__()` method.

4.4 숫자 형 — `int`, `float`, `complex`

There are three distinct numeric types: *integers*, *floating-point numbers*, and *complex numbers*. In addition, Booleans are a subtype of integers. Integers have unlimited precision. Floating-point numbers are usually implemented using `double` in C; information about the precision and internal representation of floating-point numbers for the machine on which your program is running is available in *`sys.float_info`*. Complex numbers have a real and imaginary part, which are each a floating-point number. To extract these parts from a complex number *z*, use `z.real` and

`z.imag`. (The standard library includes the additional numeric types `fractions.Fraction`, for rationals, and `decimal.Decimal`, for floating-point numbers with user-definable precision.)

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including hex, octal and binary numbers) yield integers. Numeric literals containing a decimal point or an exponent sign yield floating-point numbers. Appending `'j'` or `'J'` to a numeric literal yields an imaginary number (a complex number with a zero real part) which you can add to an integer or float to get a complex number with real and imaginary parts.

생성자 `int()`, `float()`, `complex()`를 특정 형의 숫자를 만드는데 사용할 수 있습니다.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the "narrower" type is widened to that of the other, where integer is narrower than floating point. Arithmetic with complex and real operands is defined by the usual mathematical formula, for example:

```
x + complex(u, v) = complex(x + u, v)
x * complex(u, v) = complex(x * u, x * v)
```

A comparison between numbers of different types behaves as though the exact values of those numbers were being compared.[2]

(복소수를 제외한) 모든 숫자 형은 다음과 같은 연산들을 지원합니다 (연산의 우선순위는 operator-summary 를 참조하십시오):

연산	결과	노트	전체 문서
`x + y`	*x* 와 *y* 의 합		
`x - y`	*x* 와 *y* 의 차		
`x * y`	*x* 와 *y* 의 곱		
`x / y`	*x* 와 *y* 의 몫		
`x // y`	*x* 와 *y* 의 정수로 내림한 몫	(1) (2)	
`x % y`	`x / y` 의 나머지	(2)	
`-x`	음의 *x*		
`+x`	*x* 그대로		
`abs(x)`	*x* 의 절댓값 또는 크기		`abs()`
`int(x)`	정수로 변환된 *x*	(3) (6)	`int()`
`float(x)`	실수로 변환된 *x*	(4) (6)	`float()`
`complex(re, im)`	실수부 *re* 와 허수부 *im* 으로 구성된 복소수. *im* 의 기본값은 0 입니다.	(6)	`complex()`
`c.conjugate()`	복소수 *c* 의 켤레		
`divmod(x, y)`	쌍 `(x // y, x % y)`	(2)	`divmod()`
`pow(x, y)`	*x* 의 *y* 거듭제곱	(5)	`pow()`
`x ** y`	*x* 의 *y* 거듭제곱	(5)	

노트:

(1) Also referred to as integer division. For operands of type `int`, the result has type `int`. For operands of type `float`, the result has type `float`. In general, the result is a whole integer, though the result' s type is not necessarily `int`. The result is always rounded towards minus infinity: `1//2` is `0`, `(-1)//2` is `-1`, `1//(-2)` is `-1`, and `(-1)//(-2)` is `0`.

(2) 복소수에는 사용할 수 없습니다. 적절한 경우 `abs()`를 사용하여 실수로 변환하십시오.

(3) Conversion from `float` to `int` truncates, discarding the fractional part. See functions `math.floor()` and `math.ceil()` for alternative conversions.

(4) float는 또한 숫자가 아님(NaN)과 양 또는 음의 무한대를 나타내는 문자열 "nan"과 접두사 "+" 나 "-" 가 선택적으로 붙을 수 있는 "inf"를 받아들입니다.

[2] 결과적으로, 리스트 `[1, 2]` 는 `[1.0, 2.0]` 과 같다고 취급되고, 튜플도 마찬가지입니다.

(5) 파이썬은 프로그래밍 언어들에서 흔히 그렇듯이, 있는 것처럼 `pow(0, 0)` 와 `0 ** 0` 이 `1` 이 되도록 정의합니다.

(6) 받아들여지는 숫자 리터럴은 `0` 에서 `9` 까지 또는 모든 동등한 유니코드들을 (`Nd` 속성을 가진 코드 포인트들) 포함합니다.

See the Unicode Standard for a complete list of code points with the `Nd` property.

모든 `numbers.Real` 형 (`int` 와 `float`) 은 또한 다음과 같은 연산들을 포함합니다:

연산	결과
`math.trunc(x)`	*x* 는 `Integral` 로 잘립니다
`round(x[, n])`	*x* 를 *n* 자리로 반올림하는데, 절반 값은 짝수로 반올림합니다. *n* 을 생략하면 기본값은 0입니다.
`math.floor(x)`	가장 큰 `Integral` <= *x*
`math.ceil(x)`	가장 작은 `Integral` >= *x*

추가적인 숫자 연산은 `math`와 `cmath` 모듈을 보십시오.

4.4.1 정수 형에 대한 비트 연산

비트 연산은 정수에 대해서만 의미가 있습니다. 비트 연산의 결과는 무한한 부호 비트를 갖는 2의 보수로 수행되는 것처럼 계산됩니다.

이진 비트 연산의 우선순위는 모두 숫자 연산보다 낮고 비교보다 높습니다; 일항 연산 `~` 은 다른 일항 연산들 (`+` 와 `-`) 과 같은 우선순위를 가집니다.

이 표는 비트 연산을 나열하는데, 우선순위에 따라 오름차순으로 정렬되어 있습니다:

연산	결과	노트
`x | y`	*x* 와 *y* 의 비트별 *or*	(4)
`x ^ y`	*x* 와 *y* 의 비트별 배타적 *or* (*exclusive or*)	(4)
`x & y`	*x* 와 *y* 의 비트별 *and*	(4)
`x << n`	*x* 를 *n* 비트만큼 왼쪽으로 시프트	(1)(2)
`x >> n`	*x* 를 *n* 비트만큼 오른쪽으로 시프트	(1)(3)
`~x`	*x* 의 비트 반전	

노트:

(1) 음의 시프트 수는 허락되지 않고 `ValueError` 를 일으킵니다.

(2) *n* 비트만큼의 왼쪽 시프트는 `pow(2, n)` 를 곱하는 것과 동등합니다.

(3) *n* 비트만큼 오른쪽으로 시프트 하는 것은 `pow(2, n)` 로 정수 나눗셈(floor division)하는 것과 동등합니다.

(4) 무한한 부호 비트가 있는 것과 같은 결과를 얻으려면, 유한한 2의 보수 표현으로 적어도 하나의 추가적인 부호 확장 비트를 사용하여 (`1 + max(x.bit_length(), y.bit_length())` 이상의 작업 비트 폭) 이러한 계산을 수행하는 것으로 충분합니다.

4.4.2 정수 형에 대한 추가 메서드

int 형은 `numbers.Integral` 추상 베이스 클래스 를 구현합니다. 또한, 몇 가지 메서드를 더 제공합니다:

`int.bit_length()`

부호와 선행 0을 제외하고, 이진수로 정수를 나타내는 데 필요한 비트 수를 돌려줍니다:

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

좀 더 정확하게 말하자면, `x` 가 0이 아니면, `x.bit_length()` 는 `2**(k-1) <= abs(x) < 2**k` 를 만족하는 유일한 양의 정수 `k` 입니다. 동등하게, `abs(x)` 가 정확하게 반올림된 로그값을 가질 만큼 아주 작으면, `k = 1 + int(log(abs(x), 2))` 가 됩니다. `x` 가 0이면, `x.bit_length()` 는 0 을 돌려줍니다.

다음 코드와 동등합니다:

```
def bit_length(self):
    s = bin(self)       # binary representation:  bin(-37) --> '-0b100101'
    s = s.lstrip('-0b') # remove leading zeros and minus sign
    return len(s)       # len('100101') --> 6
```

Added in version 3.1.

`int.`**`bit_count`**`()`

Return the number of ones in the binary representation of the absolute value of the integer. This is also known as the population count. Example:

```
>>> n = 19
>>> bin(n)
'0b10011'
>>> n.bit_count()
3
>>> (-n).bit_count()
3
```

다음 코드와 동등합니다:

```
def bit_count(self):
    return bin(self).count("1")
```

Added in version 3.10.

`int.`**`to_bytes`**`(`*length=1*, *byteorder='big'*, **, signed=False*`)`

정수를 나타내는 바이트의 배열을 돌려줍니다.

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() + 7) // 8, byteorder='little')
b'\xe8\x03'
```

The integer is represented using *length* bytes, and defaults to 1. An `OverflowError` is raised if the integer is not representable with the given number of bytes.

The *byteorder* argument determines the byte order used to represent the integer, and defaults to `"big"`. If *byteorder* is `"big"`, the most significant byte is at the beginning of the byte array. If *byteorder* is `"little"`, the most significant byte is at the end of the byte array.

signed 인자는 정수를 표현하는데 2의 보수가 사용되는지를 결정합니다. *signed* 가 `False` 이고 음의 정수가 주어지면, `OverflowError` 가 일어납니다. *signed* 의 기본값은 `False` 입니다.

The default values can be used to conveniently turn an integer into a single byte object:

```
>>> (65).to_bytes()
b'A'
```

However, when using the default arguments, don' t try to convert a value greater than 255 or you' ll get an *OverflowError*.

다음 코드와 동등합니다:

```
def to_bytes(n, length=1, byteorder='big', signed=False):
    if byteorder == 'little':
        order = range(length)
    elif byteorder == 'big':
        order = reversed(range(length))
    else:
        raise ValueError("byteorder must be either 'little' or 'big'")

    return bytes((n >> i*8) & 0xff for i in order)
```

Added in version 3.2.

버전 3.11 에서 변경: Added default argument values for `length` and `byteorder`.

classmethod `int.`**`from_bytes`**(*bytes, byteorder='big', *, signed=False*)

주어진 바이트 배열로 표현되는 정수를 돌려줍니다.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

인자 *bytes* 는 바이트열류 객체 이거나 바이트를 생성하는 이터러블이어야 합니다.

The *byteorder* argument determines the byte order used to represent the integer, and defaults to `"big"`. If *byteorder* is `"big"`, the most significant byte is at the beginning of the byte array. If *byteorder* is `"little"`, the most significant byte is at the end of the byte array. To request the native byte order of the host system, use *sys.byteorder* as the byte order value.

signed 인자는 정수를 표현하는데 2의 보수가 사용되는지를 나타냅니다.

다음 코드와 동등합니다:

```
def from_bytes(bytes, byteorder='big', signed=False):
    if byteorder == 'little':
        little_ordered = list(bytes)
    elif byteorder == 'big':
        little_ordered = list(reversed(bytes))
    else:
        raise ValueError("byteorder must be either 'little' or 'big'")

    n = sum(b << i*8 for i, b in enumerate(little_ordered))
    if signed and little_ordered and (little_ordered[-1] & 0x80):
        n -= 1 << 8*len(little_ordered)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    return n
```

Added in version 3.2.

버전 3.11에서 변경: Added default argument value for `byteorder`.

`int.as_integer_ratio()`

Return a pair of integers whose ratio is equal to the original integer and has a positive denominator. The integer ratio of integers (whole numbers) is always the integer as the numerator and `1` as the denominator.

Added in version 3.8.

`int.is_integer()`

Returns `True`. Exists for duck type compatibility with *`float.is_integer()`*.

Added in version 3.12.

4.4.3 실수에 대한 추가 메서드

float 형은 *`numbers.Real` 추상 베이스 클래스* 를 구현합니다. 또한, float는 다음과 같은 추가 메서드를 갖습니다.

classmethod `float.from_number(x)`

Class method to return a floating-point number constructed from a number *x*.

If the argument is an integer or a floating-point number, a floating-point number with the same value (within Python' s floating-point precision) is returned. If the argument is outside the range of a Python float, an *`OverflowError`* will be raised.

For a general Python object `x`, `float.from_number(x)` delegates to `x.__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

Added in version 3.14.

`float.as_integer_ratio()`

Return a pair of integers whose ratio is exactly equal to the original float. The ratio is in lowest terms and has a positive denominator. Raises *`OverflowError`* on infinities and a *`ValueError`* on NaNs.

`float.is_integer()`

float 인스턴스가 정숫값을 가진 유한이면 `True` 를, 그렇지 않으면 `False` 를 돌려줍니다:

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

두 가지 메서드가 16진수 문자열과의 변환을 지원합니다. 파이썬의 float는 내부적으로 이진수로 저장되기 때문에 float를 십진수 문자열로 또는 그 반대로 변환하는 것은 보통 반올림 오류를 수반합니다. 이에 반해, 16진수 문자열은 부동 소수점 숫자의 정확한 표현과 지정을 가능하게 합니다. 이것은 디버깅 및 수치 작업에 유용할 수 있습니다.

`float.hex()`

부동 소수점의 16진수 문자열 표현을 돌려줍니다. 유한 부동 소수점의 경우, 이 표현은 항상 선행하는 `0x` 와 후행하는 `p` 와 지수를 포함합니다.

classmethod `float.fromhex(s)`

16진수 문자열 *s* 로 표현되는 float를 돌려주는 클래스 메서드. 문자열 *s* 는 앞뒤 공백을 가질 수 있습니다.

`float.hex()` 는 인스턴스 메서드인 반면, *`float.fromhex()`* 는 클래스 메서드임에 주의하세요.

16진수 문자열은 다음과 같은 형식을 취합니다:

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

선택적인 sign 은 + 나 - 가 될 수 있고, integer 와 fraction 은 16진수 문자열이고, exponent 는 선택적인 선행 부호가 붙을 수 있는 십진수입니다. 대소 문자는 중요하지 않으며 integer 나 fraction 중 어느 하나에 적어도 하나의 16진수가 있어야 합니다. 이 문법은 C99 표준의 6.4.4.2 절에 지정된 문법과 비슷하며, 자바 1.5 이상에서 사용되는 문법과도 비슷합니다. 특히, *float.hex()* 의 출력은 C 또는 자바 코드에서 16진수의 부동 소수점 리터럴로 사용할 수 있으며, C의 %a 포맷 문자나 자바의 Double.toHexString 가 만들어내는 16진수 문자열은 *float.fromhex()* 가 받아들입니다.

지수는 16진수가 아닌 십진수로 쓰이고, 숫자에 곱해지는 2의 거듭제곱을 제공한다는 점에 유의하십시오. 예를 들어, 16진수 문자열 0x3.a7p10 는 부동 소수점 숫자 (3 + 10./16 + 7./16**2) * 2.0**10 또는 3740.0 를 나타냅니다:

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

3740.0 에 역변환을 적용하면 같은 숫자를 나타내는 다른 16진수 문자열을 얻을 수 있습니다:

```
>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

4.4.4 Additional Methods on Complex

The complex type implements the *numbers.Complex abstract base class*. complex also has the following additional methods.

classmethod complex.**from_number**(*x*)

Class method to convert a number to a complex number.

For a general Python object x, complex.from_number(x) delegates to x.__complex__(). If __complex__() is not defined then it falls back to __float__(). If __float__() is not defined then it falls back to __index__().

Added in version 3.14.

4.4.5 숫자 형의 해싱

For numbers x and y, possibly of different types, it' s a requirement that hash(x) == hash(y) whenever x == y (see the __hash__() method documentation for more details). For ease of implementation and efficiency across a variety of numeric types (including *int*, *float*, *decimal.Decimal* and *fractions.Fraction*) Python' s hash for numeric types is based on a single mathematical function that' s defined for any rational number, and hence applies to all instances of *int* and *fractions.Fraction*, and all finite instances of *float* and *decimal.Decimal*. Essentially, this function is given by reduction modulo P for a fixed prime P. The value of P is made available to Python as the *modulus* attribute of *sys.hash_info*.

현재, 사용되는 소수는 32-비트 C long을 가진 기계에서는 P = 2**31 - 1 이고, 64-비트 C long을 가진 기계에서는 P = 2**61 - 1 입니다.

다음은 규칙에 대한 세부 사항입니다:

- x = m / n 이 음이 아닌 유리수이고 n 이 P 로 나눠지 않는다면, hash(x) 를 m * invmod(n, P) % P 로 정의합니다. 여기서 invmod(n, P) 는 n 의 모듈로 P 역수를 줍니다.
- x = m / n 이 음이 아닌 유리수이고 n 이 P 나눠면 (하지만 m 은 나눠지 않으면) n 은 모듈로 P 역수를 가지지 않고 위의 규칙은 적용되지 않습니다; 이 경우 hash(x) 를 상숫값 sys.hash_info.inf 로 정의합니다.
- x = m / n 이 음의 유리수이면 hash(x) 를 -hash(-x) 로 정의합니다. 얻어진 해시가 -1 이면 -2 로 바꿉니다.
- The particular values sys.hash_info.inf and -sys.hash_info.inf are used as hash values for positive infinity or negative infinity (respectively).

- 복소수(*complex*) `z` 의 경우, `hash(z.real) + sys.hash_info.imag * hash(z.imag)` 를 계산하여 실수부와 허수부의 해시값을 결합하는데, `2**sys.hash_info.width` 의 모듈로로 환원해서 `range(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1))` 범위에 들어가도록 만듭니다. 다시 한번, 결과가 `-1` 이라면 `-2` 로 바꿉니다.

위의 규칙을 명확히 하기 위해, 여기에 유리수, *float*, *complex* 의 해시를 계산하는, 내장 해시와 동등한, 파이썬 코드를 예시합니다:

```
import sys, math

def hash_fraction(m, n):
    """Compute the hash of a rational number m / n.

    Assumes m and n are integers, with n positive.
    Equivalent to hash(fractions.Fraction(m, n)).

    """
    P = sys.hash_info.modulus
    # Remove common factors of P.  (Unnecessary if m and n already coprime.)
    while m % P == n % P == 0:
        m, n = m // P, n // P

    if n % P == 0:
        hash_value = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: pow(n, P-1, P) is 1, so
        # pow(n, P-2, P) gives the inverse of n modulo P.
        hash_value = (abs(m) % P) * pow(n, P - 2, P) % P
    if m < 0:
        hash_value = -hash_value
    if hash_value == -1:
        hash_value = -2
    return hash_value

def hash_float(x):
    """Compute the hash of a float x."""

    if math.isnan(x):
        return object.__hash__(x)
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""

    hash_value = hash_float(z.real) + sys.hash_info.imag * hash_float(z.imag)
    # do a signed reduction modulo 2**sys.hash_info.width
    M = 2**(sys.hash_info.width - 1)
    hash_value = (hash_value & (M - 1)) - (hash_value & M)
    if hash_value == -1:
        hash_value = -2
    return hash_value
```

4.5 Boolean Type - bool

Booleans represent truth values. The *bool* type has exactly two constant instances: `True` and `False`.

The built-in function *bool()* converts any value to a boolean, if the value can be interpreted as a truth value (see section 논리값 검사 above).

For logical operations, use the *boolean operators* `and`, `or` and `not`. When applying the bitwise operators `&`, `|`, `^` to two booleans, they return a bool equivalent to the logical operations "and", "or", "xor". However, the logical operators `and`, `or` and `!=` should be preferred over `&`, `|` and `^`.

버전 3.12부터 폐지됨: The use of the bitwise inversion operator `~` is deprecated and will raise an error in Python 3.16.

bool is a subclass of *int* (see 숫자 형 — *int, float, complex*). In many numeric contexts, `False` and `True` behave like the integers 0 and 1, respectively. However, relying on this is discouraged; explicitly convert using *int()* instead.

4.6 이터레이터 형

파이썬은 컨테이너에 대한 이터레이션 개념을 지원합니다. 이것은 두 개의 메서드를 사용해서 구현됩니다; 이것들은 사용자 정의 클래스가 이터레이션을 지원할 수 있도록 하는 데 사용됩니다. 아래에서 더 자세히 설명할 시퀀스는 항상 이터레이션 메서드를 지원합니다.

One method needs to be defined for container objects to provide *iterable* support:

`container.__iter__()`

Return an *iterator* object. The object is required to support the iterator protocol described below. If a container supports different types of iteration, additional methods can be provided to specifically request iterators for those iteration types. (An example of an object supporting multiple forms of iteration would be a tree structure which supports both breadth-first and depth-first traversal.) This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

이터레이터 객체 자체는 다음과 같은 두 가지 메서드를 지원해야 하는데, 둘이 함께 이터레이터 프로토콜 (*iterator protocol*) 를 이룹니다.:

`iterator.__iter__()`

Return the *iterator* object itself. This is required to allow both containers and iterators to be used with the `for` and `in` statements. This method corresponds to the `tp_iter` slot of the type structure for Python objects in the Python/C API.

`iterator.__next__()`

Return the next item from the *iterator*. If there are no further items, raise the *StopIteration* exception. This method corresponds to the `tp_iternext` slot of the type structure for Python objects in the Python/C API.

파이썬은 일반적이거나 특정한 시퀀스 형, 딕셔너리, 기타 더 특화된 형태에 대한 이터레이션을 지원하기 위해 여러 이터레이터 객체를 정의합니다. 이터레이터 프로토콜의 구현을 넘어서 개별적인 형이 중요하지는 않습니다.

일단 이터레이터의 *__next__()* 메서드가 *StopIteration* 를 일으키면, 그 이후의 호출에 대해서도 같이 동작해야 합니다. 이 속성을 따르지 않는 구현은 망가진 것으로 간주합니다.

4.6.1 제너레이터 형

Python's *generator*s provide a convenient way to implement the iterator protocol. If a container object's `__iter__()` method is implemented as a generator, it will automatically return an iterator object (technically, a generator object) supplying the *__iter__()* and `__next__()` methods. More information about generators can be found in the documentation for the yield expression.

4.7 시퀀스 형 — list, tuple, range

세 가지 기본 시퀀스 형이 있습니다: 리스트, 튜플, 범위 객체. 바이너리 데이터 와 텍스트 문자열 의 처리를 위해 추가된 시퀀스 형들은 별도의 섹션에서 설명합니다.

4.7.1 공통 시퀀스 연산

다음 표의 연산들은 대부분의 가변과 불변 시퀀스에서 지원됩니다. 사용자 정의 시퀀스에서 이 연산들을 올바르게 구현하기 쉽게 하려고 `collections.abc.Sequence` ABC가 제공됩니다.

이 표는 우선순위에 따라 오름차순으로 시퀀스 연산들을 나열합니다. 표에서, *s* 와 *t* 는 같은 형의 시퀀스고, *n*, *i*, *j*, *k* 는 정수이고, *x* 는 *s* 가 요구하는 형과 값 제한을 만족하는 임의의 객체입니다.

`in` 과 `not in` 연산은 비교 연산과 우선순위가 같습니다. `+` (이어 붙이기)와 `*` (반복) 연산은 대응하는 숫자 연산과 같은 우선순위를 갖습니다.[3]

연산	결과	노트
`x in s`	*s* 의 항목 중 하나가 *x* 와 같으면 `True`, 그렇지 않으면 `False`	(1)
`x not in s`	*s* 의 항목 중 하나가 *x* 와 같으면 `False`, 그렇지 않으면 `True`	(1)
`s + t`	*s* 와 *t* 의 이어 붙이기	(6)(7)
`s * n` 또는 `n * s`	*s* 를 그 자신에 *n* 번 더하는 것과 같습니다	(2)(7)
`s[i]`	*s* 의 *i* 번째 항목, 0에서 시작합니다	(3)(8)
`s[i:j]`	*s* 의 *i* 에서 *j* 까지의 슬라이스	(3)(4)
`s[i:j:k]`	*s* 의 *i* 에서 *j* 까지 스텝 *k* 의 슬라이스	(3)(5)
`len(s)`	*s* 의 길이	
`min(s)`	*s* 의 가장 작은 항목	
`max(s)`	*s* 의 가장 큰 항목	

같은 형의 시퀀스는 비교를 지원합니다. 특히, 튜플과 리스트는 대응하는 항목들을 사전적으로 비교합니다. 이것은 같다고 비교되기 위해서는, 모든 항목이 같다고 비교되고, 두 시퀀스의 형과 길이가 같아야 함을 의미합니다. (자세한 내용은 언어 레퍼런스의 comparisons를 참조하십시오.)

Forward and reversed iterators over mutable sequences access values using an index. That index will continue to march forward (or backward) even if the underlying sequence is mutated. The iterator terminates only when an `IndexError` or a `StopIteration` is encountered (or when the index drops below zero).

노트:

(1) `in` 과 `not in` 연산은 일반적으로 단순한 포함 검사를 위해서만 사용되지만, 몇몇 특수한 시퀀스 (`str`, `bytes`, `bytearray` 같은) 들은 서브 시퀀스 검사에 사용하기도 합니다:

```
>>> "gg" in "eggs"
True
```

(2) *n* 의 값이 `0` 보다 작으면 `0` 으로 처리됩니다 (*s* 와 같은 형의 빈 시퀀스가 됩니다). 시퀀스 *s* 의 항목들이 복사되지 않음에 주의해야 합니다; 그들은 여러 번 참조됩니다. 이것은 종종 새 파이썬 프로그래머들을 괴롭힙니다; 이 코드를 살펴보세요:

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

무슨 일이 일어났는가 하면, `[[]]` 는 빈 리스트를 포함하는 길이 1인 리스트인데, `[[]] * 3` 의 세 항목은 모두 같은 빈 리스트를 참조합니다. `lists` 의 어느 항목을 수정하더라도 이 하나의 리스트를 수정하게 됩니다. 서로 다른 리스트들을 포함하는 리스트는 이런 식으로 만들 수 있습니다:

[3] 파서가 피연산자 유형을 알 수 없으므로 그럴 수밖에 없습니다.

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

더 자세한 설명은 FAQ 항목 faq-multidimensional-list에서 얻을 수 있습니다.

(3) *i* 또는 *j* 가 음수인 경우, 인덱스는 시퀀스 *s* 의 끝에 상대적입니다: `len(s) + i` 이나 `len(s) + j` 로 치환됩니다. 하지만 `-0` 은 여전히 `0` 입니다.

(4) *i* 에서 *j* 까지의 *s* 의 슬라이스는 `i <= k < j` 를 만족하는 인덱스 *k* 의 항목들로 구성된 시퀀스로 정의됩니다. *i* 또는 *j* 가 `len(s)` 보다 크면 `len(s)` 을 사용합니다. *i* 가 생략되거나 `None` 이라면 `0` 을 사용합니다. *j* 가 생략되거나 `None` 이면 `len(s)` 을 사용합니다. *i* 가 *j* 보다 크거나 같으면 빈 슬라이스가 됩니다.

(5) 스텝 *k* 가 있는 *i* 에서 *j* 까지의 슬라이스는 `0 <= n < (j-i)/k` 를 만족하는 인덱스 `x = i + n*k` 의 항목들로 구성된 시퀀스로 정의됩니다. 다시 말하면, 인덱스는 `i`, `i+k`, `i+2*k`, `i+3*k` 등이며 *j* 에 도달할 때 멈춥니다 (하지만 절대 *j* 를 포함하지는 않습니다). *k* 가 양수면 *i* 와 *j* 는 더 큰 경우 `len(s)` 로 줄어듭니다. *k* 가 음수면, *i* 와 *j* 는 더 큰 경우 `len(s) - 1` 로 줄어듭니다. *i* 또는 *j* 가 생략되거나 `None` 이면, 그것들은 "끝" 값이 됩니다 (끝은 *k* 의 부호에 따라 달라집니다). *k* 는 0 일 수 없음에 주의하세요. *k* 가 `None` 이면 `1` 로 취급됩니다.

(6) 불변 시퀀스를 이어 붙이면 항상 새로운 객체가 생성됩니다. 이것은 반복적으로 이어붙이기를 해서 시퀀스를 만들 때 실행 시간이 시퀀스의 총 길이의 제곱에 비례한다는 뜻입니다. 선형 실행 시간 비용을 얻으려면 아래 대안 중 하나로 전환해야 합니다:

- *str* 객체를 이어붙이기를 한다면, 리스트를 만들고 마지막에 *str.join()* 을 사용하거나 *io.StringIO* 인스턴스에 쓰고 완료될 때 값을 꺼낼 수 있습니다
- *bytes* 객체를 연결하는 경우 비슷하게 *bytes.join()* 또는 *io.BytesIO* 를 사용하거나, *bytearray* 객체를 사용하여 제자리에서 이어붙이기를 할 수 있습니다. *bytearray* 객체는 가변이고 효율적인 과할당(overallocation) 메커니즘을 가지고 있습니다.
- *tuple* 객체를 이어붙이기를 한다면, 대신 *list*를 extend 하십시오.
- 다른 형의 경우 관련 클래스 문서를 조사하십시오.

(7) 일부 시퀀스 형 (예를 들어 *range*)은 특정 패턴을 따르는 항목 시퀀스 만 지원하기 때문에 시퀀스 이어붙이기나 반복을 지원하지 않습니다.

(8) An *IndexError* is raised if *i* is outside the sequence range.

Sequence Methods

Sequence types also support the following methods:

`sequence.`**`count`**`(`*value*, /`)`

Return the total number of occurrences of *value* in *sequence*.

`sequence.`**`index`**`(`*value*[, *start*[, *stop*]`)`

Return the index of the first occurrence of *value* in *sequence*.

Raises *ValueError* if *value* is not found in *sequence*.

The *start* or *stop* arguments allow for efficient searching of subsections of the sequence, beginning at *start* and ending at *stop*. This is roughly equivalent to `start + sequence[start:stop].index(value)`, only without copying any data.

> **조심**
>
> Not all sequence types support passing the *start* and *stop* arguments.

4.7.2 불변 시퀀스 형

불변 시퀀스 형이 일반적으로 구현하지만, 가변 시퀀스 형에서는 구현되지 않는 연산은 내장 `hash()` 에 대한 지원입니다.

이 지원은 `tuple` 인스턴스와 같은 불변 시퀀스를 `dict` 키로 사용하고 `set` 및 `frozenset` 인스턴스에 저장할 수 있도록 합니다.

해시 불가능 값을 포함하는 불변 시퀀스를 해시 하려고 하면 `TypeError` 를 일으킵니다.

4.7.3 가변 시퀀스 형

다음 표의 연산들은 가변 시퀀스 형에 정의되어 있습니다. 사용자 정의 시퀀스에서 이 연산들을 올바르게 구현하기 쉽게 하려고 `collections.abc.MutableSequence` ABC가 제공됩니다.

표에서 *s* 는 가변 시퀀스 형의 인스턴스이고, *t* 는 임의의 이터러블 객체이며, *x* 는 *s* 가 요구하는 형 및 값 제한을 충족시키는 임의의 객체입니다 (예를 들어, `bytearray` 는 값 제한 `0 <= x <= 255` 를 만족하는 정수만 받아들입니다.

연산	결과	노트
`s[i] = x`	*s* 의 항목 *i* 를 *x* 로 대체합니다	
`del s[i]`	removes item *i* of *s*	
`s[i:j] = t`	*i* 에서 *j* 까지의 *s* 슬라이스가 이터러블 *t* 의 내용으로 대체됩니다	
`del s[i:j]`	removes the elements of `s[i:j]` from the list (same as `s[i:j] = []`)	
`s[i:j:k] = t`	`s[i:j:k]` 의 항목들이 *t* 의 항목들로 대체됩니다	(1)
`del s[i:j:k]`	리스트에서 `s[i:j:k]` 의 항목들을 제거합니다	
`s += t`	*t* 의 내용으로 *s* 를 확장합니다 (대부분 `s[len(s):len(s)] = t` 와 같습니다)	
`s *= n`	내용이 *n* 번 반복되도록 *s* 를 갱신합니다	(2)

노트:

(1) If *k* is not equal to `1`, *t* must have the same length as the slice it is replacing.

(2) *n* 값은 정수이거나, `__index__()` 를 구현하는 객체입니다. *n* 이 0 이거나 음수면 시퀀스를 지웁니다. 시퀀스의 항목들은 복사되지 않습니다; 공통 시퀀스 연산에서 `s * n` 를 위해 설명한 것처럼 여러 번 참조됩니다.

Mutable Sequence Methods

Mutable sequence types also support the following methods:

`sequence.`**`append`**`(`*value*, /`)`

Append *value* to the end of the sequence This is equivalent to writing `seq[len(seq):len(seq)] = [value]`.

`sequence.`**`clear`**`()`

Added in version 3.3.

Remove all items from *sequence*. This is equivalent to writing `del sequence[:]`.

`sequence.`**`copy`**`()`

Added in version 3.3.

Create a shallow copy of *sequence*. This is equivalent to writing `sequence[:]`.

> **힌트**
>
> The `copy()` method is not part of the `MutableSequence ABC`, but most concrete mutable sequence types provide it.

sequence.**extend**(*iterable*, /)

Extend *sequence* with the contents of *iterable*. For the most part, this is the same as writing `seq[len(seq):len(seq)] = iterable`.

sequence.**insert**(*index*, *value*, /)

Insert *value* into *sequence* at the given *index*. This is equivalent to writing `sequence[index:index] = [value]`.

sequence.**pop**(*index=-1*, /)

Retrieve the item at *index* and also removes it from *sequence*. By default, the last item in *sequence* is removed and returned.

sequence.**remove**(*value*, /)

Remove the first item from *sequence* where `sequence[i] == value`.

Raises `ValueError` if *value* is not found in *sequence*.

sequence.**reverse**()

Reverse the items of *sequence* in place. This method maintains economy of space when reversing a large sequence. To remind users that it operates by side-effect, it returns `None`.

4.7.4 리스트

리스트는 가변 시퀀스로, 일반적으로 등질 항목들의 모음을 저장하는 데 사용됩니다 (정확한 유사도는 응용 프로그램마다 다를 수 있습니다).

class list(*iterable*=(), /)

리스트는 여러 가지 방법으로 만들 수 있습니다:

- 대괄호를 사용하여 빈 리스트를 표시하기: `[]`
- 대괄호를 사용하여 쉼표로 항목 구분하기: `[a]`, `[a, b, c]`
- 리스트 컴프리헨션 사용하기: `[x for x in iterable]`
- 형 생성자를 사용하기: `list()` 또는 `list(iterable)`

생성자는 항목들과 그 순서가 *iterable* 과 같은 리스트를 만듭니다. *iterable* 은 시퀀스, 이터레이션을 지원하는 컨테이너, 이터레이터 객체가 될 수 있습니다. *iterable* 이 이미 리스트라면, `iterable[:]` 과 비슷하게 복사본을 만들어서 반환합니다. 예를 들어, `list('abc')` 는 `['a', 'b', 'c']` 를 반환하고 `list( (1, 2, 3) )` 는 `[1, 2, 3]` 를 반환합니다. 인자가 주어지지 않으면, 생성자는 새로운 빈 리스트인 `[]` 을 만듭니다.

다른 많은 연산도 리스트를 만드는데, 내장 `sorted()` 도 그런 것 중 하나다.

리스트는 공통 과 가변 시퀀스 연산들을 모두 구현합니다. 또한, 리스트는 다음과 같은 추가 메서드를 제공합니다:

sort(*, *key=None*, *reverse=False*)

이 메서드는 항목 간의 `<` 비교만 사용하여 리스트를 제자리에서 정렬합니다. 예외는 억제되지 않습니다 - 비교 연산이 실패하면 전체 정렬 연산이 실패합니다 (리스트는 부분적으로 수정된 상태로 남아있게 됩니다).

`sort()` 는 키워드로만 전달할 수 있는 두 개의 인자를 받아들입니다 (키워드-전용 인자):

key 는 인자 하나를 받아들이는 함수를 지정하는데, 각 리스트 요소에서 비교 키를 추출하는 데 사용됩니다 (예를 들어, `key=str.lower`). 리스트의 각 항목에 해당하는 키는 한 번만 계산된 후 전체 정렬 프로세스에 사용됩니다. 기본값 `None` 은 리스트 항목들이 별도의 키값을 계산하지 않고 직접 정렬된다는 것을 의미합니다.

`functools.cmp_to_key()` 유틸리티는 2.x 스타일 *cmp* 함수를 *key* 함수로 변환하는 데 사용할 수 있습니다.

reverse 는 논리값입니다. `True` 로 설정되면, 각 비교가 역전된 것처럼 리스트 요소들이 정렬됩니다.

이 메서드는 큰 시퀀스를 정렬할 때 공간 절약을 위해 시퀀스를 제자리에서 수정합니다. 부작용으로 작동한다는 것을 사용자에게 상기시키기 위해 정렬된 시퀀스를 돌려주지 않습니다 (새 정렬 된 리스트 인스턴스를 명시적으로 요청하려면 `sorted()` 를 사용하십시오).

`sort()` 메서드는 안정적임이 보장됩니다. 정렬은 같다고 비교되는 요소들의 상대적 순서를 변경하지 않으면 안정적입니다 — 이는 여러 번 정렬하는 데 유용합니다 (예를 들어, 부서별로 정렬한 후에 급여 등급으로 정렬).

정렬 예제와 간단한 정렬 자습서는 sortinghowto를 참조하십시오.

리스트가 정렬되는 동안, 리스트를 변경하려고 할 때의, 또는 관찰하려고 할 때조차, 효과는 정의되지 않습니다. 파이썬의 C 구현은 그동안 리스트를 비어있는 것으로 보이게 하고, 정렬 중에 리스트가 변경되었음을 감지할 수 있다면 `ValueError` 를 일으킵니다.

4.7.5 튜플

튜플은 불변 시퀀스인데, 보통 이질적인 데이터의 모음을 저장하는 데 사용됩니다 (예를 들어, 내장 `enumerate()` 가 만드는 2-튜플). 튜플은 등질적인 데이터의 불변 시퀀스가 필요한 경우에도 사용됩니다 (예를 들어, `set` 이나 `dict` 인스턴스에 저장하고자 하는 경우).

class tuple(*iterable=()*, /)

튜플은 여러 가지 방법으로 만들 수 있습니다:

- 괄호를 사용하여 빈 튜플을 나타내기: `()`
- 단일 항목 튜플을 위해 끝에 쉼표를 붙이기: `a,` 또는 `(a,)`
- 항목을 쉼표로 구분하기: `a, b, c` 또는 `(a, b, c)`
- 내장 `tuple()` 사용하기: `tuple()` 또는 `tuple(iterable)`

생성자는 항목들과 그 순서가 *iterable* 과 같은 튜플을 만듭니다. *iterable* 은 시퀀스, 이터레이션을 지원하는 컨테이너, 이터레이터 객체가 될 수 있습니다. *iterable* 이 이미 튜플이라면 변경되지 않은 상태로 반환됩니다. 예를 들어 `tuple('abc')` 는 `('a', 'b', 'c')` 를 반환하고, `tuple( [1, 2, 3] )` 는 `(1, 2, 3)` 을 반환합니다. 인자가 주어지지 않으면, 생성자는 새로운 빈 튜플인 `()` 을 만듭니다.

튜플을 만드는 것은 실제로는 괄호가 아닌 쉼표임에 유의하십시오. 괄호는 빈 튜플의 경우를 제외하고는 선택적이거나 문법상의 모호함을 피하고자 필요합니다. 예를 들어, `f(a, b, c)` 는 3개의 인자를 가진 함수 호출이지만, `f((a, b, c))` 는 하나의 인자로 3-튜플을 갖는 함수 호출입니다.

튜플은 공통 시퀀스 연산을 모두 구현합니다.

이름에 의한 액세스가 인덱스에 의한 액세스보다 더 명확한 이질적 데이터 컬렉션의 경우, `collections.namedtuple()` 이 단순한 튜플 객체보다 더 적절한 선택일 수 있습니다.

4.7.6 범위

`range` 형은 숫자의 불변 시퀀스를 나타내며 `for` 루프에서 특정 횟수만큼 반복하는 데 흔히 사용됩니다.

class range(*stop*, /)

class range(*start*, *stop*, *step=1*, /)

The arguments to the range constructor must be integers (either built-in `int` or any object that implements the `__index__()` special method). If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. If *step* is zero, `ValueError` is raised.

양수 *step* 의 경우, 범위 `r` 의 내용은 식 `r[i] = start + step*i` 에 의해 결정됩니다. 이때 `i >= 0` 이고 `r[i] < stop` 입니다.

음수 *step* 의 경우, 범위의 내용은 여전히 식 `r[i] = start + step*i` 에 의해 결정되지만, 제약 조건은 `i >= 0` 과 `r[i] > stop` 이 됩니다.

`r[0]` 제약 조건을 만족시키지 않으면 범위 객체는 비게 됩니다. 범위는 음의 인덱스를 지원하지만, 이는 시퀀스의 끝에서부터 양의 인덱스만큼 떨어진 인덱스로 해석됩니다.

`sys.maxsize` 보다 큰 절댓값을 포함하는 범위는 허용되지만, (`len()` 과 같은) 일부 기능은 `OverflowError` 를 발생시킬 수 있습니다.

범위 예제:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

범위는 이어 붙이기와 반복을 제외한 공통 시퀀스 연산을 모두 구현합니다 (범위 객체는 엄격한 패턴을 따르는 시퀀스 만 나타낼 수 있는데 반복과 이어 붙이기는 보통 그 패턴을 위반한다는 사실에 기인합니다).

start

start 매개변수의 값 (또는 매개변수가 제공되지 않으면 `0`)

stop

stop 매개변수의 값

step

step 매개변수의 값 (또는 매개변수가 제공되지 않으면 `1`)

정규 `list` 나 `tuple` 에 비해 `range` 형의 장점은 `range` 객체는 표현하는 범위의 크기에 무관하게 항상 같은 (작은) 양의 메모리를 사용한다는 것입니다 (`start`, `stop`, `step` 값만을 저장하고, 필요에 따라 개별 항목과 하위 범위를 계산하기 때문입니다).

범위 객체는 `collections.abc.Sequence` ABC를 구현하고, 포함 검사, 요소 인덱스 검색, 슬라이싱, 음수 인덱스 지원과 같은 기능을 제공합니다 (시퀀스 형 — *list, tuple, range* 를 보세요):

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

`==` 나 `!=` 로 범위 객체가 같은지 검사하면 시퀀스처럼 비교합니다. 즉, 두 범위 객체가 같은 시퀀스의 값을 나타낼 때 같다고 취급됩니다. (같다고 비교되는 두 개의 범위 객체가 서로 다른 `start`, `stop`, `step` 어트리뷰트를 가질 수 있음에 주의하세요. 예를 들어, `range(0) == range(2, 1, 3)` 또는 `range(0, 3, 2) == range(0, 4, 2)`.)

버전 3.2에서 변경: 시퀀스 ABC를 구현합니다. `int` 객체의 포함 검사는 모든 항목을 이터레이트하는 대신 상수 시간으로 수행됩니다.

버전 3.3에서 변경: (객체 아이덴티티에 기반을 두는 대신) 범위 객체가 정의하는 값들의 시퀀스에 기반을 둔 비교를 위해 '==' 와 '!=' 를 정의합니다.

Added the `start`, `stop` and `step` attributes.

더 보기

- The linspace recipe shows how to implement a lazy version of range suitable for floating-point applications.

4.8 Text and Binary Sequence Type Methods Summary

The following table summarizes the text and binary sequence types methods by category.

Category		`str` methods	
Formatting		`str.format()`	
		`str.format_map()`	
		f-strings	
		printf 스타일 문자열 포매팅	
Searching and Replacing	`str.find()`	`str.rfind()`	
	`str.index()`	`str.rindex()`	
		`str.startswith()`	
		`str.endswith()`	
		`str.count()`	
		`str.replace()`	
Splitting and Joining	`str.split()`		`str.rsplit()`
		`str.splitlines()`	
		`str.partition()`	
		`str.rpartition()`	
		`str.join()`	
String Classification		`str.isalpha()`	
		`str.isdecimal()`	
		`str.isdigit()`	
		`str.isnumeric()`	
		`str.isalnum()`	
		`str.isidentifier()`	
		`str.islower()`	
		`str.isupper()`	
		`str.istitle()`	
		`str.isspace()`	
		`str.isprintable()`	
Case Manipulation		`str.lower()`	
		`str.upper()`	
		`str.casefold()`	
		`str.capitalize()`	
		`str.title()`	
		`str.swapcase()`	
Padding and Stripping	`str.ljust()`		`str.rjust()`
		`str.center()`	
		`str.expandtabs()`	
		`str.strip()`	
	`str.lstrip()`		
Translation and Encoding		`str.translate()`	
		`str.maketrans()`	
		`str.encode()`	

Category	str methods

4.9 텍스트 시퀀스 형 — str

파이썬의 텍스트 데이터는 *str*, 또는 문자열 (*strings*), 객체를 사용하여 처리됩니다. 문자열은 유니코드 코드 포인트의 불변 시퀀스 입니다. 문자열 리터럴은 다양한 방법으로 작성됩니다:

- 작은따옴표: `'"큰" 따옴표를 담을 수 있습니다'`
- Double quotes: `"allows embedded 'single' quotes"`
- 삼중 따옴표: `'''세 개의 작은따옴표''', """세 개의 큰따옴표"""`

삼중 따옴표로 묶인 문자열은 여러 줄에 걸쳐있을 수 있습니다 - 연관된 모든 공백이 문자열 리터럴에 포함됩니다.

단일 표현식의 일부이고 그 들 사이에 공백만 있는 문자열 리터럴은 묵시적으로 단일 문자열 리터럴로 변환됩니다. 즉, `("spam " "eggs") == "spam eggs"`.

See strings for more about the various forms of string literal, including supported escape sequences, and the `r` ("raw") prefix that disables most escape sequence processing.

문자열은 *str* 생성자를 사용하여 다른 객체로부터 만들어질 수도 있습니다.

별도의 "문자" 형이 없으므로 문자열을 인덱싱하면 길이가 1 인 문자열이 생성됩니다. 즉, 비어 있지 않은 문자열 *s* 의 경우, `s[0] == s[0:1]` 입니다.

또한, 가변 문자열형은 없지만, 여러 단편으로부터 문자열을 효율적으로 구성하는데 *str.join()* 또는 *io.StringIO* 를 사용할 수 있습니다.

버전 3.3 에서 변경: 파이썬 2 시리즈와의 하위 호환성을 위해서, `u` 접두어가 문자열 리터럴에 다시 한번 허용됩니다. 문자열 리터럴의 의미에 영향을 미치지 않으며 `r` 접두사와 결합 될 수 없습니다.

class str(**, encoding='utf-8', errors='strict'*)

class str(*object*)

class str(*object, encoding, errors='strict'*)

class str(*object, *, errors*)

object 의 문자열 버전을 돌려줍니다. *object* 가 제공되지 않으면, 빈 문자열을 돌려줍니다. 그렇지 않으면, `str()` 의 동작은 *encoding* 또는 *errors* 가 주어졌는지에 따라 달라지는데, 다음과 같습니다.

If neither *encoding* nor *errors* is given, `str(object)` returns `type(object).__str__(object)`, which is the "informal" or nicely printable string representation of *object*. For string objects, this is the string itself. If *object* does not have a `__str__()` method, then *str()* falls back to returning *repr(object)*.

encoding 또는 *errors* 중 적어도 하나가 주어지면, *object* 는 *bytes-like object* (예, *bytes* 또는 *bytearray*) 이어야 합니다. 이 경우, *object* 가 *bytes* (또는 *bytearray*) 객체이면, `str(bytes, encoding, errors)` 는 *bytes.decode(encoding, errors)* 와 동등합니다. 그 이외의 경우, *bytes.decode()* 호출 전에 버퍼 객체의 하부 바이트열 객체를 얻습니다. 버퍼 객체에 대한 정보는 바이너리 시퀀스 형 — *bytes, bytearray, memoryview* 와 bufferobjects 를 보십시오.

encoding 또는 *errors* 인자 없이 *bytes* 객체를 *str()* 에 전달하는 것은 비형식적 문자열 표현을 반환하는 첫 번째 상황에 해당합니다 (파이썬 명령행 옵션 `-b` 도 보십시오). 예를 들면:

```
>>> str(b'Zoot!')
"b'Zoot!'"
```

`str` 클래스와 그 메서드에 대한 더 자세한 정보는 텍스트 시퀀스 형 — *str* 와 아래의 문자열 메서드 섹션을 보십시오. 포맷된 문자열을 출력하려면 f-strings 및 포맷 문자열 문법 섹션을 참조하십시오. 또한, 텍스트 처리 서비스 섹션을 보십시오.

4.9.1 문자열 메서드

문자열은 공통 시퀀스 연산들을 모두 구현하고, 아래에 기술된 추가적인 메서드도 구현합니다.

문자열은 또한 두 가지 스타일의 문자열 포매팅을 지원합니다. 하나는 큰 폭의 유연성과 사용자 지정을 제공하고 (참조 `str.format()`, 포맷 문자열 문법, 사용자 지정 문자열 포매팅을 참조하세요) 다른 하나는 C `printf` 스타일에 기반을 두는데, 더 좁은 범위의 형을 처리하고 올바르게 사용하기는 다소 어렵지만, 처리할 수 있는 경우에는 종종 더 빠릅니다 (*printf* 스타일 문자열 포매팅).

표준 라이브러리의 텍스트 처리 서비스 섹션은 다양한 텍스트 관련 유틸리티를 (`re` 모듈의 정규식 지원을 포함합니다) 제공하는 많은 다른 모듈들을 다룹니다.

`str.`**`capitalize`**`()`

첫 문자가 대문자이고 나머지가 소문자인 문자열의 복사본을 돌려줍니다.

버전 3.8에서 변경: 이제 첫 번째 문자는 대문자가 아닌 제목 케이스로 바뀝니다. 이는 이중 문자 (digraph)와 같은 문자는 전체 문자 대신 첫 문자만 대문자로 표시된다는 뜻입니다.

`str.`**`casefold`**`()`

케이스 폴딩 된 문자열을 반환합니다. 케이스 폴딩 된 문자열은 대소문자를 무시한 매칭에 사용될 수 있습니다.

케이스 폴딩은 소문자로 변환하는 것과 비슷하지만 문자열의 모든 케이스 구분을 제거하기 때문에 보다 공격적입니다. 예를 들어, 독일어 소문자 `'ß'` 는 `"ss"` 와 동등합니다. 이미 소문자이므로 `lower()` 는 `'ß'` 에 아무런 영향을 미치지 않습니다; `casefold()` 는 `"ss"` 로 변환합니다.

The casefolding algorithm is described in section 3.13 'Default Case Folding' of the Unicode Standard.

Added in version 3.3.

`str.`**`center`**`(`*width*, *fillchar*=' ', /`)`

Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is an ASCII space). The original string is returned if *width* is less than or equal to `len(s)`. For example:

```
>>> 'Python'.center(10)
'  Python  '
>>> 'Python'.center(10, '-')
'--Python--'
>>> 'Python'.center(4)
'Python'
```

`str.`**`count`**`(`*sub*[, *start*[, *end*]]`)`

범위 [*start*, *end*] 에서 부분 문자열 *sub* 가 중첩되지 않고 등장하는 횟수를 돌려줍니다. 선택적 인자 *start* 와 *end* 는 슬라이스 표기법으로 해석됩니다.

If *sub* is empty, returns the number of empty strings between characters which is the length of the string plus one. For example:

```
>>> 'spam, spam, spam'.count('spam')
3
>>> 'spam, spam, spam'.count('spam', 5)
2
>>> 'spam, spam, spam'.count('spam', 5, 10)
1
>>> 'spam, spam, spam'.count('eggs')
0
>>> 'spam, spam, spam'.count('')
17
```

`str.`**`encode`**`(`*encoding*='*utf-8*', *errors*='*strict*'`)`

Return the string encoded to `bytes`.

encoding defaults to `'utf-8'`; see 표준 인코딩 for possible values.

errors controls how encoding errors are handled. If `'strict'` (the default), a `UnicodeError` exception is raised. Other possible values are `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` and any other name registered via `codecs.register_error()`. See 에러 처리기 for details.

For performance reasons, the value of *errors* is not checked for validity unless an encoding error actually occurs, 파이썬 개발 모드 is enabled or a debug build is used. For example:

```
>>> encoded_str_to_bytes = 'Python'.encode()
>>> type(encoded_str_to_bytes)
<class 'bytes'>
>>> encoded_str_to_bytes
b'Python'
```

버전 3.1에서 변경: 키워드 인자 지원이 추가되었습니다.

버전 3.9에서 변경: The value of the *errors* argument is now checked in 파이썬 개발 모드 and in debug mode.

`str.`**`endswith`**(*suffix*[, *start*[, *end*]])

Return `True` if the string ends with the specified *suffix*, otherwise return `False`. *suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position. Using *start* and *end* is equivalent to `str[start:end].endswith(suffix)`. For example:

```
>>> 'Python'.endswith('on')
True
>>> 'a tuple of suffixes'.endswith(('at', 'in'))
False
>>> 'a tuple of suffixes'.endswith(('at', 'es'))
True
>>> 'Python is amazing'.endswith('is', 0, 9)
True
```

See also `startswith()` and `removesuffix()`.

`str.`**`expandtabs`**(*tabsize=8*)

Return a copy of the string where all tab characters are replaced by one or more spaces, depending on the current column and the given tab size. Tab positions occur every *tabsize* characters (default is 8, giving tab positions at columns 0, 8, 16 and so on). To expand the string, the current column is set to zero and the string is examined character by character. If the character is a tab (`\t`), one or more space characters are inserted in the result until the current column is equal to the next tab position. (The tab character itself is not copied.) If the character is a newline (`\n`) or return (`\r`), it is copied and the current column is reset to zero. Any other character is copied unchanged and the current column is incremented by one regardless of how the character is represented when printed. For example:

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
>>> print('01\t012\n0123\t01234'.expandtabs(4))
01  012
0123    01234
```

`str.`**`find`**(*sub*[, *start*[, *end*]])

Return the lowest index in the string where substring *sub* is found within the slice `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found. For example:

```
>>> 'spam, spam, spam'.find('sp')
0
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 'spam, spam, spam'.find('sp', 5)
6
```

See also *rfind()* and *index()*.

> **참고**
>
> *find()* 메서드는 *sub* 의 위치를 알아야 할 경우에만 사용해야 합니다. *sub* 가 부분 문자열인지 확인하려면 `in` 연산자를 사용하십시오:
>
> ```
> >>> 'Py' in 'Python'
> True
> ```

`str.`**`format`**(**args, **kwargs*)

문자열 포맷 연산을 수행합니다. 이 메서드가 호출되는 문자열은 리터럴 텍스트나 중괄호 `{}` 로 구분된 치환 필드를 포함할 수 있습니다. 각 치환 필드는 위치 인자의 숫자 인덱스나 키워드 인자의 이름을 가질 수 있습니다. 각 치환 필드를 해당 인자의 문자열 값으로 치환한 문자열의 사본을 돌려줍니다.

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

포맷 문자열에 지정할 수 있는 다양한 포맷 옵션에 대한 설명은 포맷 문자열 문법 을 참조하십시오.

> **참고**
>
> 숫자(*int*, *float*, *complex*, *decimal.Decimal*와 서브 클래스)를 `n` 형식으로 포매팅할 때 (예: `'{:n}'.format(1234)`), 이 함수는 일시적으로 `LC_CTYPE` 로케일을 `LC_NUMERIC` 로케일로 설정하여 `localeconv()` 의 `decimal_point` 와 `thousands_sep` 필드를 디코드하는데, 이 필드들이 ASCII가 아니거나 1바이트보다 길고, `LC_NUMERIC` 로케일이 `LC_CTYPE` 로케일과 다를 때만 그렇게 합니다. 이 임시 변경은 다른 스레드에 영향을 줍니다.

버전 3.7에서 변경: 숫자를 `n` 형식으로 포매팅할 때, 이 함수는 어떤 경우에 일시적으로 `LC_CTYPE` 로케일을 `LC_NUMERIC` 로케일로 설정합니다.

`str.`**`format_map`**(*mapping, /*)

`str.format(**mapping)` 과 비슷하지만, *dict*로 복사되지 않고 `mapping` 을 직접 사용합니다. 예를 들어 `mapping` 이 dict 서브 클래스면 유용합니다:

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

Added in version 3.2.

`str.`**`index`**(*sub*[, *start*[, *end*]])

find() 과 비슷하지만, 부분 문자열을 찾을 수 없는 경우 *ValueError* 를 일으킵니다.

`str.`**`isalnum`**()

문자열 내의 모든 문자가 알파벳과 숫자이고, 적어도 하나의 문자가 존재하는 경우 `True`를 돌려주고, 그렇지 않으면 `False`를 돌려줍니다. 문자 `c` 는 다음 중 하나가 `True` 를 반환하면 알파벳이거나 숫자입니다: `c.isalpha()`, `c.isdecimal()`, `c.isdigit()`, `c.isnumeric()`.

str.**isalpha**()

Return True if all characters in the string are alphabetic and there is at least one character, False otherwise. Alphabetic characters are those characters defined in the Unicode character database as "Letter", i.e., those with general category property being one of "Lm", "Lt", "Lu", "Ll", or "Lo". Note that this is different from the Alphabetic property defined in the section 4.10 'Letters, Alphabetic, and Ideographic' of the Unicode Standard.

str.**isascii**()

문자열이 비어 있거나 문자열의 모든 문자가 ASCII이면 True를 돌려주고, 그렇지 않으면 False를 돌려줍니다. ASCII 문자는 U+0000-U+007F 범위의 코드 포인트를 가집니다.

Added in version 3.7.

str.**isdecimal**()

문자열 내의 모든 문자가 십진수 문자이고, 적어도 하나의 문자가 존재하는 경우 True를 돌려주고, 그렇지 않으면 False를 돌려줍니다. 십진수 문자는 십진법으로 숫자를 구성할 때 사용될 수 있는 문자들입니다. 예를 들어, U+0660, ARABIC-INDIC DIGIT ZERO. 형식적으로 십진수 문자는 유니코드 일반 범주 "Nd" 에 속하는 문자입니다.

str.**isdigit**()

문자열 내의 모든 문자가 디짓이고, 적어도 하나의 문자가 존재하는 경우 True를 돌려주고, 그렇지 않으면 False를 돌려줍니다. 디짓에는 십진수 문자와 호환성 위 첨자 숫자와 같은 특수 처리가 필요한 숫자가 포함됩니다. 여기에는 카로슈티 숫자처럼 십진법으로 숫자를 구성할 때 사용될 수 없는 것들이 포함됩니다. 형식적으로, 디짓은 속성값이 Numeric_Type=Digit 또는 Numeric_Type=Decimal 인 문자입니다.

str.**isidentifier**()

문자열이 섹션 section identifiers 의 언어 정의에 따른 유효한 식별자면 True를 돌려줍니다.

keyword.iskeyword() can be used to test whether string s is a reserved identifier, such as def and class.

예제:

```
>>> from keyword import iskeyword

>>> 'hello'.isidentifier(), iskeyword('hello')
(True, False)
>>> 'def'.isidentifier(), iskeyword('def')
(True, True)
```

str.**islower**()

문자열 내의 모든 케이스 문자가[4] 소문자이고, 적어도 하나의 케이스 문자가 존재하는 경우 True를 돌려주고, 그렇지 않으면 False를 돌려줍니다.

str.**isnumeric**()

문자열 내의 모든 문자가 숫자이고, 적어도 하나의 문자가 존재하는 경우 True를 돌려주고, 그렇지 않으면 False를 돌려줍니다. 숫자는 디짓과 유니코드 숫자 값 속성을 갖는 모든 문자를 포함합니다. 예를 들어, U+2155, VULGAR FRACTION ONE FIFTH. 형식적으로, 숫자는 속성 값이 Numeric_Type=Digit, Numeric_Type=Decimal, Numeric_Type=Numeric 인 문자입니다.

str.**isprintable**()

Return True if all characters in the string are printable, False if it contains at least one non-printable character.

Here "printable" means the character is suitable for *repr()* to use in its output; "non-printable" means that *repr()* on built-in types will hex-escape the character. It has no bearing on the handling of strings written to *sys.stdout* or *sys.stderr*.

The printable characters are those which in the Unicode character database (see *unicodedata*) have a general category in group Letter, Mark, Number, Punctuation, or Symbol (L, M, N, P, or S); plus the ASCII space 0x20. Nonprintable characters are those in group Separator or Other (Z or C), except the ASCII space.

[4] 케이스 문자는 일반 범주 속성이 "Lu" (Letter, 대문자), "Ll" (Letter, 소문자), "Lt" (Letter, 제목 문자) 중 한 가지인 경우입니다.

str.**isspace**()

문자열 내에 공백 문자만 있고, 적어도 하나의 문자가 존재하는 경우 True를 돌려주고, 그렇지 않으면 False를 돌려줍니다.

유니코드 문자 데이터베이스(*unicodedata*를 참조하십시오)에서, 일반 범주(general category)가 Zs("Separator, space")이거나 양방향 클래스(bidirectional class)가 WS, B 또는 S 중 하나이면 문자는 공백(*whitespace*)입니다.

str.**istitle**()

문자열이 제목 케이스 문자열이고 하나 이상의 문자가 있는 경우 True를 돌려줍니다. 예를 들어 대문자 앞에는 케이스 없는 문자만 올 수 있고 소문자는 케이스 문자 뒤에만 올 수 있습니다. 그렇지 않은 경우는 False를 돌려줍니다.

str.**isupper**()

문자열 내의 모든 케이스 문자가[Page 58, 4] 대문자이고, 적어도 하나의 케이스 문자가 존재하는 경우 True를 돌려주고, 그렇지 않으면 False를 돌려줍니다.

```
>>> 'BANANA'.isupper()
True
>>> 'banana'.isupper()
False
>>> 'baNana'.isupper()
False
>>> ' '.isupper()
False
```

str.**join**(*iterable*, /)

iterable 의 문자열들을 이어 붙인 문자열을 돌려줍니다. *iterable* 에 *bytes* 객체나 기타 문자열이 아닌 값이 있으면 *TypeError* 를 일으킵니다. 요소들 사이의 구분자는 이 메서드를 제공하는 문자열입니다.

str.**ljust**(*width*, *fillchar*=' ', /)

왼쪽으로 정렬된 문자열을 길이 *width* 인 문자열로 돌려줍니다. 지정된 *fillchar* (기본값은 ASCII 스페이스)을 사용하여 채웁니다. *width* 가 len(s) 보다 작거나 같은 경우 원래 문자열이 반환됩니다.

str.**lower**()

모든 케이스 문자[Page 58, 4] 가 소문자로 변환된 문자열의 복사본을 돌려줍니다.

The lowercasing algorithm used is described in section 3.13 'Default Case Folding' of the Unicode Standard.

str.**lstrip**(*chars*=*None*, /)

선행 문자가 제거된 문자열의 복사본을 돌려줍니다. *chars* 인자는 제거할 문자 집합을 지정하는 문자열입니다. 생략되거나 None 이라면, *chars* 인자의 기본값은 공백을 제거하도록 합니다. *chars* 인자는 접두사가 아닙니다; 모든 값 조합이 제거됩니다:

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

문자 집합의 모든 것이 아닌 단일 접두사 문자열을 제거하는 메서드는 *str.removeprefix()*를 참조하십시오. 예를 들면:

```
>>> 'Arthur: three!'.lstrip('Arthur: ')
'ee!'
>>> 'Arthur: three!'.removeprefix('Arthur: ')
'three!'
```

static str.**maketrans**(*dict*, /)

static str.**maketrans**(*from*, *to*, *remove*='', /)

이 정적 메서드는 *str.translate()* 에 사용할 수 있는 변환표를 돌려줍니다.

인자가 하나만 있으면 유니코드 포인트 (정수) 또는 문자 (길이가 1 인 문자열)를 유니코드 포인트, 문자열 (임의 길이) 또는 None 으로 매핑하는 딕셔너리여야 합니다. 문자 키는 유니코드 포인트로 변환됩니다.

If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in *from* will be mapped to the character at the same position in *to*. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

str.**partition**(*sep*, /)

sep 가 처음 나타나는 위치에서 문자열을 나누고, 구분자 앞에 있는 부분, 구분자 자체, 구분자 뒤에 오는 부분으로 구성된 3-튜플을 돌려줍니다. 구분자가 발견되지 않으면, 문자열 자신과 그 뒤를 따르는 두 개의 빈 문자열로 구성된 3-튜플을 돌려줍니다.

str.**removeprefix**(*prefix*, /)

문자열이 *prefix* 문자열로 시작하면, string[len(prefix):] 를 반환합니다. 그렇지 않으면, 원래 문자열의 사본을 반환합니다:

```
>>> 'TestHook'.removeprefix('Test')
'Hook'
>>> 'BaseTestCase'.removeprefix('Test')
'BaseTestCase'
```

Added in version 3.9.

str.**removesuffix**(*suffix*, /)

문자열이 *suffix* 문자열로 끝나고 해당 *suffix* 가 비어 있지 않으면, string[:-len(suffix)] 를 반환합니다. 그렇지 않으면, 원래 문자열의 사본을 반환합니다:

```
>>> 'MiscTests'.removesuffix('Tests')
'Misc'
>>> 'TmpDirMixin'.removesuffix('Tests')
'TmpDirMixin'
```

Added in version 3.9.

str.**replace**(*old*, *new*, /, *count=-1*)

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If *count* is given, only the first *count* occurrences are replaced. If *count* is not specified or -1, then all occurrences are replaced.

버전 3.13 에서 변경: *count* is now supported as a keyword argument.

str.**rfind**(*sub*[, *start*[, *end*]])

부분 문자열 *sub* 가 s[start:end] 내에 등장하는 가장 큰 문자열의 인덱스를 돌려줍니다. 선택적 인자 *start* 와 *end* 는 슬라이스 표기법으로 해석됩니다. 실패하면 -1 을 돌려줍니다.

str.**rindex**(*sub*[, *start*[, *end*]])

rfind() 와 비슷하지만, 부분 문자열 *sub* 를 찾을 수 없는 경우 *ValueError* 를 일으킵니다.

str.**rjust**(*width*, *fillchar*=' ', /)

오른쪽으로 정렬된 문자열을 길이 *width* 인 문자열로 돌려줍니다. 지정된 *fillchar* (기본값은 ASCII 스페이스)을 사용하여 채웁니다. *width* 가 len(s) 보다 작거나 같은 경우 원래 문자열이 반환됩니다.

str.**rpartition**(*sep*, /)

sep 가 마지막으로 나타나는 위치에서 문자열을 나누고, 구분자 앞에 있는 부분, 구분자 자체, 구분자 뒤에 오는 부분으로 구성된 3-튜플을 돌려줍니다. 구분자가 발견되지 않으면, 두 개의 빈 문자열과 그 뒤를 따르는 문자열 자신으로 구성된 3-튜플을 돌려줍니다.

str.**rsplit**(*sep=None, maxsplit=-1*)

sep 를 구분자 문자열로 사용하여 문자열에 있는 단어들의 리스트를 돌려줍니다. *maxsplit* 이 주어지면 가장 오른쪽에서 최대 *maxsplit* 번의 분할이 수행됩니다. *sep* 이 지정되지 않거나 `None` 이면, 구분자로 모든 공백 문자가 사용됩니다. 오른쪽에서 분리하는 것을 제외하면, `rsplit()` 는 아래에서 자세히 설명될 `split()` 처럼 동작합니다.

str.**rstrip**(*chars=None, /*)

후행 문자가 제거된 문자열의 복사본을 돌려줍니다. *chars* 인자는 제거할 문자 집합을 지정하는 문자열입니다. 생략되거나 `None` 이라면, *chars* 인자의 기본값은 공백을 제거하도록 합니다. *chars* 인자는 접미사가 아닙니다; 모든 값 조합이 제거됩니다:

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

문자 집합의 모든 것이 아닌 단일 접미사 문자열을 제거하는 메서드는 `str.removesuffix()` 를 참조하십시오. 예를 들면:

```
>>> 'Monty Python'.rstrip(' Python')
'M'
>>> 'Monty Python'.removesuffix(' Python')
'Monty'
```

str.**split**(*sep=None, maxsplit=-1*)

sep 를 구분자 문자열로 사용하여 문자열에 있는 단어들의 리스트를 돌려줍니다. *maxsplit* 이 주어지면 최대 *maxsplit* 번의 분할이 수행됩니다 (따라서, 리스트는 최대 `maxsplit+1` 개의 요소를 가지게 됩니다). *maxsplit* 이 지정되지 않았거나 `-1` 이라면 분할 수에 제한이 없습니다 (가능한 모든 분할이 만들어집니다).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters as a single delimiter (to split with multiple delimiters, use `re.split()`). Splitting an empty string with a specified separator returns `['']`.

예를 들면:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
>>> '1<>2<>3<4'.split('<>')
['1', '2', '3<4']
```

sep 이 지정되지 않거나 `None` 이면, 다른 분할 알고리즘이 적용됩니다: 연속된 공백 문자는 단일한 구분자로 간주하고, 문자열이 선행이나 후행 공백을 포함해도 결과는 시작과 끝에 빈 문자열을 포함하지 않습니다. 결과적으로, 빈 문자열이나 공백만으로 구성된 문자열을 `None` 구분자로 나누면 `[]` 를 돌려줍니다.

예를 들면:

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

If *sep* is not specified or is `None` and *maxsplit* is `0`, only leading runs of consecutive whitespace are considered.

예를 들면:

```
>>> "".split(None, 0)
[]
>>> "    ".split(None, 0)
[]
>>> "    foo    ".split(maxsplit=0)
['foo    ']
```

`str.`**`splitlines`**(*keepends=False*)

줄 경계에서 나눈 문자열의 줄 리스트를 돌려줍니다. *keepends* 가 참으로 주어지지 않는 한 결과 리스트에 줄 바꿈은 포함되지 않습니다.

이 메서드는 다음 줄 경계에서 나눕니다. 특히, 경계는 유니버설 줄 넘김 을 포함합니다.

표현	설명
`\n`	줄 넘김
`\r`	캐리지 리턴
`\r\n`	캐리지 리턴 + 줄 넘김
`\v` 또는 `\x0b`	수직 탭
`\f` 또는 `\x0c`	폼 피드
`\x1c`	파일 구분자
`\x1d`	그룹 구분자
`\x1e`	레코드 구분자
`\x85`	다음 줄 (C1 제어 코드)
`\u2028`	줄 구분자
`\u2029`	문단 구분자

버전 3.2에서 변경: `\v` 와 `\f` 를 줄 경계 목록에 추가했습니다.

예를 들면:

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

구분자 문자열 *sep* 이 주어졌을 때 *split()* 와 달리, 이 메서드는 빈 문자열에 대해서 빈 리스트를 돌려주고, 마지막 줄 바꿈은 새 줄을 만들지 않습니다:

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

비교해 보면, `split('\n')` 는 이렇게 됩니다:

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

`str.`**`startswith`**(*prefix*[, *start*[, *end*]])

문자열이 지정된 *prefix* 로 시작하면 `True` 를 돌려주고, 그렇지 않으면 `False` 를 돌려줍니다. *prefix* 는 찾고자 하는 접두사들의 튜플이 될 수도 있습니다. 선택적 *start* 가 제공되면 그 위치에서 검사를 시작합니다. 선택적 *end* 를 사용하면 해당 위치에서 비교를 중단합니다.

str.**strip**(*chars=None, /*)

선행과 후행 문자가 제거된 문자열의 복사본을 돌려줍니다. *chars* 인자는 제거할 문자 집합을 지정하는 문자열입니다. 생략되거나 None 이라면, *chars* 인자의 기본값은 공백을 제거하도록 합니다. *chars* 인자는 접두사나 접미사가 아닙니다; 모든 값 조합이 제거됩니다:

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

가장 바깥쪽의 선행 또는 후행 *chars* 인자 값들이 문자열에서 제거됩니다. 문자는 *chars* 에 있는 문자 집합에 포함되지 않은 문자에 도달할 때까지 맨 앞에서 제거됩니다. 끝에서도 유사한 동작이 수행됩니다. 예를 들면:

```
>>> comment_string = '#....... Section 3.2.1 Issue #32 .......'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

str.**swapcase**()

대문자를 소문자로, 그 반대로 마찬가지로 변환 한 문자열의 복사본을 돌려줍니다. s.swapcase().swapcase() == s 가 반드시 성립하지 않음에 주의하십시오.

str.**title**()

단어가 대문자로 시작하고 나머지 문자는 소문자가 되도록 문자열의 제목 케이스 버전을 돌려줍니다.

예를 들면:

```
>>> 'Hello world'.title()
'Hello World'
```

이 알고리즘은 단어를 글자들의 연속으로 보는 간단한 언어 독립적 정의를 사용합니다. 이 정의는 여러 상황에서 작동하지만, 축약과 소유의 아포스트로피가 단어 경계를 형성한다는 것을 의미하고, 이는 원하는 결과가 아닐 수도 있습니다:

```
>>> "they're bill's friends from the UK".title()
"They're Bill'S Friends From The Uk"
```

The *string.capwords()* function does not have this problem, as it splits words on spaces only.

Alternatively, a workaround for apostrophes can be constructed using regular expressions:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0).capitalize(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

str.**translate**(*table, /*)

Return a copy of the string in which each character has been mapped through the given translation table. The table must be an object that implements indexing via __getitem__(), typically a *mapping* or *sequence*. When indexed by a Unicode ordinal (an integer), the table object can do any of the following: return a Unicode ordinal or a string, to map the character to one or more other characters; return None, to delete the character from the return string; or raise a *LookupError* exception, to map the character to itself.

str.maketrans() 를 사용하여 다른 형식의 문자 대 문자 매핑으로 부터 변환 맵을 만들 수 있습니다.

커스텀 문자 매핑에 대한 보다 유연한 접근법은 *codecs* 모듈을 참고하십시오.

`str.upper()`

모든 케이스 문자[Page 58, 4] 가 대문자로 변환된 문자열의 복사본을 돌려줍니다. `s` 가 케이스 없는 문자를 포함하거나 결과 문자의 유니코드 범주가 "Lu" (Letter, 대문자) 가 아닌 경우, 예를 들어 "Lt" (Letter, 제목 케이스), `s.upper().isupper()` 가 `False` 일 수 있음에 주의하십시오.

The uppercasing algorithm used is described in section 3.13 'Default Case Folding' of the Unicode Standard.

`str.zfill(width, /)`

길이가 *width* 인 문자열을 만들기 위해 ASCII `'0'` 문자를 왼쪽에 채운 문자열의 복사본을 돌려줍니다. 선행 부호 접두어(`'+'/'-'`)는 부호 문자의 앞이 아니라 뒤 에 채워 넣는 것으로 처리됩니다. *width* 가 `len(s)` 보다 작거나 같은 경우 원래 문자열을 돌려줍니다.

예를 들면:

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

4.9.2 Formatted String Literals (f-strings)

Added in version 3.6.

버전 3.7에서 변경: The `await` and `async for` can be used in expressions within f-strings.

버전 3.8에서 변경: Added the debugging operator (=)

버전 3.12에서 변경: Many restrictions on expressions within f-strings have been removed. Notably, nested strings, comments, and backslashes are now permitted.

An *f-string* (formally a *formatted string literal*) is a string literal that is prefixed with `f` or `F`. This type of string literal allows embedding arbitrary Python expressions within *replacement fields*, which are delimited by curly brackets (`{}`). These expressions are evaluated at runtime, similarly to `str.format()`, and are converted into regular `str` objects. For example:

```
>>> who = 'nobody'
>>> nationality = 'Spanish'
>>> f'{who.title()} expects the {nationality} Inquisition!'
'Nobody expects the Spanish Inquisition!'
```

It is also possible to use a multi line f-string:

```
>>> f'''This is a string
... on two lines'''
'This is a string\non two lines'
```

A single opening curly bracket, `'{'`, marks a *replacement field* that can contain any Python expression:

```
>>> nationality = 'Spanish'
>>> f'The {nationality} Inquisition!'
'The Spanish Inquisition!'
```

To include a literal `{` or `}`, use a double bracket:

```
>>> x = 42
>>> f'{{x}} is {x}'
'{x} is 42'
```

Functions can also be used, and *format specifiers*:

```
>>> from math import sqrt
>>> f'√2 \N{ALMOST EQUAL TO} {sqrt(2):.5f}'
'√2 ≈ 1.41421'
```

Any non-string expression is converted using *str()*, by default:

```
>>> from fractions import Fraction
>>> f'{Fraction(1, 3)}'
'1/3'
```

To use an explicit conversion, use the `!` (exclamation mark) operator, followed by any of the valid formats, which are:

변환	뜻
`!a`	*ascii()*
`!r`	*repr()*
`!s`	*str()*

For example:

```
>>> from fractions import Fraction
>>> f'{Fraction(1, 3)!s}'
'1/3'
>>> f'{Fraction(1, 3)!r}'
'Fraction(1, 3)'
>>> question = '¿Dónde está el Presidente?'
>>> print(f'{question!a}')
'\xbfD\xf3nde est\xe1 el Presidente?'
```

While debugging it may be helpful to see both the expression and its value, by using the equals sign (=) after the expression. This preserves spaces within the brackets, and can be used with a converter. By default, the debugging operator uses the *repr()* (`!r`) conversion. For example:

```
>>> from fractions import Fraction
>>> calculation = Fraction(1, 3)
>>> f'{calculation=}'
'calculation=Fraction(1, 3)'
>>> f'{calculation = }'
'calculation = Fraction(1, 3)'
>>> f'{calculation = !s}'
'calculation = 1/3'
```

Once the output has been evaluated, it can be formatted using a *format specifier* following a colon (`':'`). After the expression has been evaluated, and possibly converted to a string, the `__format__()` method of the result is called with the format specifier, or the empty string if no format specifier is given. The formatted result is then used as the final value for the replacement field. For example:

```
>>> from fractions import Fraction
>>> f'{Fraction(1, 7):.6f}'
'0.142857'
>>> f'{Fraction(1, 7):_^+10}'
'___+1/7___'
```

4.9.3 printf 스타일 문자열 포매팅

> **참고**
>
> The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly).
>
> Using formatted string literals, the `str.format()` interface, or `string.Template` may help avoid these errors. Each of these alternatives provides their own trade-offs and benefits of simplicity, flexibility, and/or extensibility.

String objects have one unique built-in operation: the `%` operator (modulo). This is also known as the string *formatting* or *interpolation* operator. Given `format % values` (where *format* is a string), `%` conversion specifications in *format* are replaced with zero or more elements of *values*. The effect is similar to using the `sprintf()` function in the C language. For example:

```
>>> print('%s has %d quote types.' % ('Python', 2))
Python has 2 quote types.
```

format 이 하나의 인자를 요구하면, *values* 는 하나의 비 튜플 객체 일 수 있습니다.[5] 그렇지 않으면, *values* 는 format 문자열이 지정하는 항목의 수와 같은 튜플이거나 단일 매핑 객체 (예를 들어, 딕셔너리) 이어야 합니다.

변환 명세는 두 개 이상의 문자를 포함하며 다음과 같은 구성 요소들을 포함하는데, 반드시 이 순서대로 나와야 합니다:

1. `'%'` 문자: 명세의 시작을 나타냅니다.
2. 매핑 키 (선택 사항): 괄호로 둘러싸인 문자들의 시퀀스로 구성됩니다 (예를 들어, `(somename)`).
3. 변환 플래그 (선택 사항): 일부 변환 유형의 결과에 영향을 줍니다.
4. 최소 필드 폭 (선택 사항): `'*'` (애스터리스크) 로 지정하면, 실제 폭은 *values* 튜플의 다음 요소에서 읽히고, 변환할 객체는 최소 필드 폭과 선택적 정밀도 뒤에 옵니다.
5. 정밀도 (선택 사항): `'.'` (점) 다음에 정밀도가 옵니다. `'*'` (애스터리스크) 로 지정하면, 실제 정밀도는 *values* 튜플의 다음 요소에서 읽히고, 변환할 값은 정밀도 뒤에 옵니다.
6. 길이 수정자 (선택 사항).
7. 변환 유형.

오른쪽 인자가 딕셔너리 (또는 다른 매핑 형) 인 경우, 문자열에 있는 변환 명세는 반드시 `'%'` 문자 바로 뒤에 그 딕셔너리의 매핑 키를 괄호로 둘러싼 형태로 포함해야 합니다. 매핑 키는 포맷할 값을 매핑으로부터 선택합니다. 예를 들어:

```
>>> print('%(language)s has %(number)03d quote types.' %
...       {'language': "Python", "number": 2})
Python has 002 quote types.
```

이 경우 `*` 지정자를 사용할 수 없습니다 (순차적인 매개변수 목록이 필요하기 때문입니다).

변환 플래그 문자는 다음과 같습니다:

플래그	뜻
`'#'`	값 변환에 "대체 형식" (아래에 정의되어있습니다) 을 사용합니다.
`'0'`	변환은 숫자 값의 경우 0으로 채웁니다.
`'-'`	변환된 값은 왼쪽으로 정렬됩니다 (둘 다 주어지면 `'0'` 변환보다 우선 합니다).
`' '`	(스페이스) 부호 있는 변환 때문에 만들어진 양수 앞에 빈칸을 남겨둡니다 (음수면 빈 문자열입니다).
`'+'`	부호 문자 (`'+'` or `'-'`) 가 변환 앞에 놓입니다 (`' '` 플래그에 우선합니다).

[5] 그래서, 튜플만을 포매팅하려면 포맷할 튜플 하나만을 포함하는 1-튜플을 제공해야 합니다.

길이 수정자 (h, l, L) 를 제공할 수는 있지만, 파이썬에서 필요하지 않기 때문에 무시됩니다 – 예를 들어 `%ld` 는 `%d` 와 같습니다.

변환 유형은 다음과 같습니다:

변환	뜻	노트
'd'	부호 있는 정수 십진 표기.	
'i'	부호 있는 정수 십진 표기.	
'o'	부호 있는 8진수 값.	(1)
'u'	쓸데없는 유형 – 'd' 와 같습니다.	(6)
'x'	부호 있는 16진수 (소문자).	(2)
'X'	부호 있는 16진수 (대문자).	(2)
'e'	Floating-point exponential format (lowercase).	(3)
'E'	Floating-point exponential format (uppercase).	(3)
'f'	Floating-point decimal format.	(3)
'F'	Floating-point decimal format.	(3)
'g'	Floating-point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'G'	Floating-point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'c'	단일 문자 (정수 또는 길이 1인 문자열을 허용합니다).	
'r'	문자열 (*repr()* 을 사용하여 파이썬 객체를 변환합니다).	(5)
's'	문자열 (*str()* 을 사용하여 파이썬 객체를 변환합니다).	(5)
'a'	문자열 (*ascii()* 를 사용하여 파이썬 객체를 변환합니다).	(5)
'%'	인자는 변환되지 않고, 결과에 '%' 문자가 표시됩니다.	

노트:

(1) 대체 형식은 첫 번째 숫자 앞에 선행 8진수 지정자 ('0o')를 삽입합니다.

(2) 대체 형식은 첫 번째 숫자 앞에 선행 '0x' 또는 '0X' ('x' 나 'X' 유형 중 어느 것을 사용하느냐에 따라 달라집니다) 를 삽입합니다.

(3) 대체 형식은 그 뒤에 숫자가 나오지 않더라도 항상 소수점을 포함합니다.

정밀도는 소수점 이하 자릿수를 결정하며 기본값은 6입니다.

(4) 대체 형식은 결과에 항상 소수점을 포함하고 뒤에 오는 0은 제거되지 않습니다.

정밀도는 소수점 앞뒤의 유효 자릿수를 결정하며 기본값은 6입니다.

(5) 정밀도가 N 이라면, 출력은 N 문자로 잘립니다.

(6) **PEP 237**을 참조하세요.

파이썬 문자열은 명시적인 길이를 가지고 있으므로, `%s` 변환은 문자열의 끝이 '\0' 이라고 가정하지 않습니다.

버전 3.1에서 변경: 절댓값이 1e50 을 넘는 숫자에 대한 `%f` 변환은 더는 `%g` 변환으로 대체되지 않습니다.

4.10 바이너리 시퀀스 형 — bytes, bytearray, memoryview

바이너리 데이터를 조작하기 위한 핵심 내장형은 *bytes* 와 *bytearray* 입니다. 이것들은 *memoryview* 에 의해 지원되는데, 다른 바이너리 객체들의 메모리에 복사 없이 접근하기 위해 버퍼 프로토콜을 사용합니다.

array 모듈은 32-비트 정수와 IEEE754 배정도 부동 소수점 같은 기본 데이터형의 효율적인 저장을 지원합니다.

4.10.1 바이트열 객체

바이트열 객체는 단일 바이트들의 불변 시퀀스입니다. 많은 주요 바이너리 프로토콜이 ASCII 텍스트 인코딩을 기반으로 하므로, 바이트열 객체는 ASCII 호환 데이터로 작업 할 때만 유효한 여러 가지 메서드를 제공하며 다양한 다른 방법으로 문자열 객체와 밀접한 관련이 있습니다.

class bytes(*source=b''*)

class bytes(*source, encoding, errors='strict'*)

첫째로, 바이트열 리터럴의 문법은 문자열 리터럴과 거의 같지만 `b` 접두사가 추가된다는 점이 다릅니다.:

- 작은따옴표: `b'still allows embedded "double" quotes'`
- Double quotes: `b"still allows embedded 'single' quotes"`
- 삼중 따옴표: `b'''3 single quotes''',b"""3 double quotes"""`

바이트열 리터럴에는 ASCII 문자만 허용됩니다 (선언된 소스 코드 인코딩과 관계없습니다). 127 보다 큰 바이너리 값은 적절한 이스케이프 시퀀스를 사용하여 바이트열 리터럴에 입력해야 합니다.

문자열 리터럴의 경우와 마찬가지로 바이트열 리터럴은 이스케이프 시퀀스 처리를 비활성화하기 위해 `r` 접두사를 사용할 수도 있습니다. 지원되는 이스케이프 시퀀스를 포함하여 바이트열 리터럴의 다양한 형식에 대한 자세한 내용은 strings 을 참조하십시오.

바이트열 리터럴과 그 표현은 ASCII 텍스트를 기반으로 하지만, 바이트열 객체는 실제로는 정수의 불변 시퀀스처럼 동작하고, 시퀀스의 각 값은 `0 <= x < 256` 이 되도록 제한됩니다 (이 제한을 위반하려고 시도하면 `ValueError` 를 일으킵니다). 이것은 많은 바이너리 형식이 ASCII 기반 요소를 포함하고 일부 텍스트 지향 알고리즘으로 유용하게 조작될 수 있지만, 임의의 바이너리 데이터에 일반적으로 적용될 수는 없음을 강조하기 위한 것입니다 (텍스트 처리 알고리즘을 맹목적으로 ASCII 호환이 아닌 바이너리 데이터 형식에 적용하면 대개 데이터 손상으로 이어집니다).

리터럴 형식 외에도, 바이트열 객체는 여러 가지 다른 방법으로 만들 수 있습니다.:

- 지정된 길이의 0으로 채워진 바이트열 객체: `bytes(10)`
- 정수의 이터러블로부터: `bytes(range(20))`
- 버퍼 프로토콜을 통해 기존 바이너리 데이터 복사: `bytes(obj)`

내장 *bytes* 도 참조하세요.

2 개의 16 진수는 정확히 하나의 바이트에 대응하기 때문에 16 진수는 바이너리 데이터를 설명하는 데 일반적으로 사용되는 형식입니다. 따라서, 바이트열 형은 그 형식의 데이터를 읽는 추가의 클래스 메서드를 갖습니다:

classmethod fromhex(*string, /*)

이 `bytes` 클래스 메서드는 주어진 문자열 객체를 디코딩해서 바이트열 객체를 돌려줍니다. 문자열은 바이트 당 두 개의 16 진수가 포함되어야 하며 ASCII 공백은 무시됩니다.

```
>>> bytes.fromhex('2Ef0 F1f2  ')
b'.\xf0\xf1\xf2'
```

버전 3.7 에서 변경: 이제 `bytes.fromhex()` 는 스페이스뿐만 아니라 문자열에 있는 모든 ASCII 공백을 건너뜁니다.

버전 3.14 에서 변경: `bytes.fromhex()` now accepts ASCII `bytes` and *bytes-like objects* as input.

바이트열 객체를 16 진수 표현으로 변환하기 위한 역변환 함수가 있습니다.

hex(**, bytes_per_sep=1*)

hex(*sep, bytes_per_sep=1*)

인스턴스의 바이트마다 2 자릿수의 16 진수로 표현한 문자열 객체를 돌려줍니다.

```
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

If you want to make the hex string easier to read, you can specify a single character separator *sep* parameter to include in the output. By default, this separator will be included between each byte. A second optional *bytes_per_sep* parameter controls the spacing. Positive values calculate the separator position from the right, negative values from the left.

```
>>> value = b'\xf0\xf1\xf2'
>>> value.hex('-')
'f0-f1-f2'
>>> value.hex('_', 2)
'f0_f1f2'
>>> b'UUDDLRLRAB'.hex(' ', -4)
'55554444 4c524c52 4142'
```

Added in version 3.5.

버전 3.8에서 변경: 이제 `bytes.hex()`는 16진수 출력의 바이트 사이에 구분 기호를 삽입하기 위해 선택적 *sep*과 *bytes_per_sep* 매개 변수를 지원합니다.

바이트열 객체는 정수의 시퀀스(튜플과 유사)이기 때문에, 바이트열 객체 *b* 에 대해서, `b[0]` 는 정수가 됩니다. 반면, `b[0:1]` 는 길이 1인 바이트열 객체가 됩니다. (이것은 인덱싱과 슬라이싱 모두 길이 1인 문자열을 생성하는 텍스트 문자열과 대조됩니다)

바이트열 객체의 표현은 리터럴 형식 (`b'...'`)을 사용하는데, 종종 `bytes([46, 46, 46])` 보다 유용하기 때문입니다. `list(b)` 를 사용하면 바이트열 객체를 항상 정수 리스트로 변환할 수 있습니다.

4.10.2 바이트 배열 객체

`bytearray` 객체는 `bytes` 객체의 가변형입니다.

class bytearray(*source=b''*)

class bytearray(*source, encoding, errors='strict'*)

바이트 배열 객체에 대한 전용 리터럴 문법은 없으며 항상 생성자를 호출하여 만듭니다:

- 빈 인스턴스 만들기: `bytearray()`
- 주어진 길이의 0으로 채워진 인스턴스 만들기: `bytearray(10)`
- 정수의 이터러블로부터: `bytearray(range(20))`
- 버퍼 프로토콜을 통해 기존 바이너리 데이터 복사: `bytearray(b'Hi!')`

바이트 배열 객체는 가변이기 때문에, 바이트열 과 바이트 배열 연산 에 설명되어있는 공통 바이트열과 바이트 배열 연산에 더해, 가변 시퀀스 연산도 지원합니다.

내장 *bytearray* 도 참조하세요.

2개의 16진수는 정확히 하나의 바이트에 대응하기 때문에 16진수는 바이너리 데이터를 설명하는 데 일반적으로 사용되는 형식입니다. 따라서, 바이트 배열형은 그 형식의 데이터를 읽는 추가의 클래스 메서드를 갖습니다:

classmethod fromhex(*string, /*)

이 `bytearray` 클래스 메서드는 주어진 문자열 객체를 디코딩해서 바이트 배열 객체를 돌려줍니다. 문자열은 바이트 당 두 개의 16진수가 포함되어야 하며 ASCII 공백은 무시됩니다.

```
>>> bytearray.fromhex('2Ef0 F1f2  ')
bytearray(b'.\xf0\xf1\xf2')
```

버전 3.7에서 변경: 이제 `bytearray.fromhex()` 는 스페이스뿐만 아니라 문자열에 있는 모든 ASCII 공백을 건너뜁니다.

버전 3.14에서 변경: `bytearray.fromhex()` now accepts ASCII `bytes` and *bytes-like objects* as input.

바이트 배열 객체를 16진수 표현으로 변환하기 위한 역변환 함수가 있습니다.

hex(*, *bytes_per_sep=1*)

hex(*sep*, *bytes_per_sep=1*)

인스턴스의 바이트마다 2 자릿수의 16진수로 표현한 문자열 객체를 돌려줍니다.

```
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

Added in version 3.5.

버전 3.8에서 변경: `bytes.hex()`와 비슷하게, 이제 `bytearray.hex()`는 16진수 출력의 바이트 사이에 구분 기호를 삽입하기 위해 선택적 *sep*과 *bytes_per_sep* 매개 변수를 지원합니다.

resize(*size*, /)

Resize the `bytearray` to contain *size* bytes. *size* must be greater than or equal to 0.

If the `bytearray` needs to shrink, bytes beyond *size* are truncated.

If the `bytearray` needs to grow, all new bytes, those beyond *size*, will be set to null bytes.

This is equivalent to:

```
>>> def resize(ba, size):
...     if len(ba) > size:
...         del ba[size:]
...     else:
...         ba += b'\0' * (size - len(ba))
```

Examples:

```
>>> shrink = bytearray(b'abc')
>>> shrink.resize(1)
>>> (shrink, len(shrink))
(bytearray(b'a'), 1)
>>> grow = bytearray(b'abc')
>>> grow.resize(5)
>>> (grow, len(grow))
(bytearray(b'abc\x00\x00'), 5)
```

Added in version 3.14.

바이트 배열 객체는 정수의 시퀀스(리스트와 유사)이기 때문에, 바이트 배열 객체 *b* 에 대해서, `b[0]` 는 정수가 됩니다. 반면, `b[0:1]` 는 길이 1인 바이트 배열 객체가 됩니다. (이것은 인덱싱과 슬라이싱 모두 길이 1인 문자열을 생성하는 텍스트 문자열과 대조됩니다)

바이트 배열 객체의 표현은 바이트열 리터럴 형식 (`bytearray(b'...')`) 을 사용하는데, 종종 `bytearray([46, 46, 46])` 보다 유용하기 때문입니다. `list(b)` 를 사용하면 바이트 배열 객체를 항상 정수 리스트로 변환할 수 있습니다.

4.10.3 바이트열 과 바이트 배열 연산

바이트열와 바이트 배열 객체는 공통 시퀀스 연산을 지원합니다. 이것들은 같은 형의 피연산자뿐만 아니라 모든 *bytes-like object*와 상호 운용됩니다. 이러한 유연성으로 인해, 오류 없이 작업을 자유롭게 혼합할 수 있습니다. 그러나, 결과의 반환형은 피연산자의 순서에 따라 달라질 수 있습니다.

참고

바이트열 및 바이트 배열 객체의 메서드는 인자로 문자열을 받아들이지 않습니다, 문자열의 메서드가 바이트열을 인자로 허용하지 않는 것과 마찬가지입니다. 예를 들어, 다음과 같이 작성해야 합니다:

```
a = "abc"
b = a.replace("a", "f")
```

그리고:

```
a = b"abc"
b = a.replace(b"a", b"f")
```

일부 바이트열 및 바이트 배열 연산은 ASCII 호환 바이너리 형식을 가정하므로, 임의의 바이너리 데이터로 작업 할 때는 피해야 합니다. 이러한 제한 사항은 아래에서 다룹니다.

> **참고**
>
> 이러한 ASCII 기반 연산을 사용하여 ASCII 기반 형식으로 저장되지 않은 바이너리 데이터를 조작하면 데이터가 손상될 수 있습니다.

바이트열 및 바이트 배열 객체에 대한 다음 메서드는 임의의 바이너리 데이터와 함께 사용할 수 있습니다.

`bytes.count`(*sub*[, *start*[, *end*]])

`bytearray.count`(*sub*[, *start*[, *end*]])

범위 [*start*, *end*] 에서 서브 시퀀스 *sub* 가 중첩되지 않고 등장하는 횟수를 돌려줍니다. 선택적 인자 *start* 와 *end* 는 슬라이스 표기법으로 해석됩니다.

검색할 서브 시퀀스는 임의의 *bytes-like object* 또는 0에서 255 사이의 정수일 수 있습니다.

If *sub* is empty, returns the number of empty slices between characters which is the length of the bytes object plus one.

버전 3.3에서 변경: 서브 시퀀스로 0에서 255 사이의 정수도 허용합니다.

`bytes.removeprefix`(*prefix*, /)

`bytearray.removeprefix`(*prefix*, /)

바이너리 데이터가 *prefix* 문자열로 시작하면, `bytes[len(prefix):]`를 반환합니다. 그렇지 않으면, 원래 바이너리 데이터의 사본을 반환합니다:

```
>>> b'TestHook'.removeprefix(b'Test')
b'Hook'
>>> b'BaseTestCase'.removeprefix(b'Test')
b'BaseTestCase'
```

*prefix*는 임의의 바이트열류 객체 일 수 있습니다.

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

Added in version 3.9.

`bytes.removesuffix`(*suffix*, /)

`bytearray.removesuffix`(*suffix*, /)

바이너리 데이터가 *suffix* 문자열로 끝나고 해당 *suffix*가 비어 있지 않으면 `bytes[:-len(suffix)]`를 반환합니다. 그렇지 않으면, 원래 바이너리 데이터의 사본을 반환합니다:

```
>>> b'MiscTests'.removesuffix(b'Tests')
b'Misc'
>>> b'TmpDirMixin'.removesuffix(b'Tests')
b'TmpDirMixin'
```

*suffix*는 임의의 바이트열류 객체 일 수 있습니다.

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

Added in version 3.9.

bytes.**decode**(*encoding='utf-8'*, *errors='strict'*)

bytearray.**decode**(*encoding='utf-8'*, *errors='strict'*)

Return the bytes decoded to a `str`.

encoding defaults to `'utf-8'`; see 표준 인코딩 for possible values.

errors controls how decoding errors are handled. If `'strict'` (the default), a `UnicodeError` exception is raised. Other possible values are `'ignore'`, `'replace'`, and any other name registered via `codecs.register_error()`. See 에러 처리기 for details.

For performance reasons, the value of *errors* is not checked for validity unless a decoding error actually occurs, 파이썬 개발 모드 is enabled or a debug build is used.

> **참고**
>
> Passing the *encoding* argument to `str` allows decoding any *bytes-like object* directly, without needing to make a temporary `bytes` or `bytearray` object.

버전 3.1에서 변경: 키워드 인자 지원이 추가되었습니다.

버전 3.9에서 변경: The value of the *errors* argument is now checked in 파이썬 개발 모드 and in debug mode.

bytes.**endswith**(*suffix*[, *start*[, *end*]])

bytearray.**endswith**(*suffix*[, *start*[, *end*]])

바이너리 데이터가 지정된 *suffix* 로 끝나면 `True` 를 돌려주고, 그렇지 않으면 `False` 를 돌려줍니다. *suffix* 는 찾고자 하는 접미사들의 튜플이 될 수도 있습니다. 선택적 *start* 가 제공되면 그 위치에서 검사를 시작합니다. 선택적 *end* 를 사용하면 해당 위치에서 비교를 중단합니다.

검색할 접미사(들)는 임의의 *bytes-like object* 일 수 있습니다.

bytes.**find**(*sub*[, *start*[, *end*]])

bytearray.**find**(*sub*[, *start*[, *end*]])

서브 시퀀스 *sub* 가 슬라이스 `s[start:end]` 내에 등장하는 가장 작은 데이터의 인덱스를 돌려줍니다. 선택적 인자 *start* 와 *end* 는 슬라이스 표기법으로 해석됩니다. *sub* 가 없으면 `-1` 을 돌려줍니다.

검색할 서브 시퀀스는 임의의 *bytes-like object* 또는 0에서 255 사이의 정수일 수 있습니다.

> **참고**
>
> `find()` 메서드는 *sub* 의 위치를 알아야 할 경우에만 사용해야 합니다. *sub* 가 부분 문자열인지 여부를 확인하려면 `in` 연산자를 사용하십시오:
>
> ```
> >>> b'Py' in b'Python'
> True
> ```

버전 3.3에서 변경: 서브 시퀀스로 0에서 255 사이의 정수도 허용합니다.

bytes.**index**(*sub*[, *start*[, *end*]])

bytearray.**index**(*sub*[, *start*[, *end*]])

`find()` 과 비슷하지만, 서브 시퀀스를 찾을 수 없는 경우 `ValueError` 를 일으킵니다.

검색할 서브 시퀀스는 임의의 *bytes-like object* 또는 0에서 255 사이의 정수일 수 있습니다.

버전 3.3에서 변경: 서브 시퀀스로 0에서 255 사이의 정수도 허용합니다.

bytes.**join**(*iterable*, /)

bytearray.**join**(*iterable*, /)

iterable 의 바이너리 데이터 시퀀스들을 이어 붙이기 한 바이트열 또는 바이트 배열 객체를 돌려줍니다. *iterable* 에 `str` 객체나 기타 *bytes-like object* 가 아닌 값이 있으면 `TypeError` 를 일으킵니다. 요소들 사이의 구분자는 이 메서드를 제공하는 바이트열 이나 바이트 배열 객체입니다.

static bytes.**maketrans**(*from*, *to*, /)

static bytearray.**maketrans**(*from*, *to*, /)

이 정적 메서드는 `bytes.translate()` 에 사용할 수 있는 변환표를 돌려주는데, *from* 에 있는 문자를 *to* 의 같은 위치에 있는 문자로 매핑합니다; *from* 과 *to* 는 모두 *bytes-like object* 여야 하고 길이가 같아야 합니다.

Added in version 3.1.

bytes.**partition**(*sep*, /)

bytearray.**partition**(*sep*, /)

sep 가 처음 나타나는 위치에서 시퀀스를 나누고, 구분자 앞에 있는 부분, 구분자 자체, 구분자 뒤에 오는 부분으로 구성된 3-튜플을 돌려줍니다. 구분자가 발견되지 않으면, 원래 시퀀스의 복사본과 그 뒤를 따르는 두 개의 빈 바이트열 또는 바이트 배열 객체로 구성된 3-튜플을 돌려줍니다.

검색할 구분자는 임의의 *bytes-like object* 일 수 있습니다.

bytes.**replace**(*old*, *new*, *count=-1*, /)

bytearray.**replace**(*old*, *new*, *count=-1*, /)

모든 서브 시퀀스 *old* 가 *new* 로 치환된 시퀀스의 복사본을 돌려줍니다. 선택적 인자 *count* 가 주어지면, 앞의 *count* 개만 치환됩니다.

검색할 서브 시퀀스와 그 대체물은 임의의 *bytes-like object* 일 수 있습니다.

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

bytes.**rfind**(*sub*[, *start*[, *end*]])

bytearray.**rfind**(*sub*[, *start*[, *end*]])

서브 시퀀스 *sub* 가 `s[start:end]` 내에 등장하는 가장 큰 시퀀스의 인덱스를 돌려줍니다. 선택적 인자 *start* 와 *end* 는 슬라이스 표기법으로 해석됩니다. 실패하면 `-1` 을 돌려줍니다.

검색할 서브 시퀀스는 임의의 *bytes-like object* 또는 0에서 255 사이의 정수일 수 있습니다.

버전 3.3에서 변경: 서브 시퀀스로 0에서 255 사이의 정수도 허용합니다.

bytes.**rindex**(*sub*[, *start*[, *end*]])

bytearray.**rindex**(*sub*[, *start*[, *end*]])

`rfind()` 와 비슷하지만, 서브 시퀀스 *sub* 를 찾을 수 없는 경우 `ValueError` 를 일으킵니다.

검색할 서브 시퀀스는 임의의 *bytes-like object* 또는 0에서 255 사이의 정수일 수 있습니다.

버전 3.3에서 변경: 서브 시퀀스로 0에서 255 사이의 정수도 허용합니다.

bytes.**rpartition**(*sep*, /)

bytearray.**rpartition**(*sep*, /)

sep 가 마지막으로 나타나는 위치에서 시퀀스를 나누고, 구분자 앞에 있는 부분, 구분자 자체, 구분자 뒤에 오는 부분으로 구성된 3-튜플을 돌려줍니다. 구분자가 발견되지 않으면, 두 개의 빈 바이트열 또는 바이트 배열 객체와 그 뒤를 따르는 원래 시퀀스의 복사본으로 구성된 3-튜플을 돌려줍니다.

검색할 구분자는 임의의 *bytes-like object* 일 수 있습니다.

bytes.**startswith**(*prefix*[, *start*[, *end*]])

bytearray.**startswith**(*prefix*[, *start*[, *end*]])

바이너리 데이터가 지정된 *prefix* 로 시작하면 `True` 를 돌려주고, 그렇지 않으면 `False` 를 돌려줍니다. *prefix* 는 찾고자 하는 접두사들의 튜플이 될 수도 있습니다. 선택적 *start* 가 제공되면 그 위치에서 검사를 시작합니다. 선택적 *end* 를 사용하면 해당 위치에서 비교를 중단합니다.

검색할 접두사(들)는 임의의 *bytes-like object* 일 수 있습니다.

bytes.**translate**(*table*, /, *delete=b''*)

bytearray.**translate**(*table*, /, *delete=b''*)

생략 가능한 인자 *delete* 의 모든 바이트를 제거하고, 나머지 바이트들을 주어진 변환표로 매핑한 바이트열이나 바이트 배열 객체의 복사본을 돌려줍니다. table은 길이 256인 바이트열 객체이어야 합니다.

`bytes.maketrans()` 메서드를 사용하여 변환표를 만들 수 있습니다.

문자를 지우기만 하는 변환에는 *table* 인자를 `None` 으로 설정하십시오:

```
>>> b'read this short text'.translate(None, b'aeiou')
b'rd ths shrt txt'
```

버전 3.6에서 변경: 이제 *delete* 는 키워드 인자로 지원됩니다.

바이트열 및 바이트 배열 객체에 대한 다음 메서드는 ASCII 호환 바이너리 형식의 사용을 가정하는 기본 동작을 갖지만, 적절한 인자를 전달하여 임의의 바이너리 데이터와 함께 사용할 수 있습니다. 이 섹션의 바이트 배열 메서드는 모두 제자리에서 작동하지 않고 대신 새로운 객체를 생성함에 주의하십시오.

bytes.**center**(*width*, *fillbyte=b' '*, /)

bytearray.**center**(*width*, *fillbyte=b' '*, /)

길이 *width* 인 시퀀스의 가운데에 정렬한 객체의 복사본을 돌려줍니다. 지정된 *fillbyte* (기본값은 ASCII 스페이스)를 사용하여 채웁니다. `bytes` 객체의 경우, *width* 가 `len(s)` 보다 작거나 같은 경우 원래 시퀀스가 반환됩니다.

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

bytes.**ljust**(*width*, *fillbyte=b' '*, /)

bytearray.**ljust**(*width*, *fillbyte=b' '*, /)

왼쪽으로 정렬된 객체의 복사본을 길이 *width* 인 시퀀스로 돌려줍니다. 지정된 *fillbyte* (기본값은 ASCII 스페이스)을 사용하여 채웁니다. `bytes` 객체의 경우, *width* 가 `len(s)` 보다 작거나 같은 경우 원래 시퀀스가 반환됩니다.

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

bytes.**lstrip**(*bytes=None*, /)

bytearray.**lstrip**(*bytes=None, /*)

Return a copy of the sequence with specified leading bytes removed. The *bytes* argument is a binary sequence specifying the set of byte values to be removed. If omitted or `None`, the *bytes* argument defaults to removing ASCII whitespace. The *bytes* argument is not a prefix; rather, all combinations of its values are stripped:

```
>>> b'   spacious   '.lstrip()
b'spacious   '
>>> b'www.example.com'.lstrip(b'cmowz.')
b'example.com'
```

제거할 바이트 값의 바이너리 시퀀스는 임의의 바이트열류 객체일 수 있습니다. 문자 집합의 모든 것이 아닌 단일 접두사 문자열을 제거하는 메서드는 *removeprefix()*를 참조하십시오. 예를 들면:

```
>>> b'Arthur: three!'.lstrip(b'Arthur: ')
b'ee!'
>>> b'Arthur: three!'.removeprefix(b'Arthur: ')
b'three!'
```

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

bytes.**rjust**(*width, fillbyte=b' ', /*)

bytearray.**rjust**(*width, fillbyte=b' ', /*)

오른쪽으로 정렬된 객체의 복사본을 길이 *width* 인 시퀀스로 돌려줍니다. 지정된 *fillbyte* (기본값은 ASCII 스페이스)를 사용하여 채웁니다. *bytes* 객체의 경우, *width* 가 `len(s)` 보다 작거나 같은 경우 원래 시퀀스가 반환됩니다.

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

bytes.**rsplit**(*sep=None, maxsplit=-1*)

bytearray.**rsplit**(*sep=None, maxsplit=-1*)

sep 을 구분자 시퀀스로 사용하여 바이너리 시퀀스를 같은 형의 서브 시퀀스로 나눕니다. *maxsplit* 이 주어지면 가장 오른쪽에서 최대 *maxsplit* 번의 분할이 수행됩니다. *sep* 이 지정되지 않거나 `None` 이면, ASCII 공백 문자만으로 이루어진 모든 서브 시퀀스는 구분자입니다. 오른쪽에서 분리하는 것을 제외하면, *rsplit()*는 아래에서 자세히 설명될 *split()*처럼 동작합니다.

bytes.**rstrip**(*bytes=None, /*)

bytearray.**rstrip**(*bytes=None, /*)

Return a copy of the sequence with specified trailing bytes removed. The *bytes* argument is a binary sequence specifying the set of byte values to be removed. If omitted or `None`, the *bytes* argument defaults to removing ASCII whitespace. The *bytes* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> b'   spacious   '.rstrip()
b'   spacious'
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

제거할 바이트 값의 바이너리 시퀀스는 임의의 바이트열류 객체일 수 있습니다. 문자 집합의 모든 것이 아닌 단일 접미사 문자열을 제거하는 메서드는 *removesuffix()*를 참조하십시오. 예를 들면:

```
>>> b'Monty Python'.rstrip(b' Python')
b'M'
>>> b'Monty Python'.removesuffix(b' Python')
b'Monty'
```

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

bytes.**split**(*sep=None, maxsplit=-1*)

bytearray.**split**(*sep=None, maxsplit=-1*)

sep 를 구분자 시퀀스로 사용하여 바이너리 시퀀스를 같은 형의 서브 시퀀스로 나눕니다. *maxsplit* 이 지정되고 음수가 아닌 경우, 최대 *maxsplit* 분할이 수행됩니다 (따라서, 리스트는 최대 `maxsplit+1` 개의 요소를 가지게 됩니다). *maxsplit* 이 지정되지 않았거나 `-1` 이라면 분할 수에 제한이 없습니다 (가능한 모든 분할이 만들어집니다).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty subsequences (for example, `b'1,,2'.split(b',')` returns `[b'1', b'', b'2']`). The *sep* argument may consist of a multibyte sequence as a single delimiter. Splitting an empty sequence with a specified separator returns `[b'']` or `[bytearray(b'')]` depending on the type of object being split. The *sep* argument may be any *bytes-like object*.

예를 들면:

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,2,,3,'.split(b',')
[b'1', b'2', b'', b'3', b'']
>>> b'1<>2<>3<4'.split(b'<>')
[b'1', b'2', b'3<4']
```

sep 이 지정되지 않거나 `None` 이면, 다른 분할 알고리즘이 적용됩니다: 연속된 ASCII 공백 문자는 단일한 구분자로 간주하고, 시퀀스가 선행이나 후행 공백을 포함해도 결과는 시작과 끝에 빈 시퀀스를 포함하지 않습니다. 결과적으로, 빈 시퀀스나 ASCII 공백만으로 구성된 시퀀스를 `None` 구분자로 나누면 `[]` 를 돌려줍니다.

예를 들면:

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b'   1   2   3   '.split()
[b'1', b'2', b'3']
```

bytes.**strip**(*bytes=None, /*)

bytearray.**strip**(*bytes=None, /*)

Return a copy of the sequence with specified leading and trailing bytes removed. The *bytes* argument is a binary sequence specifying the set of byte values to be removed. If omitted or `None`, the *bytes* argument defaults to removing ASCII whitespace. The *bytes* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```
>>> b'   spacious   '.strip()
b'spacious'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> b'www.example.com'.strip(b'cmowz.')
b'example'
```

제거할 바이트 값의 바이너리 시퀀스는 임의의 *bytes-like object* 일 수 있습니다.

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

바이트열 및 바이트 배열 객체에 대한 다음 메서드는 ASCII 호환 바이너리 형식의 사용을 가정하며 임의의 바이너리 데이터에 적용하면 안 됩니다. 이 섹션의 바이트 배열 메서드는 모두 제자리에서 작동하지 않고 대신 새로운 객체를 생성합니다.

`bytes.`**`capitalize`**`()`

`bytearray.`**`capitalize`**`()`

각 바이트가 ASCII 문자로 해석되고 첫 번째 바이트는 대문자로, 나머지는 소문자로 만든 시퀀스의 복사본을 돌려줍니다. ASCII 바이트가 아닌 값들은 변경되지 않고 전달됩니다.

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

`bytes.`**`expandtabs`**`(`*tabsize=8*`)`

`bytearray.`**`expandtabs`**`(`*tabsize=8*`)`

모든 ASCII 탭 문자들을 현재의 열과 주어진 탭 크기에 따라 하나나 그 이상의 ASCII 스페이스로 치환한 시퀀스의 복사본을 돌려줍니다. 탭 위치는 *tabsize* 바이트마다 발생합니다 (기본값은 8 이고, 열 0, 8, 16 등에 탭 위치를 지정합니다). 시퀀스를 확장하기 위해 현재 열이 0 으로 설정되고 시퀀스를 바이트 단위로 검사합니다. 바이트가 ASCII 탭 문자 (`b'\t'`) 이면, 현재 열이 다음 탭 위치와 같아질 때까지 하나 이상의 스페이스 문자가 삽입됩니다. (탭 문자 자체는 복사되지 않습니다.) 현재 바이트가 ASCII 개행 문자 (`b'\n'`) 또는 캐리지 리턴 (`b'\r'`) 이면 복사되고 현재 열은 0 으로 재설정됩니다. 다른 바이트는 변경되지 않고 복사되고 현재 열은 인쇄할 때 바이트가 어떻게 표시되는지에 관계없이 1 씩 증가합니다.

```
>>> b'01\t012\t0123\t01234'.expandtabs()
b'01      012     0123    01234'
>>> b'01\t012\t0123\t01234'.expandtabs(4)
b'01  012 0123    01234'
```

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

`bytes.`**`isalnum`**`()`

`bytearray.`**`isalnum`**`()`

시퀀스의 모든 바이트가 알파벳 ASCII 문자 또는 ASCII 십진수이고 시퀀스가 비어 있지 않으면 `True` 를 돌려주고 그렇지 않으면 `False` 를 돌려줍니다. 알파벳 ASCII 문자는, 시퀀스 `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'` 에 있는 바이트 값입니다. ASCII 십진수는 시퀀스 `b'0123456789'` 에 있는 바이트 값입니다.

예를 들면:

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

bytes.**isalpha**()

bytearray.**isalpha**()

시퀀스의 모든 바이트가 알파벳 ASCII 문자이고 시퀀스가 비어 있지 않으면 True를 돌려주고 그렇지 않으면 False를 돌려줍니다. 알파벳 ASCII 문자는, 시퀀스 b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ' 에 있는 바이트 값입니다.

예를 들면:

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

bytes.**isascii**()

bytearray.**isascii**()

시퀀스가 비어 있거나 시퀀스의 모든 바이트가 ASCII면 True를 돌려주고, 그렇지 않으면 False를 돌려줍니다. ASCII 바이트의 범위는 0-0x7F 입니다.

Added in version 3.7.

bytes.**isdigit**()

bytearray.**isdigit**()

시퀀스의 모든 바이트가 ASCII 십진수이며 시퀀스가 비어 있지 않으면 True를 돌려주고 그렇지 않으면 False를 돌려줍니다. ASCII 십진수는 시퀀스 b'0123456789' 에 있는 바이트 값입니다.

예를 들면:

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

bytes.**islower**()

bytearray.**islower**()

시퀀스에 적어도 하나의 ASCII 소문자가 있고, ASCII 대문자가 없으면 True를, 그렇지 않으면 False를 돌려줍니다.

예를 들면:

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

ASCII 소문자는 시퀀스 b'abcdefghijklmnopqrstuvwxyz' 에 있는 바이트 값입니다. ASCII 대문자는, 시퀀스 b'ABCDEFGHIJKLMNOPQRSTUVWXYZ' 에 있는 바이트 값입니다.

bytes.**isspace**()

bytearray.**isspace**()

시퀀스의 모든 바이트가 ASCII 공백이고, 시퀀스가 비어 있지 않으면 True를 돌려주고 그렇지 않으면 False를 돌려줍니다. ASCII 공백 문자는 시퀀스 b' \t\n\r\x0b\f'(스페이스, 탭, 줄 바꿈, 캐리지 리턴, 수직 탭, 폼 피드)에 있는 바이트 값입니다.

bytes.**istitle**()

bytearray.**istitle**()

시퀀스가 ASCII 제목 케이스고 시퀀스가 비어있지 않으면 `True`를 돌려주고 그렇지 않으면 `False`를 돌려줍니다. "제목 케이스" 의 정의에 대한 자세한 내용은 *bytes.title()* 을 참조하십시오.

예를 들면:

```
>>> b'Hello World'.istitle()
True
>>> b'Hello world'.istitle()
False
```

bytes.**isupper**()

bytearray.**isupper**()

시퀀스에 적어도 하나의 ASCII 대문자가 있고, ASCII 소문자가 없으면 `True`를, 그렇지 않으면 `False`를 돌려줍니다.

예를 들면:

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

ASCII 소문자는 시퀀스 `b'abcdefghijklmnopqrstuvwxyz'` 에 있는 바이트 값입니다. ASCII 대문자는, 시퀀스 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 에 있는 바이트 값입니다.

bytes.**lower**()

bytearray.**lower**()

모든 ASCII 대문자를 해당 소문자로 변환한 시퀀스의 복사본을 돌려줍니다.

예를 들면:

```
>>> b'Hello World'.lower()
b'hello world'
```

ASCII 소문자는 시퀀스 `b'abcdefghijklmnopqrstuvwxyz'` 에 있는 바이트 값입니다. ASCII 대문자는, 시퀀스 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 에 있는 바이트 값입니다.

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

bytes.**splitlines**(*keepends=False*)

bytearray.**splitlines**(*keepends=False*)

ASCII 줄 경계에서 나눈 바이너리 시퀀스의 줄 리스트를 돌려줍니다. 이 메서드는 줄을 나누는데 *universal newlines* 접근법을 사용합니다. *keepends* 가 참으로 주어지지 않는 한 결과 리스트에 줄 바꿈은 포함되지 않습니다.

예를 들면:

```
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
[b'ab c\n', b'\n', b'de fg\r', b'kl\r\n']
```

구분자 시퀀스 *sep* 이 주어졌을 때 *split()* 와 달리, 이 메서드는 빈 시퀀스에 대해서 빈 리스트를 돌려주고, 마지막 줄 바꿈은 새 줄을 만들지 않습니다:

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([b''], [b'Two lines', b''])
>>> b"".splitlines(), b"One line\n".splitlines()
([], [b'One line'])
```

bytes.**swapcase**()

bytearray.**swapcase**()

모든 ASCII 소문자를 해당 대문자로, 그 반대도 마찬가지로 변환한 시퀀스의 복사본을 돌려줍니다.

예를 들면:

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

ASCII 소문자는 시퀀스 `b'abcdefghijklmnopqrstuvwxyz'` 에 있는 바이트 값입니다. ASCII 대문자는, 시퀀스 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 에 있는 바이트 값입니다.

Unlike *str.swapcase()*, it is always the case that `bin.swapcase().swapcase() == bin` for the binary versions. Case conversions are symmetrical in ASCII, even though that is not generally true for arbitrary Unicode code points.

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

bytes.**title**()

bytearray.**title**()

단어가 ASCII 대문자로 시작하고 나머지 문자들은 소문자인 제목 케이스 버전의 바이너리 시퀀스를 돌려줍니다. 케이스 없는 바이트 값은 수정되지 않은 상태로 남습니다.

예를 들면:

```
>>> b'Hello world'.title()
b'Hello World'
```

ASCII 소문자는 시퀀스 `b'abcdefghijklmnopqrstuvwxyz'` 에 있는 바이트 값입니다. ASCII 대문자는 시퀀스 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 에 있는 바이트 값입니다. 다른 모든 바이트 값은 케이스가 없습니다.

이 알고리즘은 단어를 글자들의 연속으로 보는 간단한 언어 독립적 정의를 사용합니다. 이 정의는 여러 상황에서 작동하지만, 축약과 소유의 아포스트로피가 단어 경계를 형성한다는 것을 의미하고, 이는 원하는 결과가 아닐 수도 있습니다:

```
>>> b"they're bill's friends from the UK".title()
b"They're Bill'S Friends From The Uk"
```

정규식을 사용하여 아포스트로피에 대한 해결 방법을 구성할 수 있습니다:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(rb"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0:1].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase(b"they're bill's friends.")
b"They're Bill's Friends."
```

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

bytes.**upper**()

bytearray.**upper**()

모든 ASCII 소문자를 해당 대문자로 변환한 시퀀스의 복사본을 돌려줍니다.

예를 들면:

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

ASCII 소문자는 시퀀스 `b'abcdefghijklmnopqrstuvwxyz'` 에 있는 바이트 값입니다. ASCII 대문자는, 시퀀스 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 에 있는 바이트 값입니다.

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

bytes.**zfill**(*width*, /)

bytearray.**zfill**(*width*, /)

길이가 *width* 인 시퀀스를 만들기 위해 ASCII `b'0'` 문자를 왼쪽에 채운 시퀀스의 복사본을 돌려줍니다. 선행 부호 접두어(`b'+'`/`b'-'`)는 부호 문자의 앞이 아니라 뒤 에 채우는 것으로 처리됩니다. *bytes* 객체의 경우, *width* 가 `len(s)` 보다 작거나 같은 경우 원래 시퀀스를 돌려줍니다.

예를 들면:

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

> **참고**
>
> 이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

4.10.4 printf 스타일 바이너리 포매팅

> **참고**
>
> 여기에 설명된 포맷 연산은 여러 가지 일반적인 오류를 (예를 들어 튜플과 딕셔너리를 올바르게 표시하지 못하는 것) 유발하는 다양한 문제점들이 있습니다. 인쇄될 값이 튜플 또는 딕셔너리일 경우 튜플로 감싸야 합니다.

바이너리 시퀀스 객체는 한가지 고유한 내장 연산을 갖고 있습니다: `%` 연산자 (모듈로). 이것은 바이너리 포매팅 또는 치환 연산자라고도 합니다. `format % values` 가 주어질 때 (*format* 은 바이너리 시퀀스입니다), *format* 내부의 `%` 변환 명세는 0개 이상의 *values* 의 요소로 대체됩니다. 이 효과는 C 언어에서 `sprintf()` 를 사용하는 것과 비슷합니다.

format 이 하나의 인자를 요구하면, *values* 는 하나의 비 튜플 객체 일 수 있습니다.[Page 66, 5] 그렇지 않으면, *values* 는 format 바이너리 시퀀스 객체가 지정하는 항목의 수와 같은 튜플이거나 단일 매핑 객체 (예를 들어, 딕셔너리) 여야 합니다.

변환 명세는 두 개 이상의 문자를 포함하며 다음과 같은 구성 요소들을 포함하는데, 반드시 이 순서대로 나와야 합니다:

1. `'%'` 문자: 명세의 시작을 나타냅니다.
2. 매핑 키 (선택 사항): 괄호로 둘러싸인 문자들의 시퀀스로 구성됩니다 (예를 들어, `(somename)`).
3. 변환 플래그 (선택 사항): 일부 변환 유형의 결과에 영향을 줍니다.
4. 최소 필드 폭 (선택 사항): `'*'` (애스터리스크) 로 지정하면, 실제 폭은 *values* 튜플의 다음 요소에서 읽히고, 변환할 객체는 최소 필드 폭과 선택적 정밀도 뒤에 옵니다.
5. 정밀도 (선택 사항): `'.'` (점) 다음에 정밀도가 옵니다. `'*'` (애스터리스크) 로 지정하면, 실제 정밀도는 *values* 튜플의 다음 요소에서 읽히고, 변환할 값은 정밀도 뒤에 옵니다.
6. 길이 수정자 (선택 사항).
7. 변환 유형.

오른쪽 인자가 딕셔너리 (또는 다른 매핑 형) 인 경우, 바이너리 시퀀스 객체에 있는 변환 명세는 반드시 `'%'` 문자 바로 뒤에 그 딕셔너리의 매핑 키를 괄호로 둘러싼 형태로 포함해야 합니다. 매핑 키는 포맷할 값을 매핑으로 부터 선택합니다. 예를 들어:

```
>>> print(b'%(language)s has %(number)03d quote types.' %
...       {b'language': b"Python", b"number": 2})
b'Python has 002 quote types.'
```

이 경우 `*` 지정자를 사용할 수 없습니다 (순차적인 매개변수 목록이 필요하기 때문입니다).

변환 플래그 문자는 다음과 같습니다:

플래그	뜻
`'#'`	값 변환에 "대체 형식" (아래에 정의되어있습니다) 을 사용합니다.
`'0'`	변환은 숫자 값의 경우 0으로 채웁니다.
`'-'`	변환된 값은 왼쪽으로 정렬됩니다 (둘 다 주어지면 `'0'` 변환보다 우선 합니다).
`' '`	(스페이스) 부호 있는 변환 때문에 만들어진 양수 앞에 빈칸을 남겨둡니다 (음수면 빈 문자열입니다).
`'+'`	부호 문자 (`'+'` or `'-'`) 가 변환 앞에 놓입니다 (`' '` 플래그에 우선합니다).

길이 수정자 (`h`, `l`, `L`) 를 제공할 수는 있지만, 파이썬에서 필요하지 않기 때문에 무시됩니다 – 예를 들어 `%ld` 는 `%d` 와 같습니다.

변환 유형은 다음과 같습니다:

변환	뜻	노트
'd'	부호 있는 정수 십진 표기.	
'i'	부호 있는 정수 십진 표기.	
'o'	부호 있는 8진수 값.	(1)
'u'	쓸데없는 유형 – 'd' 와 같습니다.	(8)
'x'	부호 있는 16진수 (소문자).	(2)
'X'	부호 있는 16진수 (대문자).	(2)
'e'	Floating-point exponential format (lowercase).	(3)
'E'	Floating-point exponential format (uppercase).	(3)
'f'	Floating-point decimal format.	(3)
'F'	Floating-point decimal format.	(3)
'g'	Floating-point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'G'	Floating-point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.	(4)
'c'	단일 바이트 (정수 또는 길이 1인 바이너리 시퀀스를 허용합니다).	
'b'	Bytes (any object that follows the buffer protocol or has __bytes__()).	(5)
's'	's' 는 'b' 의 별칭이고 파이썬 2/3에서만 사용되어야 합니다.	(6)
'a'	Bytes (converts any Python object using repr(obj).encode('ascii', 'backslashreplace')).	(5)
'r'	'r' 는 'a' 의 별칭이고 파이썬 2/3에서만 사용되어야 합니다.	(7)
'%'	인자는 변환되지 않고, 결과에 '%' 문자가 표시됩니다.	

노트:

(1) 대체 형식은 첫 번째 숫자 앞에 선행 8진수 지정자 ('0o')를 삽입합니다.

(2) 대체 형식은 첫 번째 숫자 앞에 선행 '0x' 또는 '0X' ('x' 나 'X' 유형 중 어느 것을 사용하느냐에 따라 달라집니다) 를 삽입합니다.

(3) 대체 형식은 그 뒤에 숫자가 나오지 않더라도 항상 소수점을 포함합니다.

정밀도는 소수점 이하 자릿수를 결정하며 기본값은 6입니다.

(4) 대체 형식은 결과에 항상 소수점을 포함하고 뒤에 오는 0은 제거되지 않습니다.

정밀도는 소수점 앞뒤의 유효 자릿수를 결정하며 기본값은 6입니다.

(5) 정밀도가 N 이라면, 출력은 N 문자로 잘립니다.

(6) b'%s' 는 폐지되었습니다. 하지만 3.x 시리즈에서는 제거되지 않습니다.

(7) b'%r' 는 폐지되었습니다. 하지만 3.x 시리즈에서는 제거되지 않습니다.

(8) **PEP 237**을 참조하세요.

참고

이 메서드의 바이트 배열 버전은 제자리에서 동작하지 않습니다 - 변경되지 않는 경우조차 항상 새 객체를 만듭니다.

더 보기

PEP 461 - bytes와 bytearray에 % 포매팅 추가

Added in version 3.5.

4.10.5 메모리 뷰

memoryview 객체는 파이썬 코드가 버퍼 프로토콜 을 지원하는 객체의 내부 데이터에 복사 없이 접근할 수 있게 합니다.

class memoryview(*object*)

Create a *memoryview* that references *object*. *object* must support the buffer protocol. Built-in objects that support the buffer protocol include *bytes* and *bytearray*.

A *memoryview* has the notion of an *element*, which is the atomic memory unit handled by the originating *object*. For many simple types such as *bytes* and *bytearray*, an element is a single byte, but other types such as *array.array* may have bigger elements.

`len(view)` is equal to the length of *tolist*, which is the nested list representation of the view. If `view.ndim = 1`, this is equal to the number of elements in the view.

버전 3.12에서 변경: If `view.ndim == 0`, `len(view)` now raises *TypeError* instead of returning 1.

The *itemsize* attribute will give you the number of bytes in a single element.

memoryview 는 슬라이싱과 인덱싱을 지원하여 데이터를 노출합니다. 일차원 슬라이스는 서브 뷰를 만듭니다:

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

format 이 *struct* 모듈의 네이티브 형식 지정자 중 하나인 경우, 정수 또는 정수의 튜플을 사용하는 인덱싱도 지원되며 올바른 형으로 하나의 요소를 돌려줍니다. 일차원 메모리 뷰는 정수 또는 하나의 정수를 갖는 튜플로 인덱싱 할 수 있습니다. 다차원 메모리 뷰는 정확히 *ndim* 개의 정수를 갖는 튜플로 인덱싱할 수 있습니다. 여기서 *ndim* 은 차원 수입니다. 영차원 메모리 뷰는 빈 튜플로 인덱싱할 수 있습니다.

다음은 바이트가 아닌 형식의 예입니다:

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

하부 객체가 쓰기 가능하면, 메모리 뷰는 일차원 슬라이스 대입을 지원합니다. 크기 변경은 허용되지 않습니다:

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different␣
↪structures
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

One-dimensional memoryviews of *hashable* (read-only) types with formats 'B', 'b' or 'c' are also hashable. The hash is defined as `hash(m) == hash(m.tobytes())`:

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

버전 3.3에서 변경: One-dimensional memoryviews can now be sliced. One-dimensional memoryviews with formats 'B', 'b' or 'c' are now *hashable*.

버전 3.4에서 변경: 이제 메모리 뷰는 자동으로 `collections.abc.Sequence` 로 등록됩니다

버전 3.5에서 변경: 이제 메모리 뷰는 정수의 튜플로 인덱싱될 수 있습니다.

버전 3.14에서 변경: memoryview is now a *generic type*.

`memoryview` 는 몇 가지 메서드를 가지고 있습니다:

__eq__(*exporter*)

메모리 뷰와 **PEP 3118** 제공자(exporter)는 다음과 같은 조건을 만족할 때 같다고 비교됩니다: 모양이 동등하고 피연산자의 각 형식 코드가 `struct` 문법을 사용하여 해석될 때 모든 해당 값이 같다.

현재 `tolist()` 가 지원하는 `struct` 형식 문자열의 부분 집합의 경우, `v.tolist() == w.tolist()` 면 `v` 와 `w` 는 같습니다:

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

형식 문자열이 *struct* 모듈에서 지원되지 않으면 객체는 항상 같지 않다고 비교됩니다 (형식 문자열과 버퍼 내용이 같더라도 그렇습니다):

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

Note that, as with floating-point numbers, `v is w` does *not* imply `v == w` for memoryview objects.

버전 3.3에서 변경: 이전 버전에서는 항목 형식과 논리 배열 구조를 무시하고 원시 메모리를 비교했습니다.

tobytes(*order='C'*)

버퍼의 데이터를 바이트열로 돌려줍니다. 이는 메모리 뷰에 *bytes* 생성자를 호출하는 것과 동등합니다.

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

불연속 배열의 경우 결과는 모든 요소를 바이트로 변환하여 평평한 리스트로 만든 것과 같습니다. *tobytes()* 는 *struct* 모듈 문법에 없는 것을 포함하여 모든 형식 문자열을 지원합니다.

Added in version 3.8: *order*는 { 'C' , 'F' , 'A' } 일 수 있습니다. *order*가 'C' 나 'F' 이면, 원래 배열의 데이터가 C 나 포트란 순서로 변환됩니다. 연속 뷰의 경우, 'A' 는 물리적 메모리의 정확한 사본을 반환합니다. 특히, 메모리 내 포트란 순서가 보존됩니다. 연속적이지 않은 뷰의 경우, 데이터는 먼저 C로 변환됩니다. *order=None*은 *order=' C'* 와 같습니다.

hex(**, bytes_per_sep=1*)

hex(*sep, bytes_per_sep=1*)

버퍼 내의 각 바이트를 두 개의 16진수로 표현한 문자열 객체를 돌려줍니다.

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

Added in version 3.5.

버전 3.8에서 변경: *bytes.hex()*와 비슷하게, 이제 *memoryview.hex()*는 16진수 출력의 바이트 사이에 구분 기호를 삽입하기 위해 선택적 *sep*과 *bytes_per_sep* 매개 변수를 지원합니다.

tolist()

버퍼 내의 데이터를 요소들의 리스트로 돌려줍니다.

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

버전 3.3에서 변경: *tolist()* 는 이제 *struct* 모듈 문법의 모든 단일 문자 네이티브 형식과 다차원 표현을 지원합니다.

toreadonly()

메모리 뷰 객체의 읽기 전용 버전을 반환합니다. 원래 메모리 뷰 객체는 변경되지 않습니다.

```
>>> m = memoryview(bytearray(b'abc'))
>>> mm = m.toreadonly()
>>> mm.tolist()
[97, 98, 99]
>>> mm[0] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> m[0] = 43
>>> mm.tolist()
[43, 98, 99]
```

Added in version 3.8.

release()

메모리 뷰 객체에 의해 노출된 하부 버퍼를 해제합니다. 많은 객체는 뷰가 그 객체에 연결될 때 특별한 조처를 합니다 (예를 들어, *bytearray* 는 일시적으로 크기 조절을 금지합니다); 따라서, release() 를 호출하면 가능한 한 빨리 이 제한 사항을 제거하고 붙잡힌 자원을 해제할 수 있습니다.

After this method has been called, any further operation on the view raises a *ValueError* (except *release()* itself which can be called multiple times):

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

`with` 문을 사용한 컨텍스트 관리 프로토콜은 비슷한 효과를 낼 수 있습니다:

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

Added in version 3.2.

cast(*format*, /)

cast(*format*, *shape*, /)

메모리 뷰를 새로운 형식이나 모양으로 캐스팅합니다. *shape* 의 기본값은 `[byte_length//new_itemsize]` 인데, 결과 뷰가 일차원이 된다는 의미입니다. 반환 값은 새로운 메모리 뷰이지만 버퍼 자체는 복사되지 않습니다. 지원되는 캐스팅은 1D -> C-연속 과 C-연속 -> 1D 입니다.

The destination format is restricted to a single element native format in `struct` syntax. One of the formats must be a byte format ('B' , 'b' or 'c'). The byte length of the result must be the same as the original length. Note that all byte lengths may depend on the operating system.

1D/long 을 1D/unsigned bytes 로 캐스트:

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

1D/unsigned bytes 를 1D/char 로 캐스트:

```
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  ...
TypeError: memoryview: invalid type for format 'B'
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

1D/bytes 를 3D/ints 로 캐스트 한 후 다시 1D/signed char 로 캐스트:

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

1D/unsigned long 을 2D/unsigned long 으로 캐스트:

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

Added in version 3.3.

버전 3.5에서 변경: 바이트 형식으로 변환할 때 소스 형식이 더는 제한되지 않습니다.

count(*value*, /)

Count the number of occurrences of *value*.

Added in version 3.14.

index(*value*, *start=0*, *stop=sys.maxsize*, /)

Return the index of the first occurrence of *value* (at or after index *start* and before index *stop*).

Raises a `ValueError` if *value* cannot be found.

Added in version 3.14.

몇 가지 읽기 전용 어트리뷰트도 사용할 수 있습니다:

obj

메모리 뷰의 하부 객체:

```
>>> b  = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

Added in version 3.3.

nbytes

`nbytes == product(shape) * itemsize == len(m.tobytes())`. 배열이 연속적일 때 차지하게 될 바이트 수입니다. 꼭 `len(m)` 과 같을 필요는 없습니다:

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

다차원 배열:

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96
```

Added in version 3.3.

readonly

메모리가 읽기 전용인지 여부를 나타내는 논리값.

format

뷰의 각 요소에 대한 형식(*struct* 모듈 스타일)을 포함하는 문자열입니다. 메모리 뷰는 제공자로부터 임의의 형식 문자열로 만들어질 수 있지만, 일부 메서드(예, *tolist()*)는 원시 네이티브 단일 요소 형식으로 제한됩니다.

버전 3.3에서 변경: `'B'` 형식은 이제 struct 모듈 문법에 따라 처리됩니다. 이것은 `memoryview(b'abc')[0] == b'abc'[0] == 97` 이 됨을 의미합니다.

itemsize

메모리 뷰 각 요소의 크기 (바이트):

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim

메모리가 나타내는 다차원 배열의 차원 수를 나타내는 정수.

shape

N-차원 배열로서의 메모리의 모양을 가리키는, 길이 *ndim* 인 정수의 튜플입니다.

버전 3.3에서 변경: ndim = 0 일 때 `None` 대신 빈 튜플을 제공합니다.

strides

배열의 각 차원에 대해 각 요소를 참조하는데 필요한 바이트 수를 제공하는, 길이 *ndim* 인 정수의 튜플입니다.

버전 3.3에서 변경: ndim = 0 일 때 `None` 대신 빈 튜플을 제공합니다.

suboffsets

PIL 스타일 배열에 내부적으로 사용됩니다. 값은 정보 제공용입니다.

c_contiguous

메모리가 C-연속 인지를 나타내는 논리값.

Added in version 3.3.

f_contiguous

메모리가 포트란 연속 인지를 나타내는 논리값.

Added in version 3.3.

contiguous

메모리가 연속 인지를 나타내는 논리값.

Added in version 3.3.

4.11 집합 형 — set, frozenset

집합 (*set*) 객체는 서로 다른 해시 가능 객체의 순서 없는 컬렉션입니다. 일반적인 용도는 멤버십 검사, 시퀀스에서 중복 제거와 교집합, 합집합, 차집합, 대칭 차집합과 같은 수학 연산을 계산하는 것입니다. (다른 컨테이너들은 내장 *dict*, *list*, *tuple* 클래스 및 *collections* 모듈을 참조하십시오.)

다른 컬렉션과 마찬가지로, 집합은 `x in set`, `len(set)`, `for x in set` 을 지원합니다. 순서가 없는 컬렉션이므로, 집합은 원소의 위치나 삽입 순서를 기록하지 않습니다. 따라서 집합은 인덱싱, 슬라이싱 또는 기타 시퀀스와 유사한 동작을 지원하지 않습니다.

현재 두 가지 내장형이 있습니다, *set*과 *frozenset*. *set* 형은 가변입니다 — 내용을 `add()` 나 `remove()` 와 같은 메서드를 사용하여 변경할 수 있습니다. 가변이기 때문에, 해시값이 없으며 딕셔너리 키 또는 다른 집합의 원소로 사용할 수 없습니다. *frozenset* 형은 불변이고 해시 가능 합니다 — 만들어진 후에는 내용을 바꿀 수 없습니다; 따라서 딕셔너리 키 또는 다른 집합의 원소로 사용할 수 있습니다.

비어 있지 않은 set은 (frozenset 은 아닙니다) *set* 생성자뿐만 아니라 중괄호 안에 쉼표로 구분된 원소 목록을 넣어서 만들 수 있습니다, 예를 들어: `{'jack', 'sjoerd'}`.

두 클래스의 생성자는 같게 작동합니다:

class set(*iterable*=(), /)

class frozenset(*iterable*=(), /)

iterable 에서 요소를 취하는 새 set 또는 frozenset 객체를 돌려줍니다. 집합의 원소는 반드시 해시 가능 해야 합니다. 집합의 집합을 표현하려면, 포함되는 집합은 반드시 *frozenset* 객체여야 합니다. *iterable* 을 지정하지 않으면 새 빈 집합을 돌려줍니다.

집합은 여러 가지 방법으로 만들 수 있습니다:

- 중괄호 안에 쉼표로 구분된 요소 나열하기: `{'jack', 'sjoerd'}`
- 집합 컴프리헨션 사용하기: `{c for c in 'abracadabra' if c not in 'abc'}`
- 형 생성자 사용하기: `set()`, `set('foobar')`, `set(['a', 'b', 'foo'])`

*set*과 *frozenset* 의 인스턴스는 다음과 같은 연산을 제공합니다:

len(s)

집합 *s* 의 원소 수(*s* 의 크기)를 돌려줍니다.

x in s

s 에 대해 *x* 의 멤버십을 검사합니다.

x not in s

s 에 대해 *x* 의 비 멤버십을 검사합니다.

isdisjoint(*other*, /)

집합이 *other* 와 공통 원소를 갖지 않는 경우 `True` 을 돌려줍니다. 집합은 교집합이 공집합일 때, 그리고 그때만 서로소(disjoint)라고 합니다.

issubset(*other*, /)

set <= other

집합의 모든 원소가 *other* 에 포함되는지 검사합니다.

set < other

집합이 *other* 의 진부분집합인지 검사합니다, 즉, `set <= other and set != other`.

issuperset(*other*, /)

set >= other

other 의 모든 원소가 집합에 포함되는지 검사합니다.

set > other

집합이 *other* 의 진상위집합인지 검사합니다, 즉, `set >= other and set != other`.

union(**others*)

set | other | ...

집합과 모든 others에 있는 원소들로 구성된 새 집합을 돌려줍니다.

intersection(**others*)

set & other & ...

집합과 모든 others의 공통 원소들로 구성된 새 집합을 돌려줍니다.

difference(**others*)

set - other - ...

집합에는 포함되었으나 others에는 포함되지 않은 원소들로 구성된 새 집합을 돌려줍니다.

symmetric_difference(*other*, /)

set ^ other

집합이나 other에 포함되어 있으나 둘 모두에 포함되지는 않은 원소들로 구성된 새 집합을 돌려줍니다.

copy()

집합의 얕은 복사본을 돌려줍니다.

Note, the non-operator versions of *union()*, *intersection()*, *difference()*, *symmetric_difference()*, *issubset()*, and *issuperset()* methods will accept any iterable as an argument. In contrast, their operator based counterparts require their arguments to be sets. This precludes error-prone constructions like `set('abc') & 'cbs'` in favor of the more readable `set('abc').intersection('cbs')`.

*set*과 *frozenset* 모두 집합 간의 비교를 지원합니다. 두 집합은 각 집합의 모든 원소가 다른 집합에 포함되어있는 경우에만 같습니다 (서로 다른 집합의 부분집합입니다). 집합이 다른 집합의 진부분집합(부분집합이지만 같지는 않은 경우)일 때만 첫 번째 집합이 두 번째 집합보다 작습니다. 집합이 다른 집합의 진상위집합(상위집합이지만 같지는 않은 경우)일 때만 첫 번째 집합이 두 번째 집합보다 큽니다.

set 의 인스턴스는 그 원소를 기반으로 *frozenset* 의 인스턴스와 비교됩니다. 예를 들어, `set('abc') == frozenset('abc')` 는 `True` 를 돌려주고 `set('abc') in set([frozenset('abc')])` 도 마찬가지입니다.

부분 집합 및 동등 비교는 전 순서(total ordering) 함수로 일반화되지 않습니다. 예를 들어, 비어 있지 않은 두 개의 서로소인 집합은 같지 않고 서로의 부분 집합이 아닙니다, 그래서 다음은 모두 `False` 를 돌려줍니다: `a<b`, `a==b`, `a>b`.

집합은 부분 순서(부분 집합 관계)만 정의하기 때문에, 집합의 리스트에 대한 `list.sort()` 메서드의 결과는 정의되지 않습니다.

딕셔너리 키처럼, 집합의 원소는 반드시 해시 가능 해야 합니다.

`set` 인스턴스와 `frozenset` 을 혼합 한 이항 연산은 첫 번째 피연산자의 형을 돌려줍니다. 예를 들어: `frozenset('ab') | set('bc')` 는 `frozenset` 의 인스턴스를 돌려줍니다.

다음 표는 `frozenset` 의 불변 인스턴스에는 적용되지 않고 `set` 에서만 사용할 수 있는 연산들을 나열합니다:

update(**others*)

set |= other | ...

집합을 갱신해서, 모든 others의 원소들을 더합니다.

intersection_update(**others*)

set &= other & ...

집합을 갱신해서, 그 집합과 others에 공통으로 포함된 원소들만 남깁니다.

difference_update(**others*)

set -= other | ...

집합을 갱신해서, others에 있는 원소들을 제거합니다.

symmetric_difference_update(*other*, /)

set ^= other

집합을 갱신해서, 두 집합의 어느 한 곳에만 포함된 원소들만 남깁니다.

add(*elem*, /)

원소 *elem* 을 집합에 추가합니다.

remove(*elem*, /)

원소 *elem* 을 집합에서 제거합니다. *elem* 가 집합에 포함되어 있지 않으면 `KeyError` 를 일으킵니다.

discard(*elem*, /)

원소 *elem* 이 집합에 포함되어 있으면 제거합니다.

pop()

집합으로부터 임의의 원소를 제거해 돌려줍니다. 집합이 비어있는 경우 `KeyError` 를 일으킵니다.

clear()

집합의 모든 원소를 제거합니다.

참고로, `update()`, `intersection_update()`, `difference_update()`, `symmetric_difference_update()` 메서드의 비 연산자 버전은 임의의 이터러블을 인자로 받아들입니다.

Note, the *elem* argument to the `__contains__()`, `remove()`, and `discard()` methods may be a set. To support searching for an equivalent frozenset, a temporary one is created from *elem*.

4.12 매핑 형 — dict

매핑 객체는 해시 가능 값을 임의의 객체에 대응합니다. 매핑은 가변 객체입니다. 현재 오직 하나의 표준 매핑 형이 있습니다, 딕셔너리 (*dictionary*). (다른 컨테이너들은 내장 `list`, `set`, `tuple` 클래스 및 `collections` 모듈을 참조하십시오.)

A dictionary' s keys are *almost* arbitrary values. Values that are not *hashable*, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may not be used as keys. Values that compare equal (such as `1`, `1.0`, and `True`) can be used interchangeably to index the same dictionary entry.

class dict(***kwargs*)

class dict(*mapping*, /, ***kwargs*)

class dict(*iterable*, /, ***kwargs*)

선택적 위치 인자와 (비어있을 수 있는) 키워드 인자들의 집합으로부터 초기화된 새 딕셔너리를 돌려줍니다.

딕셔너리는 여러 가지 방법으로 만들 수 있습니다:

- 중괄호 안에 쉼표로 구분된 `key: value` 쌍을 나열하기: `{'jack': 4098, 'sjoerd': 4127}` 또는 `{4098: 'jack', 4127: 'sjoerd'}`
- 딕셔너리 컴프리헨션 사용하기: `{}`, `{x: x ** 2 for x in range(10)}`
- 형 생성자 사용하기: `dict()`, `dict([('foo', 100), ('bar', 200)])`, `dict(foo=100, bar=200)`

If no positional argument is given, an empty dictionary is created. If a positional argument is given and it defines a `keys()` method, a dictionary is created by calling `__getitem__()` on the argument with each returned key from the method. Otherwise, the positional argument must be an *iterable* object. Each item in the iterable must itself be an iterable with exactly two elements. The first element of each item becomes a key in the new dictionary, and the second element the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

키워드 인자가 제공되면, 키워드 인자와 해당 값이 위치 인자로부터 만들어진 딕셔너리에 추가됩니다. 추가되는 키가 이미 존재하면, 키워드 인자에서 온 값이 위치 인자에게서 온 값을 대체합니다.

첫 번째 예제에서와같이 키워드 인자는 유효한 파이썬 식별자인 키에 대해서만 작동합니다. 그 외의 경우는 모든 유효한 키를 사용할 수 있습니다.

Dictionaries compare equal if and only if they have the same `(key, value)` pairs (regardless of ordering). Order comparisons ('<' , '<=' , '>=' , '>') raise `TypeError`. To illustrate dictionary creation and equality, the following examples all return a dictionary equal to `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> f = dict({'one': 1, 'three': 3}, two=2)
>>> a == b == c == d == e == f
True
```

첫 번째 예제에서와같이 키워드 인자는 유효한 파이썬 식별자인 키에 대해서만 작동합니다. 그 외의 경우는 모든 유효한 키를 사용할 수 있습니다.

딕셔너리는 삽입 순서를 유지합니다. 키를 갱신해도 순서에는 영향을 미치지 않습니다. 삭제 후에 추가된 키는 끝에 삽입됩니다.:

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

버전 3.7에서 변경: 딕셔너리 순서는 삽입 순서임이 보장됩니다. 이 동작은 3.6부터 CPython의 구현 세부 사항입니다.

이것들은 딕셔너리가 지원하는 연산들입니다 (그러므로, 사용자 정의 매핑 형도 지원해야 합니다):

list(d)

딕셔너리 *d* 에 사용된 모든 키의 리스트를 돌려줍니다.

len(d)

딕셔너리 *d* 에 있는 항목의 수를 돌려줍니다.

d[key]

키 *key* 인 *d* 의 항목을 돌려줍니다. *key* 가 매핑에 없는 경우 `KeyError` 를 일으킵니다.

If a subclass of dict defines a method `__missing__()` and *key* is not present, the `d[key]` operation calls that method with the key *key* as argument. The `d[key]` operation then returns or raises whatever is returned or raised by the `__missing__(key)` call. No other operations or methods invoke `__missing__()`. If `__missing__()` is not defined, `KeyError` is raised. `__missing__()` must be a method; it cannot be an instance variable:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
...
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

The example above shows part of the implementation of `collections.Counter`. A different `__missing__()` method is used by `collections.defaultdict`.

d[key] = value

`d[key]` 를 *value* 로 설정합니다.

del d[key]

d 에서 `d[key]` 를 제거합니다. *key* 가 매핑에 없는 경우 `KeyError` 를 일으킵니다.

key in d

d 에 키 *key* 가 있으면 `True` 를, 그렇지 않으면 `False` 를 돌려줍니다.

key not in d

`not key in d` 와 동등합니다.

iter(d)

딕셔너리의 키에 대한 이터레이터를 돌려줍니다. 이것은 `iter(d.keys())` 의 단축입니다.

clear()

딕셔너리에서 모든 항목을 제거합니다.

copy()

딕셔너리의 얕은 복사본을 돌려줍니다.

classmethod fromkeys(*iterable*, *value=None*, /)

iterable 이 제공하는 값들을 키로 사용하고 모든 값을 *value* 로 설정한 새 딕셔러리를 돌려줍니다.

`fromkeys()` 는 새로운 딕셔너리를 돌려주는 클래스 메서드입니다. *value* 의 기본값은 `None` 입니다. 모든 값이 단일 인스턴스를 참조하므로, *value* 가 빈 목록과 같은 가변 객체가 되는 것은 일반적으로 의미가 없습니다. 별개의 값을 얻으려면, 대신 딕셔너리 컴프리헨션을 사용하십시오.

get(*key*, *default=None*, /)

key 가 딕셔너리에 있는 경우 *key* 에 대응하는 값을 돌려주고, 그렇지 않으면 *default* 를 돌려줍니다. *default* 가 주어지지 않으면 기본값 `None` 이 사용됩니다. 그래서 이 메서드는 절대로 `KeyError` 를 일으키지 않습니다.

items()

딕셔너리 항목들(`(key, value)` 쌍들)의 새 뷰를 돌려줍니다. 뷰 객체의 설명서 을 참조하세요.

keys()

딕셔너리 키들의 새 뷰를 돌려줍니다. 뷰 객체의 설명서 을 참조하세요.

pop(*key*, /)

pop(*key*, *default*, /)

key 가 딕셔너리에 있으면 제거하고 그 값을 돌려줍니다. 그렇지 않으면 *default* 를 돌려줍니다. *default* 가 주어지지 않고 *key* 가 딕셔너리에 없으면 `KeyError` 를 일으킵니다.

popitem()

딕셔너리에서 `(key, value)` 쌍을 제거하고 돌려줍니다. 쌍은 LIFO (last-in, first-out) 순서로 반환됩니다.

`popitem()` 은 집합 알고리즘에서 종종 사용되듯이 딕셔너리를 파괴적으로 이터레이션 하는데 유용합니다. 딕셔너리가 비어 있으면 `popitem()` 호출은 `KeyError` 를 일으킵니다.

버전 3.7에서 변경: 이제 LIFO 순서가 보장됩니다. 이전 버전에서는, `popitem()` 가 임의의 키/값 쌍을 반환합니다.

reversed(d)

딕셔너리의 키에 대한 역순 이터레이터를 돌려줍니다. 이것은 `reversed(d.keys())` 의 단축입니다.

Added in version 3.8.

setdefault(*key*, *default=None*, /)

key 가 딕셔너리에 있으면 해당 값을 돌려줍니다. 그렇지 않으면, *default* 값을 갖는 *key* 를 삽입한 후 *default* 를 돌려줍니다. *default* 의 기본값은 `None` 입니다.

update(**kwargs)

update(*mapping*, /, **kwargs)

update(*iterable*, /, **kwargs)

Update the dictionary with the key/value pairs from *mapping* or *iterable* and *kwargs*, overwriting existing keys. Return `None`.

`update()` accepts either another object with a `keys()` method (in which case `__getitem__()` is called with every key returned from the method) or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: `d.update(red=1, blue=2)`.

values()

딕셔너리 값들의 새 뷰를 돌려줍니다. 뷰 객체의 설명서 을 참조하세요.

한 `dict.values()` 뷰와 다른 `dict.values()` 뷰 간의 동등 비교는 항상 `False`를 반환합니다. 이것은 `dict.values()` 를 자신과 비교할 때도 적용됩니다:

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

d | other

d 와 *other* 의 병합된 키와 값으로 새 딕셔너리를 만듭니다. 둘 다 딕셔너리이어야 합니다. *d* 와 *other* 가 키를 공유하면 *other* 의 값이 우선합니다.

Added in version 3.9.

d |= other

*other*의 키와 값으로 딕셔너리 *d*를 갱신합니다. *other*는 매핑이나 키/값 쌍의 이터러블일 수 있습니다. *d*와 *other*가 키를 공유하면 *other*의 값이 우선합니다.

Added in version 3.9.

딕셔너리와 딕셔너리 뷰는 뒤집을 수 있습니다.

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(reversed(d))
['four', 'three', 'two', 'one']
>>> list(reversed(d.values()))
[4, 3, 2, 1]
>>> list(reversed(d.items()))
[('four', 4), ('three', 3), ('two', 2), ('one', 1)]
```

버전 3.8에서 변경: 딕셔너리는 이제 뒤집을 수 있습니다.

더 보기

types.MappingProxyType 를 *dict* 의 읽기 전용 뷰를 만드는 데 사용할 수 있습니다.

4.12.1 딕셔너리 뷰 객체

dict.keys(), *dict.values()*, *dict.items()* 가 돌려주는 객체는 뷰 객체 입니다. 딕셔너리의 항목들에 대한 동적 뷰를 제공합니다. 즉, 딕셔너리가 변경되면 뷰는 이러한 변경 사항을 반영합니다.

딕셔너리 뷰는 이터레이션을 통해 각각의 데이터를 산출할 수 있고, 멤버십 검사를 지원합니다:

len(dictview)

딕셔너리에 있는 항목 수를 돌려줍니다.

iter(dictview)

딕셔너리에서 키, 값, 항목(`(key, value)` 튜플로 표현됩니다)에 대한 이터레이터를 돌려줍니다.

키와 값은 삽입 순서로 이터레이션 됩니다. 이 때문에 *zip()*을 사용해서 `(value, key)` 쌍을 만들 수 있습니다: `pairs = zip(d.values(), d.keys())`. 같은 리스트를 만드는 다른 방법은 `pairs = [(v, k) for (k, v) in d.items()]` 입니다.

딕셔너리에 항목을 추가하거나 삭제하는 동안 뷰를 이터레이션 하면 *RuntimeError* 를 일으키거나 모든 항목을 이터레이션 하지 못할 수 있습니다.

버전 3.7에서 변경: 딕셔너리의 순서가 삽입 순서임이 보장됩니다.

x in dictview

*x*가 하부 딕셔너리의 키, 갑, 항목에 있는 경우 `True` 를 돌려줍니다(마지막의 경우 *x*는 `(key, value)` 튜플이어야 합니다).

reversed(dictview)

딕셔너리의 키, 값 또는 항목에 대한 역방향 이터레이터를 반환합니다. 뷰는 삽입의 역순으로 이터레이트됩니다.

버전 3.8에서 변경: 딕셔너리 뷰는 이제 역 탐색할 수 있습니다.

dictview.mapping

Return a *types.MappingProxyType* that wraps the original dictionary to which the view refers.

Added in version 3.10.

Keys views are set-like since their entries are unique and *hashable*. Items views also have set-like operations since the (key, value) pairs are unique and the keys are hashable. If all values in an items view are hashable as well, then the items view can interoperate with other sets. (Values views are not treated as set-like since the entries are generally not unique.) For set-like views, all of the operations defined for the abstract base class `collections.abc.Set` are available (for example, ==, <, or ^). While using set operators, set-like views accept any iterable as the other operand, unlike sets which only accept sets as the input.

딕셔너리 뷰 사용의 예:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
...
>>> print(n)
504

>>> # keys and values are iterated over in the same order (insertion order)
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'} == {'juice', 'sausage', 'bacon', 'spam'}
True
>>> keys | ['juice', 'juice', 'juice'] == {'bacon', 'spam', 'juice'}
True

>>> # get back a read-only proxy for the original dictionary
>>> values.mapping
mappingproxy({'bacon': 1, 'spam': 500})
>>> values.mapping['spam']
500
```

4.13 컨텍스트 관리자 형

파이썬의 `with` 문은 컨텍스트 관리자가 정의한 실행 시간 컨텍스트 개념을 지원합니다. 이는 한 쌍의 메서드를 사용해서 구현되는데, 사용자 정의 클래스가 문장 바디가 실행되기 전에 진입하고, 문장이 끝날 때 탈출하는 실행 시간 컨텍스트를 정의할 수 있게 합니다:

`contextmanager.__enter__()`

실행시간 컨텍스트에 진입하고 이 객체 자신이나 실행 시간 컨텍스트와 관련된 다른 객체를 돌려줍니다. 이 메서드가 돌려주는 값은, 이 컨텍스트 관리자를 사용하는 `with` 문의 `as` 절의 식별자에 연결됩니다.

자신을 돌려주는 컨텍스트 관리자의 예는 파일 객체 입니다. 파일 객체는 __enter__() 에서 자기 자신을 돌려주는데 `with` 문의 컨텍스트 표현식으로 `open()` 을 사용할 수 있도록 하기 위함입니다.

관련 객체를 돌려주는 컨텍스트 관리자의 예는 `decimal.localcontext()` 가 돌려주는 것입니다. 이 관리자들은 활성 십진 소수 컨텍스트를 원래 십진 소수 컨텍스트의 복사본으로 설정한 다음 복사본을 돌려줍니다. 이것은 `with` 문 바깥의 코드에 영향을 주지 않으면서 `with` 문 바디에 있는 현재 십진 소수 컨텍스트를 변경할 수 있게 합니다.

`contextmanager.__exit__`(*exc_type*, *exc_val*, *exc_tb*)

실행 시간 컨텍스트를 탈출하고 발생한 예외를 막아야 하는지를 가리키는 논리 플래그를 돌려줍니다. `with` 문의 바디를 실행하는 동안 예외가 발생하면, 인자에 예외 형, 값 및 추적 정보가 포함됩니다. 그렇지 않으면, 세 가지 인자 모두 `None` 입니다.

이 메서드에서 참 값을 돌려주면 `with` 문이 예외를 막고 `with` 문 바로 뒤에 오는 문장에서 계속 실행됩니다. 그 이외의 경우, 이 메서드의 실행이 완료된 후에 예외는 계속 퍼집니다. 이 메서드의 실행 중에 발생하는 예외는 `with` 문의 바디에서 발생한 모든 예외를 대체합니다.

The exception passed in should never be reraised explicitly - instead, this method should return a false value to indicate that the method completed successfully and does not want to suppress the raised exception. This allows context management code to easily detect whether or not an `__exit__()` method has actually failed.

파이썬은 쉬운 스레드 동기화, 파일이나 다른 객체의 신속한 닫기, 그리고 활성 십진 소수 산술 컨텍스트의 보다 간단한 조작을 지원하기 위해 몇 가지 컨텍스트 관리자를 정의합니다. 컨텍스트 관리 프로토콜의 구현을 넘어 구체적인 형은 특별히 취급되지 않습니다. 몇 가지 예제는 `contextlib` 모듈을 보십시오.

Python' s *generator*s and the `contextlib.contextmanager` decorator provide a convenient way to implement these protocols. If a generator function is decorated with the `contextlib.contextmanager` decorator, it will return a context manager implementing the necessary `__enter__()` and `__exit__()` methods, rather than the iterator produced by an undecorated generator function.

파이썬/C API의 파이썬 객체에 대한 형 구조체에는 이러한 메서드들을 위해 준비된 슬롯이 없다는 점에 유의하십시오. 이러한 메서드를 정의하고자 하는 확장형은 일반적인 파이썬 액세스가 가능한 메서드로 제공해야 합니다. 실행 시간 컨텍스트를 설정하는 오버헤드와 비교할 때 한 번의 클래스 딕셔너리 조회의 오버헤드는 무시할 수 있습니다.

4.14 Type Annotation Types — Generic Alias, Union

The core built-in types for *type annotations* are *Generic Alias* and *Union*.

4.14.1 제네릭 에일리어스 형

`GenericAlias` objects are generally created by subscripting a class. They are most often used with container classes, such as `list` or `dict`. For example, `list[int]` is a `GenericAlias` object created by subscripting the `list` class with the argument `int`. `GenericAlias` objects are intended primarily for use with *type annotations*.

> **참고**
>
> It is generally only possible to subscript a class if the class implements the special method `__class_getitem__()`.

A `GenericAlias` object acts as a proxy for a *generic type*, implementing *parameterized generics*.

For a container class, the argument(s) supplied to a subscription of the class may indicate the type(s) of the elements an object contains. For example, `set[bytes]` can be used in type annotations to signify a `set` in which all the elements are of type `bytes`.

For a class which defines `__class_getitem__()` but is not a container, the argument(s) supplied to a subscription of the class will often indicate the return type(s) of one or more methods defined on an object. For example, `regular expressions` can be used on both the `str` data type and the `bytes` data type:

- If `x = re.search('foo', 'foo')`, `x` will be a *re.Match* object where the return values of `x.group(0)` and `x[0]` will both be of type *str*. We can represent this kind of object in type annotations with the `GenericAlias` `re.Match[str]`.
- If `y = re.search(b'bar', b'bar')`, (note the `b` for *bytes*), `y` will also be an instance of `re.Match`, but the return values of `y.group(0)` and `y[0]` will both be of type *bytes*. In type annotations, we would represent this variety of *re.Match* objects with `re.Match[bytes]`.

`GenericAlias` objects are instances of the class *types.GenericAlias*, which can also be used to create `GenericAlias` objects directly.

T[X, Y, ...]

Creates a `GenericAlias` representing a type `T` parameterized by types *X*, *Y*, and more depending on the `T` used. For example, a function expecting a *list* containing *float* elements:

```
def average(values: list[float]) -> float:
    return sum(values) / len(values)
```

키 형과 값 형을 나타내는 두 개의 형 매개 변수를 기대하는 제네릭 형인 *dict*를 사용하는 매핑 객체의 또 다른 예. 이 예에서, 함수는 *str* 형의 키와 *int* 형의 값을 갖는 `dict`를 기대합니다:

```
def send_post_request(url: str, body: dict[str, int]) -> None:
    ...
```

내장 함수 *isinstance()*와 *issubclass()*는 두 번째 인자로 `GenericAlias` 형을 받아들이지 않습니다:

```
>>> isinstance([1, 2], list[str])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: isinstance() argument 2 cannot be a parameterized generic
```

The Python runtime does not enforce *type annotations*. This extends to generic types and their type parameters. When creating a container object from a `GenericAlias`, the elements in the container are not checked against their type. For example, the following code is discouraged, but will run without errors:

```
>>> t = list[str]
>>> t([1, 2, 3])
[1, 2, 3]
```

또한, 매개 변수화된 제네릭은 객체 생성 중에 형 매개 변수를 지웁니다:

```
>>> t = list[str]
>>> type(t)
<class 'types.GenericAlias'>

>>> l = t()
>>> type(l)
<class 'list'>
```

제네릭에서 *repr()*이나 *str()*을 호출하면 매개 변수화된 형이 표시됩니다:

```
>>> repr(list[int])
'list[int]'

>>> str(list[int])
'list[int]'
```

The `__getitem__()` method of generic containers will raise an exception to disallow mistakes like `dict[str][str]`:

```
>>> dict[str][str]
Traceback (most recent call last):
  ...
TypeError: dict[str] is not a generic class
```

However, such expressions are valid when *type variables* are used. The index must have as many elements as there are type variable items in the `GenericAlias` object' s `__args__`.

```
>>> from typing import TypeVar
>>> Y = TypeVar('Y')
>>> dict[str, Y][int]
dict[str, int]
```

Standard Generic Classes

The following standard library classes support parameterized generics. This list is non-exhaustive.

- *tuple*
- *list*
- *dict*
- *set*
- *frozenset*
- *type*
- *asyncio.Future*
- *asyncio.Task*
- *collections.deque*
- *collections.defaultdict*
- *collections.OrderedDict*
- *collections.Counter*
- *collections.ChainMap*
- *collections.abc.Awaitable*
- *collections.abc.Coroutine*
- *collections.abc.AsyncIterable*
- *collections.abc.AsyncIterator*
- *collections.abc.AsyncGenerator*
- *collections.abc.Iterable*
- *collections.abc.Iterator*
- *collections.abc.Generator*
- *collections.abc.Reversible*
- *collections.abc.Container*
- *collections.abc.Collection*
- *collections.abc.Callable*
- *collections.abc.Set*
- *collections.abc.MutableSet*

- *collections.abc.Mapping*
- *collections.abc.MutableMapping*
- *collections.abc.Sequence*
- *collections.abc.MutableSequence*
- *collections.abc.ByteString*
- *collections.abc.MappingView*
- *collections.abc.KeysView*
- *collections.abc.ItemsView*
- *collections.abc.ValuesView*
- *contextlib.AbstractContextManager*
- *contextlib.AbstractAsyncContextManager*
- *dataclasses.Field*
- *functools.cached_property*
- *functools.partialmethod*
- *os.PathLike*
- *queue.LifoQueue*
- *queue.Queue*
- *queue.PriorityQueue*
- *queue.SimpleQueue*
- *re.Pattern*
- *re.Match*
- *shelve.BsdDbShelf*
- *shelve.DbfilenameShelf*
- *shelve.Shelf*
- *types.MappingProxyType*
- *weakref.WeakKeyDictionary*
- *weakref.WeakMethod*
- *weakref.WeakSet*
- *weakref.WeakValueDictionary*

Special Attributes of `GenericAlias` objects

모든 매개 변수화된 제네릭은 특수 읽기 전용 어트리뷰트를 구현합니다.

`genericalias.__origin__`

이 어트리뷰트는 매개 변수화되지 않은 제네릭 클래스를 가리킵니다:

```
>>> list[int].__origin__
<class 'list'>
```

`genericalias.__args__`

This attribute is a *tuple* (possibly of length 1) of generic types passed to the original `__class_getitem__()` of the generic class:

```
>>> dict[str, list[int]].__args__
(<class 'str'>, list[int])
```

genericalias.**__parameters__**

이 어트리뷰트는 `__args__`에서 발견된 고유한 형 변수의 게으르게(lazily) 계산된 튜플(비어있을 수 있습니다)입니다:

```
>>> from typing import TypeVar

>>> T = TypeVar('T')
>>> list[T].__parameters__
(~T,)
```

> **참고**
>
> A `GenericAlias` object with *typing.ParamSpec* parameters may not have correct `__parameters__` after substitution because *typing.ParamSpec* is intended primarily for static type checking.

genericalias.**__unpacked__**

A boolean that is true if the alias has been unpacked using the `*` operator (see *TypeVarTuple*).

Added in version 3.11.

> **더 보기**
>
> **PEP 484 - Type Hints**
> Introducing Python' s framework for type annotations.
>
> **PEP 585 - Type Hinting Generics In Standard Collections**
> Introducing the ability to natively parameterize standard-library classes, provided they implement the special class method `__class_getitem__()`.
>
> 제네릭, ***user-defined generics*** **and** ***typing.Generic***
> Documentation on how to implement generic classes that can be parameterized at runtime and understood by static type-checkers.

Added in version 3.9.

4.14.2 Union Type

A union object holds the value of the `|` (bitwise or) operation on multiple *type objects*. These types are intended primarily for *type annotations*. The union type expression enables cleaner type hinting syntax compared to subscripting *typing.Union*.

X | Y | ...

Defines a union object which holds types *X*, *Y*, and so forth. `X | Y` means either X or Y. It is equivalent to `typing.Union[X, Y]`. For example, the following function expects an argument of type *int* or *float*:

```
def square(number: int | float) -> int | float:
    return number ** 2
```

> **참고**
>
> The `|` operand cannot be used at runtime to define unions where one or more members is a forward reference. For example, `int | "Foo"`, where `"Foo"` is a reference to a class not yet defined, will fail at

runtime. For unions which include forward references, present the whole expression as a string, e.g. `"int | Foo"`.

union_object == other

Union objects can be tested for equality with other union objects. Details:

- Unions of unions are flattened:

```
(int | str) | float == int | str | float
```

- Redundant types are removed:

```
int | str | int == int | str
```

- When comparing unions, the order is ignored:

```
int | str == str | int
```

- It creates instances of *typing.Union*:

```
int | str == typing.Union[int, str]
type(int | str) is typing.Union
```

- Optional types can be spelled as a union with `None`:

```
str | None == typing.Optional[str]
```

isinstance(obj, union_object)

issubclass(obj, union_object)

Calls to *isinstance()* and *issubclass()* are also supported with a union object:

```
>>> isinstance("", int | str)
True
```

However, *parameterized generics* in union objects cannot be checked:

```
>>> isinstance(1, int | list[int])  # short-circuit evaluation
True
>>> isinstance([1], int | list[int])
Traceback (most recent call last):
  ...
TypeError: isinstance() argument 2 cannot be a parameterized generic
```

The user-exposed type for the union object can be accessed from *typing.Union* and used for *isinstance()* checks:

```
>>> import typing
>>> isinstance(int | str, typing.Union)
True
>>> typing.Union()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create 'typing.Union' instances
```

참고

The `__or__()` method for type objects was added to support the syntax `X | Y`. If a metaclass implements `__or__()`, the Union may override it:

```
>>> class M(type):
...     def __or__(self, other):
...         return "Hello"
...
>>> class C(metaclass=M):
...     pass
...
>>> C | int
'Hello'
>>> int | C
int | C
```

더 보기

PEP 604 – PEP proposing the `X | Y` syntax and the Union type.

Added in version 3.10.

버전 3.14에서 변경: Union objects are now instances of `typing.Union`. Previously, they were instances of `types.UnionType`, which remains an alias for `typing.Union`.

4.15 기타 내장형

인터프리터는 여러 가지 다른 객체를 지원합니다. 이것들 대부분은 한두 가지 연산만 지원합니다.

4.15.1 모듈

모듈에 대한 유일한 특별한 연산은 어트리뷰트 액세스입니다: `m.name`. 여기서 *m* 은 모듈이고 *name* 은 *m* 의 심볼 테이블에 정의된 이름에 액세스합니다. 모듈 어트리뷰트는 대입할 수 있습니다. (`import` 문은 엄밀히 말하면 모듈 객체에 대한 연산이 아닙니다; `import foo` 는 *foo* 라는 이름의 모듈 객체가 존재할 것을 요구하지 않고, 어딘가에 있는 *foo* 라는 이름의 (외부) 정의 를 요구합니다.

모든 모듈의 특수 어트리뷰트는 `__dict__` 입니다. 이것은 모듈의 심볼 테이블을 저장하는 딕셔너리입니다. 이 딕셔너리를 수정하면 모듈의 심볼 테이블이 실제로 변경되지만, `__dict__` 어트리뷰트에 대한 직접 대입은 불가능합니다 (`m.__dict__['a'] = 1` 라고 쓸 수 있고, `m.a` 가 `1` 이 되지만, `m.__dict__ = {}` 라고 쓸 수는 없습니다). `__dict__` 의 직접적인 수정은 추천하지 않습니다.

인터프리터에 내장된 모듈은 다음과 같이 쓰입니다: `<module 'sys' (built-in)>`. 파일에서 로드되면, `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>` 처럼 쓰입니다.

4.15.2 클래스와 클래스 인스턴스

여기에 대해서는 objects와 class를 참조하세요.

4.15.3 함수

함수 객체는 함수 정의로 만들어집니다. 함수 객체에 대한 유일한 연산은 호출하는 것입니다: `func(argument-list)`.

함수 객체에는 내장 함수와 사용자 정의 함수라는 두 가지 종류가 있습니다. 두 함수 모두 같은 연산(함수 호출)을 지원하지만, 구현이 다르므로 서로 다른 객체 형입니다.

자세한 정보는 function을 보십시오.

4.15.4 메서드

Methods are functions that are called using the attribute notation. There are two flavors: built-in methods (such as *append()* on lists) and class instance method. Built-in methods are described with the types that support them.

If you access a method (a function defined in a class namespace) through an instance, you get a special object: a *bound method* (also called instance method) object. When called, it will add the `self` argument to the argument list. Bound methods have two special read-only attributes: `m.__self__` is the object on which the method operates, and `m.__func__` is the function implementing the method. Calling `m(arg-1, arg-2, ..., arg-n)` is completely equivalent to calling `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`.

Like function objects, bound method objects support getting arbitrary attributes. However, since method attributes are actually stored on the underlying function object (`method.__func__`), setting method attributes on bound methods is disallowed. Attempting to set an attribute on a method results in an *AttributeError* being raised. In order to set a method attribute, you need to explicitly set it on the underlying function object:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

See instance-methods for more information.

4.15.5 코드 객체

Code objects are used by the implementation to represent "pseudo-compiled" executable Python code such as a function body. They differ from function objects because they don' t contain a reference to their global execution environment. Code objects are returned by the built-in *compile()* function and can be extracted from function objects through their `__code__` attribute. See also the *code* module.

Accessing `__code__` raises an *auditing event* `object.__getattr__` with arguments `obj` and `"__code__"`.

코드 객체는 *exec()* 또는 *eval()* 내장 함수에 (소스 문자열 대신) 전달하여 실행하거나 값을 구할 수 있습니다.

자세한 정보는 types를 보십시오.

4.15.6 형 객체

형 객체는 다양한 객체 형을 나타냅니다. 객체의 형은 내장 함수 *type()* 으로 액세스할 수 있습니다. 형에는 특별한 연산이 없습니다. 표준 모듈 *types* 는 모든 표준 내장형의 이름을 정의합니다.

형은 다음과 같이 쓰입니다: `<class 'int'>`.

4.15.7 널 객체

이 객체는 명시적으로 값을 돌려주지 않는 함수에 의해 반환됩니다. 특별한 연산을 지원하지 않습니다. 정확하게 하나의 널 객체가 있으며, 이름은 `None`(내장 이름)입니다. `type(None)()` 은 같은 싱글톤을 만듭니다.

`None` 이라고 쓰입니다.

4.15.8 Ellipsis 객체

This object is commonly used used to indicate that something is omitted. It supports no special operations. There is exactly one ellipsis object, named `Ellipsis` (a built-in name). `type(Ellipsis)()` produces the `Ellipsis` singleton.

`Ellipsis` 나 `...` 로 쓰입니다.

In typical use, `...` as the `Ellipsis` object appears in a few different places, for instance:

- In type annotations, such as *callable arguments* or *tuple elements*.
- As the body of a function instead of a pass statement.
- In third-party libraries, such as Numpy' s slicing and striding.

Python also uses three dots in ways that are not `Ellipsis` objects, for instance:

- Doctest' s `ELLIPSIS`, as a pattern for missing content.
- The default Python prompt of the *interactive* shell when partial input is incomplete.

Lastly, the Python documentation often uses three dots in conventional English usage to mean omitted content, even in code examples that also use them as the `Ellipsis`.

4.15.9 NotImplemented 객체

This object is returned from comparisons and binary operations when they are asked to operate on types they don' t support. See comparisons for more information. There is exactly one `NotImplemented` object. `type(NotImplemented)()` produces the singleton instance.

It is written as `NotImplemented`.

4.15.10 내부 객체

See types for this information. It describes stack frame objects, traceback objects, and slice objects.

4.16 특수 어트리뷰트

관련성이 있을 때, 구현은 몇 가지 객체 유형에 몇 가지 특수 읽기 전용 어트리뷰트를 추가합니다. 이 중 일부는 `dir()` 내장 함수에 의해 보고되지 않습니다.

`definition.`**`__name__`**

클래스, 함수, 메서드, 디스크립터 또는 제너레이터 인스턴스의 이름.

`definition.`**`__qualname__`**

클래스, 함수, 메서드, 디스크립터 또는 제너레이터 인스턴스의 정규화된 이름.

Added in version 3.3.

`definition.`**`__module__`**

The name of the module in which a class or function was defined.

`definition.`**`__doc__`**

The documentation string of a class or function, or `None` if undefined.

`definition.`**`__type_params__`**

The type parameters of generic classes, functions, and *type aliases*. For classes and functions that are not generic, this will be an empty tuple.

Added in version 3.12.

4.17 Integer string conversion length limitation

CPython has a global limit for converting between `int` and `str` to mitigate denial of service attacks. This limit *only* applies to decimal or other non-power-of-two number bases. Hexadecimal, octal, and binary conversions are unlimited. The limit can be configured.

The `int` type in CPython is an arbitrary length number stored in binary form (commonly known as a "bignum"). There exists no algorithm that can convert a string to a binary integer or a binary integer to a string in linear time, *unless* the base is a power of 2. Even the best known algorithms for base 10 have sub-quadratic complexity. Converting a large value such as `int('1' * 500_000)` can take over a second on a fast CPU.

Limiting conversion size offers a practical way to avoid **CVE 2020-10735**.

The limit is applied to the number of digit characters in the input or output string when a non-linear conversion algorithm would be involved. Underscores and the sign are not counted towards the limit.

When an operation would exceed the limit, a `ValueError` is raised:

```
>>> import sys
>>> sys.set_int_max_str_digits(4300)  # Illustrative, this is the default.
>>> _ = int('2' * 5432)
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300 digits) for integer string conversion: value␣
↪has 5432 digits; use sys.set_int_max_str_digits() to increase the limit
>>> i = int('2' * 4300)
>>> len(str(i))
4300
>>> i_squared = i*i
>>> len(str(i_squared))
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300 digits) for integer string conversion; use sys.
↪set_int_max_str_digits() to increase the limit
>>> len(hex(i_squared))
7144
>>> assert int(hex(i_squared), base=16) == i*i  # Hexadecimal is unlimited.
```

The default limit is 4300 digits as provided in `sys.int_info.default_max_str_digits`. The lowest limit that can be configured is 640 digits as provided in `sys.int_info.str_digits_check_threshold`.

Verification:

```
>>> import sys
>>> assert sys.int_info.default_max_str_digits == 4300, sys.int_info
>>> assert sys.int_info.str_digits_check_threshold == 640, sys.int_info
>>> msg = int('578966293710682886880994035146873798396722250538762761564'
...           '925292551438391548333381274358054977943610470626069636660 0'
...           '571186405732').to_bytes(53, 'big')
...
```

Added in version 3.11.

4.17.1 Affected APIs

The limitation only applies to potentially slow conversions between `int` and `str` or `bytes`:

- `int(string)` with default base 10.
- `int(string, base)` for all bases that are not a power of 2.
- `str(integer)`.

- `repr(integer)`.
- any other string conversion to base 10, for example `f"{integer}"`, `"{}".format(integer)`, or `b"%d" % integer`.

The limitations do not apply to functions with a linear algorithm:

- `int(string, base)` with base 2, 4, 8, 16, or 32.
- *`int.from_bytes()`* and *`int.to_bytes()`*.
- *`hex()`*, *`oct()`*, *`bin()`*.
- 포맷 명세 미니 언어 for hex, octal, and binary numbers.
- *`str`* to *`float`*.
- *`str`* to *`decimal.Decimal`*.

4.17.2 Configuring the limit

Before Python starts up you can use an environment variable or an interpreter command line flag to configure the limit:

- `PYTHONINTMAXSTRDIGITS`, e.g. `PYTHONINTMAXSTRDIGITS=640 python3` to set the limit to 640 or `PYTHONINTMAXSTRDIGITS=0 python3` to disable the limitation.
- `-X int_max_str_digits`, e.g. `python3 -X int_max_str_digits=640`
- *`sys.flags.int_max_str_digits`* contains the value of `PYTHONINTMAXSTRDIGITS` or `-X int_max_str_digits`. If both the env var and the `-X` option are set, the `-X` option takes precedence. A value of *-1* indicates that both were unset, thus a value of *`sys.int_info.default_max_str_digits`* was used during initialization.

From code, you can inspect the current limit and set a new one using these *`sys`* APIs:

- *`sys.get_int_max_str_digits()`* and *`sys.set_int_max_str_digits()`* are a getter and setter for the interpreter-wide limit. Subinterpreters have their own limit.

Information about the default and minimum can be found in *`sys.int_info`*:

- *`sys.int_info.default_max_str_digits`* is the compiled-in default limit.
- *`sys.int_info.str_digits_check_threshold`* is the lowest accepted value for the limit (other than 0 which disables it).

Added in version 3.11.

> **조심**
>
> Setting a low limit *can* lead to problems. While rare, code exists that contains integer constants in decimal in their source that exceed the minimum threshold. A consequence of setting the limit is that Python source code containing decimal integer literals longer than the limit will encounter an error during parsing, usually at startup time or import time or even at installation time - anytime an up to date `.pyc` does not already exist for the code. A workaround for source that contains such large constants is to convert them to `0x` hexadecimal form as it has no limit.
>
> Test your application thoroughly if you use a low limit. Ensure your tests run with the limit set early via the environment or flag so that it applies during startup and even during any installation step that may invoke Python to precompile `.py` sources to `.pyc` files.

4.17.3 Recommended configuration

The default *sys.int_info.default_max_str_digits* is expected to be reasonable for most applications. If your application requires a different limit, set it from your main entry point using Python version agnostic code as these APIs were added in security patch releases in versions before 3.12.

Example:

```
>>> import sys
>>> if hasattr(sys, "set_int_max_str_digits"):
...     upper_bound = 68000
...     lower_bound = 4004
...     current_limit = sys.get_int_max_str_digits()
...     if current_limit == 0 or current_limit > upper_bound:
...         sys.set_int_max_str_digits(upper_bound)
...     elif current_limit < lower_bound:
...         sys.set_int_max_str_digits(lower_bound)
```

If you need to disable it entirely, set it to `0`.

CHAPTER 5

내장 예외

파이썬에서, 모든 예외는 *BaseException* 에서 파생된 클래스의 인스턴스여야 합니다. 특정 클래스를 언급하는 `except` 절을 갖는 `try` 문에서, 그 절은 그 클래스에서 파생된 모든 예외 클래스를 처리합니다 (하지만 그것 이 계승하는 예외 클래스는 처리하지 않습니다). 서브 클래싱을 통해 관련되지 않은 두 개의 예외 클래스는 같은 이름을 갖는다 할지라도 결코 등등하게 취급되지 않습니다.

이 장에서 나열된 내장 예외는 인터프리터나 내장 함수에 의해 생성될 수 있습니다. 따로 언급된 경우를 제외하고는, 에러의 자세한 원인을 나타내는 "연관된 값" 을 갖습니다. 이것은 여러 항목의 정보 (예, 에러 코드와 그 코드를 설명하는 문자열)를 담은 문자열이나 튜플 일 수 있습니다. 연관된 값은 보통 예외 클래스의 생성자에 인자로 전달됩니다.

사용자 코드는 내장 예외를 일으킬 수 있습니다. 이것은 예외 처리기를 검사하거나 인터프리터가 같은 예외를 발생시키는 상황과 "같은" 에러 조건을 보고하는 데 사용할 수 있습니다. 그러나 사용자 코드가 부적절한 에러를 발생시키는 것을 막을 방법이 없음을 유의하십시오.

내장 예외 클래스는 새 예외를 정의하기 위해 서브 클래싱 될 수 있습니다. *BaseException* 이 아니라 *Exception* 클래스 나 그 서브 클래스 중 하나에서 새로운 예외를 파생시킬 것을 권장합니다. 예외 정의에 대한 더 많은 정보는 파이썬 자습서의 tut-userexceptions 에 있습니다.

5.1 예외 컨텍스트

Three attributes on exception objects provide information about the context in which the exception was raised:

`BaseException.`**`__context__`**

`BaseException.`**`__cause__`**

`BaseException.`**`__suppress_context__`**

When raising a new exception while another exception is already being handled, the new exception' s `__context__` attribute is automatically set to the handled exception. An exception may be handled when an `except` or `finally` clause, or a `with` statement, is used.

이 묵시적인 예외 컨텍스트는 암시적인 예외 상황은 명시적 원인으로 보충될 수 있는데, `raise`와 `from`을 사용합니다:

```
raise new_exc from original_exc
```

`from` 다음의 표현식은 예외이거나 `None` 이어야 합니다. 이 표현식을 새로 일으키는 예외의 `__cause__` 로 설정합니다. `__cause__` 를 설정하면, 묵시적으로 `__suppress_context__` 를 `True` 로 설정합니다. 그래서, `raise new_exc from None` 을 사용하면 표시의 목적상 이전 예외를 새로운

것으로 대체 하는 효과를 주면서 (예를 들어 *KeyError* 를 *AttributeError* 로), 디버깅할 때 검사할 수 있도록 이전의 예외를 `__context__` 에 남겨둡니다.

기본 트레이스백 표시 코드는 예외 자체의 트레이스백 뿐만 아니라 이러한 연결된 예외를 보여줍니다. `__cause__` 에 명시적으로 연결된 예외는 있으면 항상 표시됩니다. `__context__` 에 묵시적으로 연결된 예외는 `__cause__` 가 *None* 이고 `__suppress_context__` 가 거짓인 경우에만 표시됩니다.

두 경우 모두, 예외 자신은 항상 연결된 예외 뒤에 표시되어서, 트레이스백의 마지막 줄은 항상 마지막에 발생한 예외를 보여줍니다.

5.2 내장 예외를 상속받기

User code can create subclasses that inherit from an exception type. It' s recommended to only subclass one exception type at a time to avoid any possible conflicts between how the bases handle the `args` attribute, as well as due to possible memory layout incompatibilities.

CPython 구현 상세: Most built-in exceptions are implemented in C for efficiency, see: Objects/exceptions.c. Some have custom memory layouts which makes it impossible to create a subclass that inherits from multiple exception types. The memory layout of a type is an implementation detail and might change between Python versions, leading to new conflicts in the future. Therefore, it' s recommended to avoid subclassing multiple exception types altogether.

5.3 베이스 클래스

다음 예외는 주로 다른 예외의 베이스 클래스로 사용됩니다.

exception BaseException

모든 내장 예외의 베이스 클래스입니다. 사용자 정의 클래스에 의해 직접 상속되는 것이 아닙니다 (그런 목적으로는 *Exception*을 사용하세요). 이 클래스의 인스턴스에 대해 *str()* 이 호출되면, 인스턴스로 전달된 인자(들)의 표현을 돌려줍니다. 인자가 없는 경우는 빈 문자열을 돌려줍니다.

args

예외 생성자에 주어진 인자들의 튜플. 일부 내장 예외(예, *OSError*)는 특정 수의 인자를 기대하고 이 튜플의 요소에 특별한 의미를 할당하는 반면, 다른 것들은 보통 오류 메시지를 제공하는 단일 문자열로만 호출됩니다.

with_traceback(*tb*)

This method sets *tb* as the new traceback for the exception and returns the exception object. It was more commonly used before the exception chaining features of **PEP 3134** became available. The following example shows how we can convert an instance of `SomeException` into an instance of `OtherException` while preserving the traceback. Once raised, the current frame is pushed onto the traceback of the `OtherException`, as would have happened to the traceback of the original `SomeException` had we allowed it to propagate to the caller.

```
try:
    ...
except SomeException:
    tb = sys.exception().__traceback__
    raise OtherException(...).with_traceback(tb)
```

__traceback__

A writable field that holds the traceback object associated with this exception. See also: raise.

add_note(*note*)

Add the string `note` to the exception' s notes which appear in the standard traceback after the exception string. A *TypeError* is raised if `note` is not a string.

Added in version 3.11.

__notes__

A list of the notes of this exception, which were added with `add_note()`. This attribute is created when `add_note()` is called.

Added in version 3.11.

exception Exception

모든 시스템 종료 외의 내장 예외는 이 클래스 파생됩니다. 모든 사용자 정의 예외도 이 클래스에서 파생되어야 합니다.

exception ArithmeticError

다양한 산술 에러가 일으키는 내장 예외들의 베이스 클래스: `OverflowError`, `ZeroDivisionError`, `FloatingPointError`.

exception BufferError

버퍼 관련 연산을 수행할 수 없을 때 발생합니다.

exception LookupError

매핑 또는 시퀀스에 사용된 키 나 인덱스가 잘못되었을 때 발생하는 예외의 베이스 클래스: `IndexError`, `KeyError`. `codecs.lookup()` 은 이 예외를 직접 일으킬 수 있습니다.

5.4 구체적인 예외

다음 예외는 일반적으로 직접 일으키는데 사용하는 예외입니다.

exception AssertionError

`assert` 문이 실패할 때 발생합니다.

exception AttributeError

어트리뷰트 참조(attribute-references를 보세요)나 대입이 실패할 때 발생합니다. (객체가 어트리뷰트 참조나 어트리뷰트 대입을 아예 지원하지 않으면 `TypeError` 가 발생합니다.)

The optional *name* and *obj* keyword-only arguments set the corresponding attributes:

name

The name of the attribute that was attempted to be accessed.

obj

The object that was accessed for the named attribute.

버전 3.10에서 변경: `name`과 `obj` 어트리뷰트를 추가했습니다.

exception EOFError

Raised when the `input()` function hits an end-of-file condition (EOF) without reading any data. (Note: the `io.IOBase.read()` and `io.IOBase.readline()` methods return an empty string when they hit EOF.)

exception FloatingPointError

현재 사용되지 않습니다.

exception GeneratorExit

제너레이터 또는 코루틴 이 닫힐 때 발생합니다; `generator.close()` 와 `coroutine.close()` 를 보십시오. 기술적으로 에러가 아니므로 `Exception` 대신에 `BaseException` 을 직접 계승합니다.

exception ImportError

`import` 문이 모듈을 로드하는 데 문제가 있을 때 발생합니다. 또한 `from ... import` 에서 임포트 하려는 이름을 찾을 수 없을 때도 발생합니다.

The optional *name* and *path* keyword-only arguments set the corresponding attributes:

name

임포트가 시도된 모듈의 이름.

path

The path to any file which triggered the exception.

버전 3.3에서 변경: *name*과 *path* 어트리뷰트를 추가했습니다.

exception ModuleNotFoundError

ImportError 의 서브 클래스인데, 모듈을 찾을 수 없을 때 `import` 가 일으킵니다. *sys.modules* 에서 `None` 이 발견될 때도 발생합니다.

Added in version 3.6.

exception IndexError

시퀀스 인덱스가 범위를 벗어날 때 발생합니다. (슬라이스 인덱스는 허용된 범위 내에 들어가도록 자동으로 잘립니다; 인덱스가 정수가 아니면 *TypeError* 가 발생합니다.)

exception KeyError

매핑 (딕셔너리) 키가 기존 키 집합에서 발견되지 않을 때 발생합니다.

exception KeyboardInterrupt

사용자가 인터럽트 키(일반적으로 `Control-C` 또는 `Delete`)를 누를 때 발생합니다. 실행 중에 인터럽트 검사가 정기적으로 수행됩니다. *Exception*을 잡는 코드에 의해 우연히 잡혀서, 인터프리터가 종료하는 것을 막지 못하도록 *BaseException* 를 계승합니다.

> **참고**
>
> Catching a *KeyboardInterrupt* requires special consideration. Because it can be raised at unpredictable points, it may, in some circumstances, leave the running program in an inconsistent state. It is generally best to allow *KeyboardInterrupt* to end the program as quickly as possible or avoid raising it entirely. (See 신호 처리기와 예외에 대한 참고 사항.)

exception MemoryError

작업에 메모리가 부족하지만, 상황이 여전히 (일부 객체를 삭제해서) 복구될 수 있는 경우 발생합니다. 연관된 값은 어떤 종류의 (내부) 연산이 메모리를 다 써 버렸는지를 나타내는 문자열입니다. 하부 메모리 관리 아키텍처(C의 `malloc()` 함수)때문에, 인터프리터가 항상 이 상황을 완벽하게 복구할 수 있는 것은 아닙니다; 그런데도 통제를 벗어난 프로그램이 원인인 경우를 위해, 스택 트레이스백을 인쇄할 수 있도록 예외를 일으킵니다.

exception NameError

지역 또는 전역 이름을 찾을 수 없을 때 발생합니다. 이는 정규화되지 않은 이름에만 적용됩니다. 연관된 값은 찾을 수 없는 이름을 포함하는 에러 메시지입니다.

The optional *name* keyword-only argument sets the attribute:

name

The name of the variable that was attempted to be accessed.

버전 3.10에서 변경: *name* 어트리뷰트를 추가했습니다.

exception NotImplementedError

이 예외는 *RuntimeError* 에서 파생됩니다. 사용자 정의 베이스 클래스에서, 파생 클래스가 재정의하도록 요구하는 추상 메서드나, 클래스가 개발되는 도중에 실제 구현이 추가될 필요가 있음을 나타낼 때 이 예외를 발생시켜야 합니다.

> **참고**
>
> 연산자 나 메서드가 아예 지원되지 않는다는 것을 나타내는 데 사용해서는 안 됩니다 – 그 경우는 연산자 / 메서드를 정의하지 않거나, 서브 클래스면 *None* 으로 설정하십시오.

> **조심**
>
> `NotImplementedError` 와 `NotImplemented` 는 바꿔쓸 수 없습니다. 이 예외는 위에서 설명한 대로만 사용되어야 합니다; 내장 상수의 올바른 사용에 대한 자세한 내용은 `NotImplemented` 를 참조하세요.

exception OSError([*arg*])

exception OSError(*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]]])

이 예외는 시스템 함수가 시스템 관련 에러를 돌려줄 때 발생하는데, "파일을 찾을 수 없습니다(file not found)" 나 "디스크가 꽉 찼습니다(disk full)" 와 같은 (잘못된 인자형이나 다른 부수적인 에러가 아닌) 입출력 실패를 포함합니다.

생성자의 두 번째 형식은 아래에 설명된 해당 어트리뷰트를 설정합니다. 어트리뷰트를 지정하지 않으면 기본적으로 `None` 이 됩니다. 이전 버전과의 호환성을 위해, 세 개의 인자가 전달되면, `args` 어트리뷰트는 처음 두 생성자 인자의 2-튜플만 포함합니다.

아래의 *OS 예외* 에서 설명하는 것처럼, 생성자는 종종 `OSError` 의 서브 클래스를 돌려줍니다. 구체적인 서브 클래스는 최종 `errno` 값에 따라 다릅니다. 이 동작은 `OSError` 를 직접 혹은 별칭을 통해 생성할 때만 일어나고, 서브 클래싱할 때는 상속되지 않습니다.

errno

C 변수 `errno` 로부터 온 숫자 에러 코드.

winerror

윈도우에서, 네이티브 윈도우 에러 코드를 제공합니다. `errno` 어트리뷰트는 이 네이티브 에러 코드를 POSIX 코드로 대략 변환한 것입니다.

윈도우에서, *winerror* 생성자 인자가 정수인 경우, `errno` 어트리뷰트는 윈도우 에러 코드에서 결정되며 *errno* 인자는 무시됩니다. 다른 플랫폼에서는 *winerror* 인자가 무시되고 `winerror` 어트리뷰트가 없습니다.

strerror

The corresponding error message, as provided by the operating system. It is formatted by the C functions `perror()` under POSIX, and `FormatMessage()` under Windows.

filename

filename2

(`open()` 또는 `os.unlink()` 와 같은) 파일 시스템 경로와 관련된 예외의 경우, `filename` 은 함수에 전달 된 파일 이름입니다. (`os.rename()` 처럼) 두 개의 파일 시스템 경로를 수반하는 함수의 경우, `filename2` 는 두 번째 파일 이름에 해당합니다.

버전 3.3에서 변경: `EnvironmentError`, `IOError`, `WindowsError`, `socket.error`, `select.error` and `mmap.error` have been merged into `OSError`, and the constructor may return a subclass.

버전 3.4에서 변경: `filename` 속성은 이제 파일 시스템 인코딩과 에러 처리기로 인코딩되거나 디코딩된 이름 대신 함수에 전달 된 원래 파일 이름입니다. 또한 *filename2* 생성자 인자 및 어트리뷰트가 추가되었습니다.

exception OverflowError

산술 연산의 결과가 너무 커서 표현할 수 없을 때 발생합니다. 정수에서는 발생하지 않습니다 (포기하기보다는 `MemoryError` 를 일으키게 될 겁니다). 그러나, 역사적인 이유로, 때로 OverflowError 는 요구되는 범위를 벗어난 정수의 경우도 발생합니다. C에서 부동 소수점 예외 처리의 표준화가 부족하므로, 대부분의 부동 소수점 연산은 검사되지 않습니다.

exception PythonFinalizationError

This exception is derived from `RuntimeError`. It is raised when an operation is blocked during interpreter shutdown also known as *Python finalization*.

Examples of operations which can be blocked with a `PythonFinalizationError` during the Python finalization:

- Creating a new Python thread.
- *Joining* a running daemon thread.
- *os.fork()*.

See also the *sys.is_finalizing()* function.

Added in version 3.13: 이전에는 평범한 *RuntimeError* 가 발생했습니다.

버전 3.14에서 변경: *threading.Thread.join()* can now raise this exception.

exception RecursionError

이 예외는 *RuntimeError* 에서 파생됩니다. 인터프리터가 최대 재귀 깊이(*sys.getrecursionlimit()* 참조)가 초과하였음을 감지할 때 발생합니다.

Added in version 3.5: 이전에는 평범한 *RuntimeError* 가 발생했습니다.

exception ReferenceError

이 예외는 *weakref.proxy()* 함수가 만든 약한 참조 프락시가 이미 가비지 수집된 참조 대상의 어트리뷰트를 액세스하는 데 사용될 때 발생합니다. 약한 참조에 대한 더 자세한 정보는 *weakref* 모듈을 보십시오.

exception RuntimeError

다른 범주에 속하지 않는 에러가 감지될 때 발생합니다. 연관된 값은 정확히 무엇이 잘못되었는지를 나타내는 문자열입니다.

exception StopIteration

이터레이터에 의해 생성된 항목이 더 없다는 것을 알려주기 위해, 내장 함수 *next()*와 이터레이터의 *__next__()* 메서드가 일으킵니다.

value

예외 객체는 `value`라는 하나의 어트리뷰트를 가지고 있습니다. 이 어트리뷰트는 예외를 생성할 때 인자로 주어지며, 기본값은 *None* 입니다.

제너레이터 나 코루틴 함수가 복귀할 때, 새 *StopIteration* 인스턴스를 발생시키고, 함수가 돌려주는 값을 예외 생성자의 *value* 매개변수로 사용합니다.

제너레이터 코드가 직간접적으로 *StopIteration* 를 일으키면, *RuntimeError* 로 변환됩니다 (*StopIteration* 은 새 예외의 원인(`__cause__`)으로 남겨둡니다).

버전 3.3에서 변경: `value` 어트리뷰트와 제너레이터 함수가 이 값을 돌려주는 기능을 추가했습니다.

버전 3.5에서 변경: `from __future__ import generator_stop` 를 통한 RuntimeError 변환을 도입했습니다. **PEP 479**를 참조하세요.

버전 3.7에서 변경: 기본적으로 모든 코드에서 **PEP 479**를 활성화합니다: 제너레이터에서 발생한 *StopIteration* 에러는 *RuntimeError* 로 변환됩니다.

exception StopAsyncIteration

반드시 비동기 이터레이터 객체의 `__anext__()` 메서드가 이터레이션을 멈추고자 할 때 발생시켜야 합니다.

Added in version 3.5.

exception SyntaxError(*message*, *details*)

파서가 문법 오류를 만날 때 발생합니다. `import` 문에서, 내장 함수 *compile()*, *exec()* 또는 *eval()* 호출에서, 초기 스크립트나 (대화형으로) 표준 입력을 읽을 때 발생할 수 있습니다.

예외 인스턴스의 *str()*은 에러 메시지만 돌려줍니다. details는 각 멤버가 별도의 어트리뷰트로도 제공되는 튜플입니다.

filename

문법 오류가 발생한 파일의 이름.

lineno

오류가 발생한 파일의 줄 번호. 1-인덱싱됩니다: 파일의 첫 번째 줄은 `lineno`가 1입니다.

offset

오류가 발생한 줄의 열. 1-인덱싱됩니다: 줄의 첫 번째 문자는 `offset` 이 1 입니다.

text

오류를 수반한 소스 코드 텍스트.

end_lineno

오류가 발생한 파일의 끝나는 줄 번호. 1-인덱싱됩니다: 파일의 첫 번째 줄은 `lineno` 가 1 입니다.

end_offset

발생한 오류가 끝나는 줄의 열. 1-인덱싱됩니다: 줄의 첫 번째 문자는 `offset` 이 1 입니다.

For errors in f-string fields, the message is prefixed by "f-string: " and the offsets are offsets in a text constructed from the replacement expression. For example, compiling f' Bad {a b} field' results in this args attribute: ('f-string: ⋯' , ('' , 1, 2, '(a b)n' , 1, 5)).

버전 3.10에서 변경: *end_lineno*과 *end_offset* 어트리뷰트를 추가했습니다.

exception IndentationError

잘못된 들여쓰기와 관련된 문법 오류의 베이스 클래스입니다. *SyntaxError* 의 서브 클래스입니다.

exception TabError

들여쓰기가 일관성없는 탭과 스페이스 사용을 포함하는 경우 발생합니다. *IndentationError* 의 서브 클래스입니다.

exception SystemError

인터프리터가 내부 에러를 발견했지만, 모든 희망을 포기할 만큼 상황이 심각해 보이지는 않을 때 발생합니다. 연관된 값은 무엇이 잘못되었는지 (저수준의 용어로) 나타내는 문자열입니다. *CPython*에서, 이는 가령 예외를 설정하지 않고 `NULL`을 반환하는 것과 같이, 파이썬의 C API를 잘못 사용함으로써 발생할 수 있습니다.

이 예외가 여러분이나 사용 중인 패키지의 책임이 아니라고 확신한다면, 파이썬 인터프리터의 저자나 관리자에게 알려야 합니다. 파이썬 인터프리터의 버전 (`sys.version`; 대화식 파이썬 세션의 시작 부분에도 출력됩니다), 정확한 에러 메시지 (예외의 연관된 값) 그리고 가능하다면 에러를 일으킨 프로그램의 소스를 제공해 주십시오.

exception SystemExit

This exception is raised by the *sys.exit()* function. It inherits from *BaseException* instead of *Exception* so that it is not accidentally caught by code that catches *Exception*. This allows the exception to properly propagate up and cause the interpreter to exit. When it is not handled, the Python interpreter exits; no stack traceback is printed. The constructor accepts the same optional argument passed to *sys.exit()*. If the value is an integer, it specifies the system exit status (passed to C' s `exit()` function); if it is `None`, the exit status is zero; if it has another type (such as a string), the object' s value is printed and the exit status is one.

sys.exit() 에 대한 호출은 예외로 변환되어 뒷정리 처리기 (`try` 문의 `finally` 절) 가 실행될 수 있도록 합니다. 그래서 디버거는 제어권을 잃을 위험 없이 스크립트를 실행할 수 있습니다. 즉시 종료가 절대적으로 필요한 경우에는 *os._exit()* 함수를 사용할 수 있습니다 (예를 들어, *os.fork()* 호출 후의 자식 프로세스에서).

code

생성자에 전달되는 종료 상태 또는 에러 메시지입니다. (기본값은 `None` 입니다.)

exception TypeError

연산이나 함수가 부적절한 형의 객체에 적용될 때 발생합니다. 연관된 값은 형 불일치에 대한 세부 정보를 제공하는 문자열입니다.

이 예외는 객체에 시도된 연산이 지원되지 않으며 그럴 의도도 없음을 나타내기 위해 사용자 코드가 발생시킬 수 있습니다. 만약 객체가 주어진 연산을 지원할 의사는 있지만, 아직 구현을 제공하지 않는 경우라면, *NotImplementedError* 를 발생시키는 것이 적합합니다.

잘못된 형의 인자를 전달하면 (가령 *int* 를 기대하는데 *list*를 전달하기), *TypeError* 를 일으켜야 합니다. 하지만 잘못된 값을 갖는 인자를 전달하면 (가령 범위를 넘어서는 숫자) *ValueError* 를 일으켜야 합니다.

exception UnboundLocalError

함수 나 메서드에서 지역 변수를 참조하지만, 해당 변수에 값이 연결되지 않으면 발생합니다. 이것은 *NameError* 의 서브 클래스입니다.

exception UnicodeError

유니코드 관련 인코딩 또는 디코딩 에러가 일어날 때 발생합니다. *ValueError* 의 서브 클래스입니다.

UnicodeError 는 인코딩이나 디코딩 에러를 설명하는 어트리뷰트를 가지고 있습니다. 예를 들어, `err.object[err.start:err.end]` 는 코덱이 실패한 잘못된 입력을 제공합니다.

encoding

에러를 발생시킨 인코딩의 이름입니다.

reason

구체적인 코덱 오류를 설명하는 문자열입니다.

object

코덱이 인코딩 또는 디코딩하려고 시도한 객체입니다.

start

object 에 있는 잘못된 데이터의 최초 인덱스입니다.

This value should not be negative as it is interpreted as an absolute offset but this constraint is not enforced at runtime.

end

object 에 있는 마지막으로 잘못된 데이터의 바로 다음 인덱스입니다.

This value should not be negative as it is interpreted as an absolute offset but this constraint is not enforced at runtime.

exception UnicodeEncodeError

인코딩 중에 유니코드 관련 에러가 일어나면 발생합니다. *UnicodeError* 의 서브 클래스입니다.

exception UnicodeDecodeError

디코딩 중에 유니코드 관련 에러가 일어나면 발생합니다. *UnicodeError* 의 서브 클래스입니다.

exception UnicodeTranslateError

번역 중에 유니코드 관련 에러가 일어나면 발생합니다. *UnicodeError* 의 서브 클래스입니다.

exception ValueError

연산이나 함수가 올바른 형이지만 부적절한 값을 가진 인자를 받았고, 상황이 *IndexError* 처럼 더 구체적인 예외로 설명되지 않는 경우 발생합니다.

exception ZeroDivisionError

나누기 또는 모듈로 연산의 두 번째 인자가 0일 때 발생합니다. 연관된 값은 피연산자의 형과 연산을 나타내는 문자열입니다.

다음 예외는 이전 버전과의 호환성을 위해 유지됩니다; 파이썬 3.3부터는 *OSError* 의 별칭입니다.

exception EnvironmentError

exception IOError

exception WindowsError

윈도우에서만 사용할 수 있습니다.

5.4.1 OS 예외

다음의 예외는 *OSError* 의 서브 클래스이며, 시스템 에러 코드에 따라 발생합니다.

exception BlockingIOError

비 블록 동작으로 설정된 객체(가령 소켓)에 블록이 필요한 연산이 수행되면 발생합니다. errno *EAGAIN*, *EALREADY*, *EWOULDBLOCK*, *EINPROGRESS* 에 해당합니다.

OSError 의 것 외에도, *BlockingIOError* 는 어트리뷰트를 하나 더 가질 수 있습니다:

characters_written

블록 되기 전에 스트림에 쓴 문자 수를 포함하는 정수. 이 어트리뷰트는 *io* 모듈에서 버퍼링 된 입출력 클래스를 사용할 때 쓸 수 있습니다.

exception ChildProcessError

자식 프로세스에 대한 작업이 실패할 때 발생합니다. errno *ECHILD* 에 해당합니다.

exception ConnectionError

연결 관련 문제에 대한 베이스 클래스입니다.

서브 클래스는 *BrokenPipeError*, *ConnectionAbortedError*, *ConnectionRefusedError* 및 *ConnectionResetError* 입니다.

exception BrokenPipeError

ConnectionError 의 서브 클래스로서, 다른 쪽 끝이 닫힌 파이프에 쓰려고 하거나, 쓰기가 종료된 소켓에 쓰려고 할 때 발생합니다. errno *EPIPE* 와 *ESHUTDOWN* 에 해당합니다.

exception ConnectionAbortedError

ConnectionError 의 서브 클래스로서, 연결 시도가 상대방에 의해 중단(abort)될 때 발생합니다. errno *ECONNABORTED* 에 해당합니다.

exception ConnectionRefusedError

ConnectionError 의 서브 클래스로서, 연결 시도가 상대방에 의해 거부(refuse)될 때 발생합니다. errno *ECONNREFUSED* 에 해당합니다.

exception ConnectionResetError

ConnectionError 의 서브 클래스로서, 연결이 상대방에 의해 강제 종료(reset)될 때 발생합니다. errno *ECONNRESET* 에 해당합니다.

exception FileExistsError

이미 존재하는 파일이나 디렉터리를 만들려고 할 때 발생합니다. errno *EEXIST* 에 해당합니다.

exception FileNotFoundError

파일이나 디렉터리가 요청되었지만 존재하지 않을 때 발생합니다. errno *ENOENT* 에 해당합니다.

exception InterruptedError

시스템 호출이 들어오는 시그널에 의해 중단될 때 발생합니다. errno *EINTR* 에 해당합니다.

버전 3.5에서 변경: 이제 파이썬은 시스템 호출이 시그널에 의해 중단될 때, 시그널 처리기가 예외를 일으키는 경우를 제외하고 (이유는 **PEP 475** 를 참조하세요), *InterruptedError* 를 일으키는 대신 시스템 호출을 재시도합니다.

exception IsADirectoryError

디렉터리에 파일 연산(가령 *os.remove()*)이 요청되었을 때 발생합니다. errno *EISDIR* 에 해당합니다.

exception NotADirectoryError

디렉터리가 아닌 것에 디렉터리 연산(가령 *os.listdir()*)이 요청되었을 때 발생합니다. 대부분의 POSIX 플랫폼에서, 연산이 디렉터리가 아닌 파일을 디렉터리인 것처럼 열거나 탐색하려고 할 때도 발생합니다. errno *ENOTDIR* 에 해당합니다.

exception PermissionError

적절한 접근권 (가령 파일 시스템 권한) 없이 연산을 실행하려고 할 때 발생합니다. errno *EACCES*, *EPERM* 및 *ENOTCAPABLE* 에 해당합니다.

버전 3.11.1에서 변경: WASI' s *ENOTCAPABLE* is now mapped to *PermissionError*.

exception ProcessLookupError

주어진 프로세스가 존재하지 않을 때 발생합니다. errno *ESRCH* 에 해당합니다.

exception TimeoutError

시스템 함수가 시스템 수준에서 시간 초과 될 때 발생합니다. errno *ETIMEDOUT* 에 해당합니다.

Added in version 3.3: 위의 모든 *OSError* 서브 클래스가 추가되었습니다.

> **더 보기**
>
> **PEP 3151** - OS 및 IO 예외 계층 구조 재작업

5.5 경고

다음 예외는 경고 범주로 사용됩니다; 자세한 정보는 경고 범주 설명서를 보십시오.

exception Warning

경고 범주의 베이스 클래스입니다.

exception UserWarning

사용자 코드에 의해 만들어지는 경고의 베이스 클래스입니다.

exception DeprecationWarning

폐지된 기능에 대한 경고의 베이스 클래스인데, 그 경고가 다른 파이썬 개발자를 대상으로 하는 경우입니다.

__main__ 모듈을 제외하고, 기본 경고 필터에 의해 무시됩니다 (**PEP 565**). 파이썬 개발 모드를 활성화하면 이 경고가 표시됩니다.

The deprecation policy is described in **PEP 387**.

exception PendingDeprecationWarning

더는 사용되지 않고 장래에 폐지될 예정이지만, 지금 당장 폐지되지는 않은 기능에 관한 경고의 베이스 클래스입니다.

앞으로 있을 수도 있는 폐지에 관한 경고는 일반적이지 않기 때문에, 이 클래스는 거의 사용되지 않습니다. 이미 활성화된 폐지에는 *DeprecationWarning*을 선호합니다.

기본 경고 필터에 의해 무시됩니다. 파이썬 개발 모드를 활성화하면 이 경고가 표시됩니다.

The deprecation policy is described in **PEP 387**.

exception SyntaxWarning

모호한 문법에 대한 경고의 베이스 클래스입니다.

This warning is typically emitted when compiling Python source code, and usually won' t be reported when running already compiled code.

exception RuntimeWarning

모호한 실행 시간 동작에 대한 경고의 베이스 클래스입니다.

exception FutureWarning

폐지된 기능에 대한 경고의 베이스 클래스인데, 그 경고가 파이썬으로 작성된 응용 프로그램의 최종 사용자를 대상으로 하는 경우입니다.

exception ImportWarning

모듈 임포트에 있을 수 있는 실수에 대한 경고의 베이스 클래스입니다.

기본 경고 필터에 의해 무시됩니다. 파이썬 개발 모드를 활성화하면 이 경고가 표시됩니다.

exception UnicodeWarning

유니코드와 관련된 경고의 베이스 클래스입니다.

exception EncodingWarning

인코딩과 관련된 경고의 베이스 클래스입니다.

See *Opt-in EncodingWarning* for details.

Added in version 3.10.

exception BytesWarning

`bytes` 및 `bytearray` 와 관련된 경고의 베이스 클래스입니다.

exception ResourceWarning

자원 사용과 관련된 경고의 베이스 클래스입니다.

기본 경고 필터에 의해 무시됩니다. 파이썬 개발 모드를 활성화하면 이 경고가 표시됩니다.

Added in version 3.2.

5.6 예외 그룹

The following are used when it is necessary to raise multiple unrelated exceptions. They are part of the exception hierarchy so they can be handled with `except` like all other exceptions. In addition, they are recognised by `except*`, which matches their subgroups based on the types of the contained exceptions.

exception ExceptionGroup(*msg*, *excs*)

exception BaseExceptionGroup(*msg*, *excs*)

Both of these exception types wrap the exceptions in the sequence `excs`. The `msg` parameter must be a string. The difference between the two classes is that `BaseExceptionGroup` extends `BaseException` and it can wrap any exception, while `ExceptionGroup` extends `Exception` and it can only wrap subclasses of `Exception`. This design is so that `except Exception` catches an `ExceptionGroup` but not `BaseExceptionGroup`.

The `BaseExceptionGroup` constructor returns an `ExceptionGroup` rather than a `BaseExceptionGroup` if all contained exceptions are `Exception` instances, so it can be used to make the selection automatic. The `ExceptionGroup` constructor, on the other hand, raises a `TypeError` if any contained exception is not an `Exception` subclass.

message

The `msg` argument to the constructor. This is a read-only attribute.

exceptions

A tuple of the exceptions in the `excs` sequence given to the constructor. This is a read-only attribute.

subgroup(*condition*)

Returns an exception group that contains only the exceptions from the current group that match *condition*, or `None` if the result is empty.

The condition can be an exception type or tuple of exception types, in which case each exception is checked for a match using the same check that is used in an `except` clause. The condition can also be a callable (other than a type object) that accepts an exception as its single argument and returns true for the exceptions that should be in the subgroup.

The nesting structure of the current exception is preserved in the result, as are the values of its `message`, `__traceback__`, `__cause__`, `__context__` and `__notes__` fields. Empty nested groups are omitted from the result.

The condition is checked for all exceptions in the nested exception group, including the top-level and any nested exception groups. If the condition is true for such an exception group, it is included in the result in full.

Added in version 3.13: `condition` can be any callable which is not a type object.

split(*condition*)

Like *subgroup()*, but returns the pair `(match, rest)` where `match` is `subgroup(condition)` and `rest` is the remaining non-matching part.

derive(*excs*)

Returns an exception group with the same *message*, but which wraps the exceptions in `excs`.

This method is used by *subgroup()* and *split()*, which are used in various contexts to break up an exception group. A subclass needs to override it in order to make *subgroup()* and *split()* return instances of the subclass rather than *ExceptionGroup*.

subgroup() and *split()* copy the *__traceback__*, *__cause__*, *__context__* and *__notes__* fields from the original exception group to the one returned by *derive()*, so these fields do not need to be updated by *derive()*.

```
>>> class MyGroup(ExceptionGroup):
...     def derive(self, excs):
...         return MyGroup(self.message, excs)
...
>>> e = MyGroup("eg", [ValueError(1), TypeError(2)])
>>> e.add_note("a note")
>>> e.__context__ = Exception("context")
>>> e.__cause__ = Exception("cause")
>>> try:
...    raise e
... except Exception as e:
...    exc = e
...
>>> match, rest = exc.split(ValueError)
>>> exc, exc.__context__, exc.__cause__, exc.__notes__
(MyGroup('eg', [ValueError(1), TypeError(2)]), Exception('context'),␣
↪Exception('cause'), ['a note'])
>>> match, match.__context__, match.__cause__, match.__notes__
(MyGroup('eg', [ValueError(1)]), Exception('context'), Exception('cause'),␣
↪['a note'])
>>> rest, rest.__context__, rest.__cause__, rest.__notes__
(MyGroup('eg', [TypeError(2)]), Exception('context'), Exception('cause'), [
↪'a note'])
>>> exc.__traceback__ is match.__traceback__ is rest.__traceback__
True
```

Note that *BaseExceptionGroup* defines `__new__()`, so subclasses that need a different constructor signature need to override that rather than `__init__()`. For example, the following defines an exception group subclass which accepts an exit_code and constructs the group' s message from it.

```
class Errors(ExceptionGroup):
   def __new__(cls, errors, exit_code):
      self = super().__new__(Errors, f"exit code: {exit_code}", errors)
      self.exit_code = exit_code
      return self

   def derive(self, excs):
      return Errors(excs, self.exit_code)
```

Like *ExceptionGroup*, any subclass of *BaseExceptionGroup* which is also a subclass of *Exception* can only wrap instances of *Exception*.

Added in version 3.11.

5.7 예외 계층 구조

내장 예외의 클래스 계층 구조는 다음과 같습니다:

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 ├── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
      │    ├── ChildProcessError
      │    ├── ConnectionError
      │    │    ├── BrokenPipeError
      │    │    ├── ConnectionAbortedError
      │    │    ├── ConnectionRefusedError
      │    │    └── ConnectionResetError
      │    ├── FileExistsError
      │    ├── FileNotFoundError
      │    ├── InterruptedError
      │    ├── IsADirectoryError
      │    ├── NotADirectoryError
      │    ├── PermissionError
      │    ├── ProcessLookupError
      │    └── TimeoutError
      ├── ReferenceError
      ├── RuntimeError
      │    ├── NotImplementedError
      │    ├── PythonFinalizationError
      │    └── RecursionError
      ├── StopAsyncIteration
      ├── StopIteration
      ├── SyntaxError
      │    ├── IndentationError
      │         └── TabError
      ├── SystemError
      ├── TypeError
      ├── ValueError
      │    ├── UnicodeError
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    □         □□□ UnicodeDecodeError
    □         □□□ UnicodeEncodeError
    □         □□□ UnicodeTranslateError
    □□□ Warning
         □□□ BytesWarning
         □□□ DeprecationWarning
         □□□ EncodingWarning
         □□□ FutureWarning
         □□□ ImportWarning
         □□□ PendingDeprecationWarning
         □□□ ResourceWarning
         □□□ RuntimeWarning
         □□□ SyntaxWarning
         □□□ UnicodeWarning
         □□□ UserWarning
```

CHAPTER 6

텍스트 처리 서비스

이 장에서 설명하는 모듈은 광범위한 문자열 조작 연산과 기타 텍스트 처리 서비스를 제공합니다.

바이너리 데이터 서비스 에 기술되어 있는 *codecs* 모듈 또한 텍스트 처리와 밀접한 관련이 있습니다. 또한, 텍스트 시퀀스 형 — *str* 에 있는 파이썬의 내장 문자열형에 대한 설명서를 참조하십시오.

6.1 string — 일반적인 문자열 연산

소스 코드: Lib/string.py

더 보기

텍스트 시퀀스 형 — *str*

문자열 메서드

6.1.1 문자열 상수

이 모듈에 정의된 상수는 다음과 같습니다:

string.**ascii_letters**

아래에 나오는 *ascii_lowercase*와 *ascii_uppercase* 상수를 이어붙인 것입니다. 이 값은 로케일에 의존적이지 않습니다.

string.**ascii_lowercase**

소문자 `'abcdefghijklmnopqrstuvwxyz'`. 이 값은 로케일에 의존적이지 않고 변경되지 않습니다.

string.**ascii_uppercase**

대문자 `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. 이 값은 로케일에 의존적이지 않고 변경되지 않습니다.

string.**digits**

문자열 `'0123456789'`.

string.**hexdigits**

문자열 `'0123456789abcdefABCDEF'`.

string.**octdigits**

문자열 '01234567'.

string.**punctuation**

C 로케일에서 구두점 문자로 간주하는 ASCII 문자의 문자열: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~.

string.**printable**

파이썬이 인쇄할 수 있는 것으로 간주하는 ASCII 문자의 문자열. *digits*, *ascii_letters*, *punctuation*, *whitespace* 의 조합입니다.

> **참고**
>
> By design, *string.printable.isprintable()* returns *False*. In particular, string.printable is not printable in the POSIX sense (see *LC_CTYPE*).

string.**whitespace**

공백으로 간주하는 모든 ASCII 문자를 포함하는 문자열. 여기에는 스페이스, 탭, 줄 바꿈, 캐리지 리턴, 세로 탭 및 폼 피드 문자가 포함됩니다.

6.1.2 사용자 지정 문자열 포매팅

내장 문자열 클래스는 **PEP 3101** 에 설명된 *format()* 메서드를 통해 복잡한 변수 치환 및 값 포매팅을 수행할 수 있는 기능을 제공합니다. *string* 모듈의 *Formatter* 클래스는 내장 *format()* 메서드와 같은 구현을 사용하여 자신만의 문자열 포매팅 동작을 만들고 사용자 정의할 수 있게 합니다.

class string.**Formatter**

Formatter 클래스에는 다음과 같은 공개 메서드가 있습니다:

format(*format_string*, /, **args*, ***kwargs*)

기본 API 메서드입니다. 포맷 문자열과 임의의 위치 및 키워드 인자의 집합을 받아들입니다. 이것은 *vformat()* 을 호출하는 래퍼일 뿐입니다.

버전 3.7에서 변경: 포맷 문자열 인자는 이제 위치 전용입니다.

vformat(*format_string*, *args*, *kwargs*)

이 함수는 실제 포맷 작업을 수행합니다. *args 와 **kwargs 문법을 사용하여 딕셔너리를 개별적인 인자로 언 패킹한 후 다시 패킹하는 대신 미리 정의된 인자 딕셔너리를 전달하고자 하는 경우를 위해 별도의 함수로 노출합니다. *vformat()* 은 포맷 문자열을 문자 데이터와 치환 필드로 분리하는 작업을 수행합니다. 아래에 설명된 다양한 메서드를 호출합니다.

이에 더해, *Formatter* 는 서브 클래스에 의해 대체될 목적으로 많은 메서드를 정의합니다:

parse(*format_string*)

format_string을 루핑하면서 튜플 (*literal_text*, *field_name*, *format_spec*, *conversion*) 의 이터러블을 반환합니다. 이것은 *vformat()* 이 문자열을 리터럴 텍스트와 치환 필드로 나누는 데 사용합니다.

The values in the tuple conceptually represent a span of literal text followed by a single replacement field. If there is no literal text (which can happen if two replacement fields occur consecutively), then *literal_text* will be a zero-length string. If there is no replacement field, then the values of *field_name*, *format_spec* and *conversion* will be None. The value of *field_name* is unmodified and auto-numbering of non-numbered positional fields is done by *vformat()*.

get_field(*field_name*, *args*, *kwargs*)

Given *field_name*, convert it to an object to be formatted. Auto-numbering of *field_name* returned from *parse()* is done by *vformat()* before calling this method. Returns a tuple (obj, used_key). The default version takes strings of the form defined in **PEP 3101**, such as "0[name]" or "label.title". *args* and *kwargs* are as passed in to *vformat()*. The return value *used_key* has the same meaning as the *key* parameter to *get_value()*.

get_value(*key*, *args*, *kwargs*)

지정된 필드의 값을 가져옵니다. *key* 인자는 정수 또는 문자열입니다. 정수의 경우, *args* 에 있는 위치 인자의 인덱스를 나타냅니다; 문자열인 경우, *kwargs* 에 있는 이름있는 인자를 나타냅니다.

args 매개 변수는 `vformat()` 의 위치 인자 목록으로 설정되고, *kwargs* 매개 변수는 키워드 인자 딕셔너리로 설정됩니다.

복합 필드 이름의 경우, 이러한 함수는 필드 이름의 첫 번째 구성 요소에 대해서만 호출됩니다; 후속 구성 요소는 일반 어트리뷰트 및 인덱싱 연산을 통해 처리됩니다.

그래서 예를 들어, 필드 표현식 '0.name' 은 `get_value()` 가 *key* 인자 0으로 호출되도록 합니다. `name` 어트리뷰트는 `get_value()` 가 반환한 후에 내장 `getattr()` 함수를 호출하여 조회합니다.

인덱스 또는 키워드가 존재하지 않는 항목을 참조하면, `IndexError` 나 `KeyError` 가 발생합니다.

check_unused_args(*used_args*, *args*, *kwargs*)

원하는 경우 사용하지 않는 인자를 검사하도록 구현합니다. 이 함수에 대한 인자는 포맷 문자열에서 참조되는 모든 인자 키의 집합과 (위치 인자의 경우 정수, 이름있는 인자의 경우 문자열), vformat 으로 전달된 *args* 와 *kwargs* 에 대한 참조입니다. 사용되지 않은 인자의 집합은 이 매개 변수들로 계산할 수 있습니다. `check_unused_args()` 는 검사가 실패할 경우 예외를 발생시킬 것으로 가정합니다.

format_field(*value*, *format_spec*)

`format_field()` 는 단순히 전역 `format()` 내장 함수를 호출합니다. 서브 클래스가 재정의할 수 있도록 메서드가 제공됩니다.

convert_field(*value*, *conversion*)

(`get_field()` 가 반환한) 값(value)을 (`parse()` 메서드가 반환하는 튜플에 있는 것과 같은) 주어진 변환 유형(conversion)으로 변환합니다. 기본 버전은 's' (str), 'r' (repr) 및 'a' (ascii) 변환 유형을 인식합니다.

6.1.3 포맷 문자열 문법

The `str.format()` method and the `Formatter` class share the same syntax for format strings (although in the case of `Formatter`, subclasses can define their own format string syntax). The syntax is related to that of formatted string literals and template string literals, but it is less sophisticated and, in particular, does not support arbitrary expressions in interpolations.

포맷 문자열에는 중괄호 `{}` 로 둘러싸인 "치환 필드" 가 들어 있습니다. 중괄호 안에 포함되지 않은 것은 리터럴 텍스트로 간주하며 변경되지 않고 그대로 출력으로 복사됩니다. 리터럴 텍스트에 중괄호를 포함해야 하는 경우, 중복으로 이스케이프 할 수 있습니다: `{{` 와 `}}`.

치환 필드의 문법은 다음과 같습니다:

```
replacement_field: "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name:        arg_name ("." attribute_name | "[" element_index "]")*
arg_name:          [identifier | digit+]
attribute_name:    identifier
element_index:     digit+ | index_string
index_string:      <any source character except "]"> +
conversion:        "r" | "s" | "a"
format_spec:       format-spec:format_spec
```

덜 형식적인 용어로, 치환 필드는 *field_name* 으로 시작할 수 있는데, 값이 포맷되어 출력에 치환 필드 대신 삽입될 객체를 지정합니다. *field_name* 다음에는 선택적으로 느낌표 `'!'` 가 앞에 오는 *conversion* 필드와 콜론 `':'` 이 앞에 오는 *format_spec* 이 옵니다. 이 값은 치환 값에 대해 기본값이 아닌 포맷을 지정합니다.

포맷 명세 미니 언어 섹션을 참고하십시오.

field_name 자체는 숫자나 키워드인 *arg_name* 으로 시작합니다. 숫자면 위치 인자를 나타내고, 키워드면 이름이 있는 키워드 인자를 나타냅니다. *arg_name*은 문자열에 `str.isdecimal()`을 호출했을 때 참을 반환한다면 숫자로 취급됩니다. 포맷 문자열의 숫자 arg_name이 0, 1, 2, … 순으로 나열되는 경우, (일부가

아니라) 전부 생략 할 수 있으며 숫자 0, 1, 2, … 이 순서대로 자동 삽입됩니다. *arg_name* 이 따옴표로 분리되어 있지 않기 때문에, 포맷 문자열 내에서 임의의 딕셔너리 키(예를 들어, `'10'` 이나 `':-]'`)를 지정할 수 없습니다. *arg_name* 다음에는 제한 없는 개수의 인덱스나 어트리뷰트 표현식이 올 수 있습니다. `'.name'` 형태의 표현식은 `getattr()` 을 사용하여 이름있는 어트리뷰트를 선택하는 반면, `'[index]'` 형태의 표현식은 `__getitem__()` 을 사용해서 인덱스 조회를 합니다.

버전 3.1 에서 변경: 위치 인자 지정자는 `str.format()` 에서 생략 할 수 있습니다. 그래서, `'{} {}'.format(a, b)` 는 `'{0} {1}'.format(a, b)` 과 동등합니다.

버전 3.4 에서 변경: 위치 인자 지정자는 `Formatter`에서 생략 할 수 있습니다.

몇 가지 간단한 포맷 문자열 예제:

```
"First, thou shalt count to {0}"  # 첫번째 위치 인자를 참조합니다
"Bring me a {}"                   # 묵시적으로 첫번째 위치 인자를 참조합니다
"From {} to {}"                   # "From {0} to {1}" 과 같습니다
"My quest is {name}"              # 키워드 인자 'name' 을 참조합니다
"Weight in tons {0.weight}"       # 첫번째 위치 인자의 'weight' 어트리뷰트
"Units destroyed: {players[0]}"   # 키워드 인자 'players' 의 첫번째 요소.
```

conversion 필드는 포매팅 전에 형 코어션을 일으킵니다. 보통은, 값을 포매팅하는 작업은 값 자체의 `__format__()` 메서드에 의해 수행됩니다. 그러나 어떤 경우에는 형 자신의 포매팅 정의를 무시하고 문자열로 포맷되도록 강제할 필요가 있습니다. `__format__()` 을 호출하기 전에 값을 문자열로 변환하면, 일반적인 포매팅 논리가 무시됩니다.

현재 세 가지 변환 플래그가 지원됩니다: `'!s'` 는 값에 `str()` 을 호출하고, `'!r'` 은 값에 `repr()` 을 호출하고, `'!a'` 는 값에 `ascii()` 를 호출합니다.

몇 가지 예:

```
"Harold's a clever {0!s}"         # 먼저 인자에 str() 을 호출합니다
"Bring out the holy {name!r}"     # 먼저 인자에 repr() 을 호출합니다
"More {!a}"                       # 먼저 인자에 ascii() 를 호출합니다
```

format_spec 필드에는 값을 표시하는 방법에 대한 명세가 포함되어 있는데, 필드 너비, 정렬, 채움, 십진 정밀도 등이 포함됩니다. 각 값 형은 자체 "포매팅 미니 언어" 또는 *format_spec* 의 해석을 정의 할 수 있습니다.

대부분의 내장형은 다음 절에서 설명하는 공통 포매팅 미니 언어를 지원합니다.

A *format_spec* 필드는 그 안에 중첩된 치환 필드를 포함할 수도 있습니다. 이러한 중첩 된 치환 필드에는 필드 이름, 변환 플래그 및 포맷 명세가 포함될 수 있지만, 더 깊은 중첩은 허용되지 않습니다. format_spec 내의 치환 필드는 *format_spec* 문자열이 해석되기 전에 치환됩니다. 이렇게 해서 값의 포매팅을 동적으로 지정할 수 있게 합니다.

몇 가지 예제는 포맷 예제 섹션을 보십시오.

포맷 명세 미니 언어

"Format specifications" are used within replacement fields contained within a format string to define how individual values are presented (see 포맷 문자열 문법, f-strings, and t-strings). They can also be passed directly to the built-in `format()` function. Each formattable type may define how the format specification is to be interpreted.

대부분의 내장형은 포맷 명세에 대해 다음 옵션을 구현하지만, 일부 포맷 옵션은 숫자 형에서만 지원됩니다.

일반적인 관례는 빈 포맷 명세가 값에 `str()` 을 호출한 것과 같은 결과를 만드는 것입니다. 비어 있지 않은 포맷 명세는 보통 결과를 수정합니다.

표준 포맷 지정자 의 일반적인 형식은 다음과 같습니다:

```
format_spec:        [options][width_and_precision][type]
options:            [[fill]align][sign]["z"]["#"]["0"]
fill:               <any character>
align:              "<" | ">" | "=" | "^"
sign:               "+" | "-" | " "
```

```
width_and_precision:     [width_with_grouping][precision_with_grouping]
width_with_grouping:     [width][grouping]
precision_with_grouping: "." [precision][grouping] | "." grouping
width:                   digit+
precision:               digit+
grouping:                "," | "_"
type:                    "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g"
                         | "G" | "n" | "o" | "s" | "x" | "X" | "%"
```

유효한 *align* 값이 지정되면, *fill* 문자가 앞에 나올 수 있는데 임의의 문자가 될 수 있고, 생략된 경우에는 스페이스가 기본값으로 사용됩니다. 포맷 문자열 리터럴 에서나 `str.format()` 메서드를 사용할 때는, 리터럴 중괄호("{" 또는 "}")를 *fill* 문자로 사용할 수 없습니다. 그러나, 중첩된 치환 필드로 중괄호를 삽입할 수 있습니다. 이 제한은 `format()` 함수에는 영향을 미치지 않습니다.

다양한 정렬 옵션의 의미는 다음과 같습니다:

옵션	의미
'<'	사용 가능한 공간 내에서 필드가 왼쪽 정렬되도록 합니다 (대부분 객체에서 이것이 기본값입니다).
'>'	사용 가능한 공간 내에서 필드가 오른쪽 정렬되도록 합니다 (숫자에서 이것이 기본값입니다).
'='	채움이 부호 (있다면) 뒤에, 숫자 앞에 오도록 강제합니다. 이것은 '+000000120' 형식으로 필드를 인쇄하는 데 사용됩니다. 이 정렬 옵션은 `complex`를 제외한 숫자 형에게만 유효합니다. 이것은 필드 너비 바로 앞에 '0' 이 있으면 숫자에 대한 기본값이 됩니다.
'^'	사용 가능한 공간 내에서 필드를 가운데에 배치합니다.

최소 필드 너비가 정의되지 않으면, 필드 너비는 항상 필드를 채울 데이터와 같은 크기이므로, 정렬 옵션은 이 경우 의미가 없습니다.

sign 옵션은 숫자 형에게만 유효하며, 다음 중 하나일 수 있습니다:

옵션	의미
'+'	음수뿐만 아니라 양수에도 부호를 사용해야 함을 나타냅니다.
'-'	음수에 대해서만 부호를 사용해야 함을 나타냅니다 (이것이 기본 동작입니다).
스페이스	양수에는 선행 스페이스를 사용하고, 음수에는 마이너스 부호를 사용해야 함을 나타냅니다.

The `'z'` option coerces negative zero floating-point values to positive zero after rounding to the format precision. This option is only valid for floating-point presentation types.

버전 3.11에서 변경: `'z'` 옵션을 추가했습니다 (**PEP 682** 도 보세요).

`'#'` 옵션은 변환에 "대안 형식" 이 사용되도록 만듭니다. 대안 형식은 형별로 다르게 정의됩니다. 이 옵션은 정수, 실수와 복소수 형에게만 유효합니다. 정수의 경우, 이진수, 8진수 또는 16진수 출력이 사용될 때, 이 옵션은 출력값에 각각 접두사 `'0b'`, `'0o'`, `'0x'` 또는 `'0X'` 를 추가합니다. 실수와 복소수의 경우, 대안 형식은 변환 결과의 소수점 아래 숫자가 없어도 항상 소수점 문자가 포함되게 합니다. 보통은, 소수점 문자는 그 뒤에 숫자가 있는 경우에만 변환 결과에 나타납니다. 이에 더해, `'g'` 및 `'G'` 변환의 경우 끝에 붙는 0이 결과에서 제거되지 않습니다.

width 는 최소 총 필드 너비를 정의하는 십진 정수인데, 접두사, 구분자 및 다른 포매팅 문자들을 포함합니다. 지정하지 않으면, 필드 너비는 내용에 의해 결정됩니다.

명시적 정렬이 주어지지 않을 때, *width* 필드 앞에 `'0'` 문자를 붙이면 `complex`를 제외한 숫자 형에 대해 부호를 고려하는 0 채움을 사용할 수 있습니다. 이것은 `'0'` 의 *fill* 문자와 `'='` 의 *alignment* 유형을 갖는 것과 동등합니다.

버전 3.10에서 변경: Preceding the *width* field by `'0'` no longer affects the default alignment for strings.

precision 은 표시 유형 `'f'`와 `'F'`의 경우 소수점 뒤에 표시될 자릿수를, `'g'`와 `'G'`의 경우 소수점 앞, 뒤로 표시할 자릿수를 나타내는 십진 정수입니다. 문자열 표시 유형의 경우 필드는 최대 필드 크기를 나타냅니

다 - 즉, 필드 내용에서 몇 개의 문자가 사용되는지 나타냅니다. 정수 표시 유형에는 *precision* 이 허용되지 않습니다.

The *grouping* option after *width* and *precision* fields specifies a digit group separator for the integral and fractional parts of a number respectively. It can be one of the following:

옵션	의미
`','`	정수 표시 유형 `'d'`와, `'n'`을 제외한 부동 소수점 표시 유형에 대해 3자리마다 쉼표를 삽입합니다. 다른 표시 유형에는 이 옵션이 지원되지 않습니다.
`'_'`	정수 표시 유형 `'d'`와, `'n'`을 제외한 부동 소수점 표시 유형에 대해 3자리마다 밑줄을 삽입합니다. 정수 표시 유형 `'b'`, `'o'`, `'x'` 및 `'X'` 의 경우 밑줄이 4자리마다 삽입됩니다. 다른 표시 유형에는 이 옵션이 지원되지 않습니다.

로케일을 고려하는 구분자의 경우, 대신 `'n'` 표시 유형을 사용하십시오.

버전 3.1에서 변경: `','` 옵션을 추가했습니다 (**PEP 378** 도 보세요).

버전 3.6에서 변경: `'_'` 옵션을 추가했습니다 (**PEP 515** 도 보세요).

버전 3.14에서 변경: Support the *grouping* option for the fractional part.

마지막으로 *type* 은 데이터를 표시하는 방법을 결정합니다.

사용 가능한 문자열 표시 유형은 다음과 같습니다:

유형	의미
`'s'`	문자열 포맷. 이것은 문자열의 기본 유형이고 생략될 수 있습니다.
없음	`'s'` 와 같습니다.

사용 가능한 정수 표시 유형은 다음과 같습니다:

유형	의미
`'b'`	이진 형식. 이진법으로 숫자를 출력합니다.
`'c'`	문자. 인쇄하기 전에 정수를 해당 유니코드 문자로 변환합니다.
`'d'`	십진 정수. 십진법으로 숫자를 출력합니다.
`'o'`	8진 형식. 8진법으로 숫자를 출력합니다.
`'x'`	16진 형식. 9보다 큰 숫자의 경우 소문자를 사용하여 16진법으로 숫자를 출력합니다.
`'X'`	16진 형식. 9보다 큰 숫자의 경우 대문자를 사용하여 16진법으로 숫자를 출력합니다. `'#'`가 지정되면, 접두사 `'0x'`도 대문자로 변환되어 `'0X'`로 표시됩니다.
`'n'`	숫자. 이는 현재 로케일 설정을 사용하여 적절한 숫자 그룹 구분자를 삽입한다는 점을 제외하고는 `'d'` 와 같습니다.
없음	`'d'` 와 같습니다.

위의 표시 유형에 더해, 정수는 아래에 나열된 부동 소수점 표시 유형으로 포맷될 수 있습니다 (`'n'` 및 없음 제외). 그렇게 할 때, 포매팅 전에 정수를 부동 소수점 숫자로 변환하기 위해 `float()` 가 사용됩니다.

`float` 및 `Decimal` 값에 사용할 수 있는 표시 유형은 다음과 같습니다:

유형	의미
'e'	과학적 표기법. 주어진 정밀도 p에 대해, 지수에서 계수를 구분하는 문자 'e' 를 사용하여 과학적 표기법으로 숫자를 포맷합니다. 계수는 소수점 앞의 한 자리와 소수점 뒤에 p 자리를 가져서, 총 p + 1 유효 자릿수를 갖습니다. 정밀도가 지정되지 않으면, *float*의 경우는 소수점 뒤에 6 숫자의 정밀도를 사용하고, *Decimal*의 경우는 모든 계수 숫자를 표시합니다. p=0 이면, # 옵션을 사용하지 않는 한 소수점은 생략됩니다.
'E'	과학적 표기법. 구분 문자로 대문자 'E' 를 사용한다는 것을 제외하고 'e' 와 같습니다.
'f'	고정 소수점 표기법. 주어진 정밀도 p에 대해, 소수점 뒤에 정확히 p 자리가 있는 십진수로 숫자를 포맷합니다. 정밀도가 지정되지 않으면, *float*의 경우는 소수점 뒤에 6 숫자의 정밀도를 사용하고, *Decimal*의 경우는 모든 계수 숫자를 표시할 만큼 충분히 큰 정밀도를 사용합니다. p=0 이면 , # 옵션을 사용하지 않는 한 소수점은 생략됩니다.
'F'	고정 소수점 표기법. 'f' 와 같지만, nan 을 NAN 으로, inf 를 INF 로 변환합니다.
'g'	범용 형식. 주어진 정밀도 p >= 1 에 대해, 숫자를 유효 숫자 p 로 자리 올림 한 다음, 결과를 크기에 따라 고정 소수점 형식이나 과학 표기법으로 포맷합니다. 정밀도 0은 정밀도 1과 동등하게 처리됩니다. 정확한 규칙은 다음과 같습니다: 표시 유형 'e' 와 정밀도 p-1 로 포맷된 결과의 지수가 exp 라고 가정하십시오. 이때 -m <= exp < p 이면 (여기서 m은 float에서 -4이고 *Decimal* 이면 -6입니다), 숫자는 표시 형식 'f' 와 정밀도 p-1-exp 로 포맷됩니다. 그렇지 않으면, 숫자는 표시 유형 'e' 와 정밀도 p-1 로 포맷됩니다. 두 경우 유효하지 않은 후행 0은 모두 유효숫자부에서 제거되고, 뒤에 남아있는 숫자가 없다면 '#' 옵션이 사용되지 않는 한 소수점도 제거됩니다. 정밀도가 지정되지 않으면, *float*의 경우 6 유효 자릿수의 정밀도를 사용합니다. *Decimal*의 경우, 결과 계수는 값의 계수 숫자로 구성됩니다; 과학적 표기법은 절댓값이 1e-6 보다 작은 값과 최하위 숫자의 자릿값이 1보다 큰 값에 사용되며, 그렇지 않으면 고정 소수점 표기법이 사용됩니다. 양과 음의 무한대, 양과 음의 0, nans는 정밀도와 무관하게 각각 inf, -inf, 0, -0, nan 으로 포맷됩니다.
'G'	범용 형식. 숫자가 너무 커지면 'E' 로 전환하는 것을 제외하고 'g' 와 같습니다. 무한과 NaN 의 표현도 대문자로 바뀝니다.
'n'	숫자. 현재 로케일 설정을 사용하여 숫자의 정수부에 적절한 숫자 그룹 구분자를 삽입한다는 점을 제외하면 'g' 와 같습니다.
'%'	백분율. 숫자에 100을 곱해서 고정 ('f') 형식으로 표시한 다음 백분율 기호를 붙입니다.
없음	*float*의 경우, 'g' 유형과 유사하지만, 고정 소수점 표기법이 결과를 포맷하는데 사용될 때, 소수점 이하로 적어도 하나의 숫자를 항상 포함하고, exp >= p - 1일 때 과학적 표기법으로 전환합니다. 정밀도가 지정되지 않으면, 후자는 주어진 값을 충실히 표현하는 데 필요한 만큼 크게 설정됩니다. *Decimal*의 경우, 현재 십진 컨텍스트의 context.capitals 값에 따라 'g'나 'G'와 같습니다. 전체적인 효과는 *str()* 의 출력을 다른 포맷 수정자에 의해 변경된 것처럼 만드는 것입니다.

The result should be correctly rounded to a given precision p of digits after the decimal point. The rounding mode for *float* matches that of the *round()* builtin. For *Decimal*, the rounding mode of the current *context* will be used.

The available presentation types for *complex* are the same as those for *float* ('%' is not allowed). Both the real and imaginary components of a complex number are formatted as floating-point numbers, according to the specified presentation type. They are separated by the mandatory sign of the imaginary part, the latter being terminated by a j suffix. If the presentation type is missing, the result will match the output of *str()* (complex numbers with a non-zero real part are also surrounded by parentheses), possibly altered by other format modifiers.

포맷 예제

이 절은 *str.format()* 문법의 예와 예전 %-포매팅과의 비교를 포함합니다.

대부분은 문법이 예전의 %-포매팅과 유사하며, {} 가 추가되고 % 대신 및 : 이 사용됩니다. 예를 들어, '%03.2f' 는 '{:03.2f}' 로 번역될 수 있습니다.

새 포맷 문법은 다음 예제에 보이는 것과 같이 새롭고 다양한 옵션도 지원합니다.

위치로 인자 액세스:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ 에서만
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # 인자 시퀀스를 언패킹
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # 인자의 인덱스는 반복할 수 있습니다
'abracadabra'
```

이름으로 인자 액세스:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-
↪115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

인자의 어트리뷰트 액세스:

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part␣
↪-5.0.'
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

인자의 항목 액세스:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

%s 과 %r 대체:

```
>>> "repr() 은 따옴표를 표시하지만: {!r}; str() 은 그렇지 않습니다: {!s}".format(
↪'test1', 'test2')
"repr() 은 따옴표를 표시하지만: 'test1'; str() 은 그렇지 않습니다: test2"
```

텍스트 정렬과 너비 지정:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # 채움 문자로 '*' 를 사용합니다
'***********centered***********'
```

%+f, %-f, % f 대체와 부호 지정:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # 항상 표시합니다
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # 양수는 스페이스를 표시합니다
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # 음수만 표시합니다 -- '{:f}; {:f}' 와 같습니
다
'3.140000; -3.140000'
```

%x, %o 대체와 다른 진법으로 값 변환:

```
>>> # format 은 이진수도 지원합니다
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

쉼표나 밑줄을 숫자 그룹 구분자로 사용:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
>>> '{:_}'.format(1234567890)
'1_234_567_890'
>>> '{:_b}'.format(1234567890)
'100_1001_1001_0110_0000_0010_1101_0010'
>>> '{:_x}'.format(1234567890)
'4996_02d2'
>>> '{:_}'.format(123456789.123456789)
'123_456_789.12345679'
>>> '{:.,}'.format(123456789.123456789)
'123456789.123,456,79'
>>> '{:,._}'.format(123456789.123456789)
'123,456,789.123_456_79'
```

백분율 표현:

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

형별 포매팅 사용:

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

인자 중첩과 보다 복잡한 예제:

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
...     print()
...
    5     5     5   101
    6     6     6   110
    7     7     7   111
    8     8    10  1000
    9     9    11  1001
   10     A    12  1010
   11     B    13  1011
```

6.1.4 Template strings ($-strings)

> **참고**
>
> The feature described here was introduced in Python 2.4; a simple templating method based upon regular expressions. It predates `str.format()`, formatted string literals, and *template string literals*.
>
> It is unrelated to template string literals (t-strings), which were introduced in Python 3.14. These evaluate to `string.templatelib.Template` objects, found in the `string.templatelib` module.

템플릿 문자열은 **PEP 292** 에 설명된 대로 더 간단한 문자열 치환을 제공합니다. 템플릿 문자열의 주요 사례는 국제화(i18n) 입니다. 이 문맥에서, 더 간단한 문법과 기능은 파이썬의 다른 내장 문자열 포매팅 기능보다 번역하기 쉽게 만들기 때문입니다. i18n을 위해 템플릿 문자열을 기반으로 구축된 라이브러리의 예는 flufl.i18n 패키지를 보십시오.

템플릿 문자열은 다음 규칙을 사용하여 `$`-기반 치환을 지원합니다:

- `$$` 는 이스케이프입니다. 이것은 하나의 `$` 로 치환됩니다.
- `$identifier` 는 매핑 키 `"identifier"` 와 일치하는 치환 자리 표시자를 지정합니다. 기본적으로, `"identifier"` 는 밑줄이나 ASCII 알파벳으로 시작하는 대소문자 구분 없는 ASCII 영숫자(밑줄 포함) 문자열로 제한됩니다. `$` 문자 뒤의 첫 번째 비 식별자 문자는 이 자리 표시자 명세를 종료합니다.
- `${identifier}` 는 `$identifier` 와 동등합니다. 유효한 식별자 문자가 자리 표시자 뒤에 오지만, 자리 표시자의 일부가 아니면 필요합니다, 가령 `"${noun}ification"`.

문자열에 다른 방식으로 `$` 이 등장하면 `ValueError` 가 발생합니다.

`string` 모듈은 이 규칙들을 구현하는 `Template` 클래스를 제공합니다. `Template` 의 메서드는 다음과 같습니다:

class string.**Template**(*template*)

생성자는 템플릿 문자열 하나를 받아들입니다.

substitute(*mapping*={}, /, **kwds*)

템플릿 치환을 수행하고, 새 문자열을 반환합니다. *mapping* 은 템플릿의 자리 표시자와 일치하는 키를 가진 임의의 딕셔너리류 객체입니다. 또는, 키워드가 자리 표시자인 키워드 인자를 제공할 수 있습니다. *mapping* 및 *kwds* 가 모두 제공되고 중복이 있는 경우, *kwds* 의 자리 표시자가 우선합니다.

safe_substitute(*mapping*={}, /, **kwds*)

substitute() 와 비슷하지만, *mapping* 과 *kwds* 에 자리 표시자가 없는 경우, *KeyError* 예외를 발생시키지 않고 원래 자리 표시자가 결과 문자열에 그대로 나타납니다. 또한 *substitute()* 와는 달리, `$` 가 잘못 사용되는 경우 *ValueError* 를 일으키는 대신 단순히 `$` 를 반환합니다.

다른 예외가 여전히 발생할 수 있지만, 이 메서드가 항상 예외를 발생시키는 대신 사용 가능한 문자열을 반환하려고 시도하기 때문에 "안전(safe)" 하다고 합니다. 다른 의미에서, *safe_substitute()* 는 안전하다고 할 수 없습니다. 길 잃은(dangling) 구분 기호, 쌍을 이루지 않는 중괄호, 유효한 파이썬 식별자가 아닌 자리 표시자를 포함하는 잘못된 템플릿을 조용히 무시하기 때문입니다.

is_valid()

Returns `False` if the template has invalid placeholders that will cause *substitute()* to raise *ValueError*.

Added in version 3.11.

get_identifiers()

Returns a list of the valid identifiers in the template, in the order they first appear, ignoring any invalid identifiers.

Added in version 3.11.

Template 인스턴스는 공개 데이터 어트리뷰트도 하나 제공합니다:

template

이것은 생성자의 *template* 인자로 전달된 객체입니다. 일반적으로, 변경해서는 안 되지만, 읽기 전용 액세스가 강제되지는 않습니다.

다음은 Template 사용 방법의 예입니다:

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

고급 사용법: *Template* 의 서브 클래스를 파생하여, 자리 표시자 문법, 구분 기호 문자 또는 템플릿 문자열을 파싱하는데 사용되는 전체 정규식을 사용자 정의 할 수 있습니다. 이렇게 하려면, 다음 클래스 어트리뷰트를 재정의할 수 있습니다:

- *delimiter* – 자리 표시자를 도입하는 구분자를 나타내는 리터럴 문자열입니다. 기본값은 `$` 입니다. 구현체는 필요할 때 이 문자열에 *re.escape()* 를 호출하므로, 이 문자열은 정규식이 아니어야 합니

다. 또한, 클래스 생성 후에 구분자를 변경할 수 없습니다 (즉, 다른 구분자는 반드시 서브 클래스의 클래스 이름 공간에 설정해야 합니다).

- *idpattern* – 중괄호로 둘러싸지 않은 자리 표시자의 패턴을 설명하는 정규식입니다. 기본값은 정규식 `(?a:[_a-z][_a-z0-9]*)` 입니다. *braceidpattern* 이 `None` 인 경우, 이 패턴은 중괄호가 있는 자리 표시자에게도 적용됩니다.

> **참고**
>
> 기본 *flags* 가 `re.IGNORECASE` 이기 때문에, 패턴 `[a-z]` 는 비 ASCII 문자와 일치 할 수 있습니다. 이 때문에 정규식에 `a` 플래그를 사용했습니다.

  버전 3.7에서 변경: *braceidpattern* 은 중괄호로 싸여있을 때와 그렇지 않을 때 사용되는 별도의 패턴을 정의하는데 사용할 수 있습니다.

- *braceidpattern* – *idpattern* 과 유사하지만, 중괄호로 싸인 자리 표시자에 대한 패턴을 설명합니다. 기본값은 `None` 인데, *idpattern* 을 사용하는 것을 의미합니다 (즉, 같은 패턴이 중괄호가 있을 때와 없을 때 모두 사용됩니다). 이 값을 주면, 중괄호가 있을 때와 없을 때의 자리 표시자에 서로 다른 패턴을 정의할 수 있습니다.

  Added in version 3.7.

- *flags* – 치환 인식에 사용되는 정규식을 컴파일할 때 적용될 정규식 플래그입니다. 기본값은 `re.IGNORECASE` 입니다. `re.VERBOSE` 가 항상 플래그에 추가되므로, 사용자 정의 *idpattern* 은 상세한 정규식의 규칙을 따라야 합니다.

  Added in version 3.2.

또는, 클래스 어트리뷰트 *pattern* 을 재정의하여 전체 정규식 패턴을 제공 할 수 있습니다. 이렇게 하는 경우, 값은 네 개의 이름있는 캡처 그룹이 있는 정규식 객체여야 합니다. 캡처 그룹은 위에 제공된 규칙과 함께 유효하지 않은 자리 표시자 규칙에 해당합니다:

- *escaped* – 이 그룹은 이스케이프 시퀀스를 일치시킵니다, 예를 들어 기본 패턴에서 `$$`.
- *named* – 이 그룹은 중괄호가 없는 자리 표시자 이름을 일치합니다; 캡처 그룹에 구분자를 포함해서는 안 됩니다.
- *braced* – 이 그룹은 중괄호로 묶인 자리 표시자 이름을 일치시킵니다; 캡처 그룹에 구분자나 중괄호를 포함해서는 안 됩니다.
- *invalid* – 이 그룹은 그 외의 구분자 패턴(일반적으로 단일 구분자)을 일치시키고, 정규식의 마지막에 나타나야 합니다.

The methods on this class will raise `ValueError` if the pattern matches the template without one of these named groups matching.

6.1.5 도움 함수

`string.`**`capwords`**(*s*, *sep=None*)

인자를 `str.split()` 을 사용하여 단어로 나누고, `str.capitalize()` 를 사용하여 각 단어의 첫 글자를 대문자로 만들고, 이렇게 만들어진 단어들을 `str.join()` 을 사용하여 결합합니다. 선택적 두 번째 인자 *sep* 가 없거나 `None` 이면, 연속된 공백 문자는 단일 스페이스로 바뀌고 앞뒤 공백이 제거됩니다. 그렇지 않으면 *sep* 가 단어를 나누고 합치는 데 사용됩니다.

6.2 `string.templatelib` — Support for template string literals

Source code: Lib/string/templatelib.py

더 보기

- Format strings
- Template string literal (t-string) syntax
- **PEP 750**

6.2.1 Template strings

Added in version 3.14.

Template strings are a mechanism for custom string processing. They have the full flexibility of Python' s f-strings, but return a *Template* instance that gives access to the static and interpolated (in curly braces) parts of a string *before* they are combined.

To write a t-string, use a `'t'` prefix instead of an `'f'`, like so:

```
>>> pi = 3.14
>>> t't-strings are new in Python {pi!s}!'
Template(
   strings=('t-strings are new in Python ', '!'),
   interpolations=(Interpolation(3.14, 'pi', 's', ''),)
)
```

6.2.2 Types

class `string.templatelib.`**`Template`**

The `Template` class describes the contents of a template string. It is immutable, meaning that attributes of a template cannot be reassigned.

The most common way to create a `Template` instance is to use the template string literal syntax. This syntax is identical to that of f-strings, except that it uses a `t` prefix in place of an `f`:

```
>>> cheese = 'Red Leicester'
>>> template = t"We're fresh out of {cheese}, sir."
>>> type(template)
<class 'string.templatelib.Template'>
```

Templates are stored as sequences of literal *strings* and dynamic *interpolations*. A *values* attribute holds the values of the interpolations:

```
>>> cheese = 'Camembert'
>>> template = t'Ah! We do have {cheese}.'
>>> template.strings
('Ah! We do have ', '.')
>>> template.interpolations
(Interpolation('Camembert', ...),)
>>> template.values
('Camembert',)
```

The `strings` tuple has one more element than `interpolations` and `values`; the interpolations "belong" between the strings. This may be easier to understand when tuples are aligned

```
template.strings:  ('Ah! We do have ',              '.')
template.values:   (                  'Camembert',     )
```

Attributes

strings: *tuple*[*str*, ...]

A *tuple* of the static strings in the template.

```
>>> cheese = 'Camembert'
>>> template = t'Ah! We do have {cheese}.'
>>> template.strings
('Ah! We do have ', '.')
```

Empty strings *are* included in the tuple:

```
>>> response = 'We do have '
>>> cheese = 'Camembert'
>>> template = t'Ah! {response}{cheese}.'
>>> template.strings
('Ah! ', '', '.')
```

The `strings` tuple is never empty, and always contains one more string than the `interpolations` and `values` tuples:

```
>>> t''.strings
('',)
>>> t''.values
()
>>> t'{'cheese'}'.strings
('', '')
>>> t'{'cheese'}'.values
('cheese',)
```

interpolations: *tuple*[*Interpolation*, ...]

A *tuple* of the interpolations in the template.

```
>>> cheese = 'Camembert'
>>> template = t'Ah! We do have {cheese}.'
>>> template.interpolations
(Interpolation('Camembert', 'cheese', None, ''),)
```

The `interpolations` tuple may be empty and always contains one fewer values than the `strings` tuple:

```
>>> t'Red Leicester'.interpolations
()
```

values: *tuple*[*object*, ...]

A tuple of all interpolated values in the template.

```
>>> cheese = 'Camembert'
>>> template = t'Ah! We do have {cheese}.'
>>> template.values
('Camembert',)
```

The `values` tuple always has the same length as the `interpolations` tuple. It is always equivalent to `tuple(i.value for i in template.interpolations)`.

Methods

__new__(**args:* str | Interpolation)

While literal syntax is the most common way to create a `Template`, it is also possible to create them directly using the constructor:

```
>>> from string.templatelib import Interpolation, Template
>>> cheese = 'Camembert'
>>> template = Template(
...     'Ah! We do have ', Interpolation(cheese, 'cheese'), '.'
... )
>>> list(template)
['Ah! We do have ', Interpolation('Camembert', 'cheese', None, ''), '.']
```

If multiple strings are passed consecutively, they will be concatenated into a single value in the *strings* attribute. For example, the following code creates a *Template* with a single final string:

```
>>> from string.templatelib import Template
>>> template = Template('Ah! We do have ', 'Camembert', '.')
>>> template.strings
('Ah! We do have Camembert.',)
```

If multiple interpolations are passed consecutively, they will be treated as separate interpolations and an empty string will be inserted between them. For example, the following code creates a template with empty placeholders in the *strings* attribute:

```
>>> from string.templatelib import Interpolation, Template
>>> template = Template(
...     Interpolation('Camembert', 'cheese'),
...     Interpolation('.', 'punctuation'),
... )
>>> template.strings
('', '', '')
```

iter(template)

Iterate over the template, yielding each non-empty string and *Interpolation* in the correct order:

```
>>> cheese = 'Camembert'
>>> list(t'Ah! We do have {cheese}.')
['Ah! We do have ', Interpolation('Camembert', 'cheese', None, ''), '.']
```

조심

Empty strings are **not** included in the iteration:

```
>>> response = 'We do have '
>>> cheese = 'Camembert'
>>> list(t'Ah! {response}{cheese}.')
['Ah! ',
 Interpolation('We do have ', 'response', None, ''),
 Interpolation('Camembert', 'cheese', None, ''),
 '.']
```

template + other

template += other

Concatenate this template with another, returning a new `Template` instance:

```
>>> cheese = 'Camembert'
>>> list(t'Ah! ' + t'We do have {cheese}.')
['Ah! We do have ', Interpolation('Camembert', 'cheese', None, ''), '.']
```

Concatenating a `Template` and a `str` is **not** supported. This is because it is unclear whether the string should be treated as a static string or an interpolation. If you want to concatenate a `Template` with a string, you should either wrap the string directly in a `Template` (to treat it as a static string) or use an `Interpolation` (to treat it as dynamic):

```
>>> from string.templatelib import Interpolation, Template
>>> template = t'Ah! '
>>> # Treat 'We do have ' as a static string
>>> template += Template('We do have ')
>>> # Treat cheese as an interpolation
>>> cheese = 'Camembert'
>>> template += Template(Interpolation(cheese, 'cheese'))
>>> list(template)
['Ah! We do have ', Interpolation('Camembert', 'cheese', None, '')]
```

class `string.templatelib.`**`Interpolation`**

The `Interpolation` type represents an expression inside a template string. It is immutable, meaning that attributes of an interpolation cannot be reassigned.

Interpolations support pattern matching, allowing you to match against their attributes with the match statement:

```
>>> from string.templatelib import Interpolation
>>> interpolation = t'{1. + 2.:.2f}'.interpolations[0]
>>> interpolation
Interpolation(3.0, '1. + 2.', None, '.2f')
>>> match interpolation:
...     case Interpolation(value, expression, conversion, format_spec):
...         print(value, expression, conversion, format_spec, sep=' | ')
...
3.0 | 1. + 2. | None | .2f
```

Attributes

`value`: *`object`*

The evaluated value of the interpolation.

```
>>> t'{1 + 2}'.interpolations[0].value
3
```

`expression`: *`str`*

The text of a valid Python expression, or an empty string.

The *`expression`* is the original text of the interpolation' s Python expression, if the interpolation was created from a t-string literal. Developers creating interpolations manually should either set this to an empty string or choose a suitable valid Python expression.

```
>>> t'{1 + 2}'.interpolations[0].expression
'1 + 2'
```

`conversion`: *`Literal`*`['a', 'r', 's'] | `*`None`*

The conversion to apply to the value, or `None`.

The `conversion` is the optional conversion to apply to the value:

```
>>> t'{1 + 2!a}'.interpolations[0].conversion
'a'
```

> **참고**
>
> Unlike f-strings, where conversions are applied automatically, the expected behavior with t-strings is that code that *processes* the `Template` will decide how to interpret and whether to apply the `conversion`. For convenience, the `convert()` function can be used to mimic f-string conversion semantics.

format_spec: *str*

The format specification to apply to the value.

The `format_spec` is an optional, arbitrary string used as the format specification to present the value:

```
>>> t'{1 + 2:.2f}'.interpolations[0].format_spec
'.2f'
```

> **참고**
>
> Unlike f-strings, where format specifications are applied automatically via the `format()` protocol, the expected behavior with t-strings is that code that *processes* the interpolation will decide how to interpret and whether to apply the format specification. As a result, `format_spec` values in interpolations can be arbitrary strings, including those that do not conform to the `format()` protocol.

Methods

__new__(*value:* object, *expression:* str, *conversion:* Literal['a', 'r', 's'] | None = *None*, *format_spec:* str = '')

Create a new `Interpolation` object from component parts.

매개변수

- **value** – The evaluated, in-scope result of the interpolation.
- **expression** – The text of a valid Python expression, or an empty string.
- **conversion** – The *conversion* to be used, one of `None`, `'a'`, `'r'`, or `'s'`.
- **format_spec** – An optional, arbitrary string used as the *format specification* to present the value.

6.2.3 Helper functions

string.templatelib.**convert**(*obj*, /, *conversion*)

Applies formatted string literal *conversion* semantics to the given object *obj*. This is frequently useful for custom template string processing logic.

Three conversion flags are currently supported:

- `'s'` which calls `str()` on the value (like `!s`),
- `'r'` which calls `repr()` (like `!r`), and
- `'a'` which calls `ascii()` (like `!a`).

If the conversion flag is `None`, *obj* is returned unchanged.

6.3 re — Regular expression operations

Source code: Lib/re/

이 모듈은 Perl에 있는 것과 유사한 정규식 일치 연산을 제공합니다.

Both patterns and strings to be searched can be Unicode strings (*str*) as well as 8-bit strings (*bytes*). However, Unicode strings and 8-bit strings cannot be mixed: that is, you cannot match a Unicode string with a bytes pattern or vice-versa; similarly, when asking for a substitution, the replacement string must be of the same type as both the pattern and the search string.

Regular expressions use the backslash character (`'\'`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python' s usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write `'\\\\'` as the pattern string, because the regular expression must be `\\`, and each backslash must be expressed as `\\` inside a regular Python string literal. Also, please note that any invalid escape sequences in Python' s usage of the backslash in string literals now generate a *SyntaxWarning* and in the future this will become a *SyntaxError*. This behaviour will happen even if it is a valid escape sequence for a regular expression.

해결책은 정규식 패턴에 파이썬의 날 문자열(raw string) 표기법을 사용하는 것입니다; 역 슬래시는 `'r'` 접두어가 붙은 문자열 리터럴에서 특별한 방법으로 처리되지 않습니다. 따라서 `r"\n"`은 `'\'`와 `'n'`을 포함하는 두 글자 문자열이고, `"\n"`은 개행을 포함하는 한 글자 문자열입니다. 일반적으로 패턴은, 이 날 문자열 표기법을 사용하여 파이썬 코드로 표현됩니다.

대부분 정규식 연산은 모듈 수준 함수와 컴파일된 정규식의 메서드로 사용할 수 있다는 점에 유의해야 합니다. 함수는 정규식 객체를 먼저 컴파일할 필요가 없도록 하는 바로 가기이지만, 일부 미세 조정 매개 변수가 빠져있습니다.

> **더 보기**
>
> The third-party regex module, which has an API compatible with the standard library *re* module, but offers additional functionality and a more thorough Unicode support.

6.3.1 정규식 문법

정규식(또는 RE)은 일치하는 문자열 집합을 지정합니다; 이 모듈의 함수는 특정 문자열이 주어진 정규식과 일치하는지 확인할 수 있도록 합니다 (또는 주어진 정규식이 특정 문자열과 일치하는지, 결국 같은 결과를 줍니다).

정규식을 이어붙여서 새로운 정규식을 만들 수 있습니다; A와 B가 모두 정규식이면 AB도 정규식입니다. 일반적으로 문자열 p가 A와 일치하고 다른 문자열 q가 B와 일치하면 문자열 pq가 AB와 일치합니다. 이것은 A나 B가 우선순위가 낮은 연산, A와 B 사이의 경계 조건 또는 숫자 그룹 참조를 포함하지 않는 한 성립합니다. 따라서, 복잡한 정규식은 여기에 설명된 것과 같은 더 단순한 기본 정규식으로 쉽게 구성할 수 있습니다. 정규식의 이론과 구현에 관한 자세한 내용은 Friedl 책 [Frie09], 또는 컴파일러 작성에 관한 거의 모든 교과서를 참조하십시오.

정규식의 형식에 대한 간단한 설명이 이어집니다. 더 자세한 정보와 더 친절한 소개는 regex-howto를 참조하십시오.

정규식은 특수 문자와 일반 문자를 모두 포함 할 수 있습니다. `'A'`, `'a'` 또는 `'0'`과 같은 대부분의 일반 문자는 가장 단순한 정규식입니다; 그들은 단순히 자신과 일치합니다. 일반 문자를 이어 붙일 수 있어서, `last`는 `'last'` 문자열과 일치합니다. (이 절의 나머지 부분에서는, RE를 (보통 따옴표 없이) 이런 특별한 스타일로, 일치할 문자열은 '작은따옴표 안에' 씁니다.)

`'|'`나 `'('`와 같은 일부 문자는 특수합니다. 특수 문자는 일반 문자의 클래스를 나타내거나, 그 주변의 정규식이 해석되는 방식에 영향을 줍니다.

Repetition operators or quantifiers (`*`, `+`, `?`, `{m,n}`, etc) cannot be directly nested. This avoids ambiguity with the non-greedy modifier suffix `?`, and with other modifiers in other implementations. To apply a second repetition to an inner repetition, parentheses may be used. For example, the expression `(?:a{6})*` matches any multiple of six `'a'` characters.

특수 문자는 다음과 같습니다:

.

(Dot.) In the default mode, this matches any character except a newline. If the *DOTALL* flag has been specified, this matches any character including a newline. `(?s:.)` matches any character regardless of flags.

^

(캐럿.) 문자열의 시작과 일치하고, *MULTILINE* 모드에서는 각 개행 직후에도 일치합니다.

$

문자열의 끝이나 문자열 끝의 개행 문자 바로 직전과 일치하고, *MULTILINE* 모드에서는 개행 문자 앞에서도 일치합니다. `foo`는 'foo' 와 'foobar' 를 모두 일치시키는 반면, 정규식 `foo$`는 'foo' 만 일치합니다. 흥미롭게도, `'foo1\nfoo2\n'`에서 `foo.$`를 검색하면 'foo2' 는 정상적으로 일치되지만, 'foo1' 은 *MULTILINE* 모드에서 검색됩니다; `'foo\n'`에서 단일 `$`를 검색하면 두 개의 (빈) 일치가 발견됩니다: 하나는 개행 직전에, 다른 하나는 문자열 끝에.

*****

결과 RE가 선행 RE의 가능한 한 많은 0회 이상의 반복과 일치하도록 합니다. `ab*`는 'a' , 'ab' 또는 'a' 다음에 임의의 수의 'b' 가 오는 것과 일치합니다.

+

결과 RE가 선행 RE의 1회 이상의 반복과 일치하도록 합니다. `ab+`는 'a' 다음에 하나 이상의 'b' 가 오는 것과 일치합니다; 단지 'a' 와는 일치하지 않습니다.

?

결과 RE가 선행 RE의 0 또는 1 반복과 일치하도록 합니다. `ab?`는 'a' 나 'ab' 와 일치합니다.

***?, +?, ??**

The `'*'`, `'+'`, and `'?'` quantifiers are all *greedy*; they match as much text as possible. Sometimes this behaviour isn' t desired; if the RE `<.*>` is matched against `'<a> b <c>'`, it will match the entire string, and not just `'<a>'`. Adding `?` after the quantifier makes it perform the match in *non-greedy* or *minimal* fashion; as *few* characters as possible will be matched. Using the RE `<.*?>` will match only `'<a>'`.

***+, ++, ?+**

Like the `'*'`, `'+'`, and `'?'` quantifiers, those where `'+'` is appended also match as many times as possible. However, unlike the true greedy quantifiers, these do not allow back-tracking when the expression following it fails to match. These are known as *possessive* quantifiers. For example, `a*a` will match `'aaaa'` because the `a*` will match all 4 `'a'`s, but, when the final `'a'` is encountered, the expression is backtracked so that in the end the `a*` ends up matching 3 `'a'`s total, and the fourth `'a'` is matched by the final `'a'`. However, when `a*+a` is used to match `'aaaa'`, the `a*+` will match all 4 `'a'`, but when the final `'a'` fails to find any more characters to match, the expression cannot be backtracked and will thus fail to match. `x*+`, `x++` and `x?+` are equivalent to `(?>x*)`, `(?>x+)` and `(?>x?)` correspondingly.

Added in version 3.11.

{m}

선행 RE의 정확히 *m* 복사가 일치하도록 지정합니다; 적은 횟수의 일치는 전체 RE가 일치하지 않게 됩니다. 예를 들어, `a{6}`는 정확히 6개의 `'a'` 문자와 일치하지만, 5개의 문자와는 일치하지 않습니다.

{m,n}

결과 RE를 선행 RE의 *m*에서 *n* 사이의 최대한 많은 반복과 일치하도록 합니다. 예를 들어, `a{3,5}`는 3에서 5개의 `'a'` 문자와 일치합니다. *m*을 생략하면 하한값 0이 지정되고, *n*을 생략하면 무한한 상한이 지정됩니다. 예를 들어, `a{4,}b`는 `'aaaab'` 나 1000개의 `'a'` 문자와 `'b'`가 일치하지만, `'aaab'`는 일치하지 않습니다. 콤마는 생략할 수 없습니다, 그렇지 않으면 한정자가 앞에서 설명한 형식과 혼동될 수 있습니다.

{m,n}?

Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as *few* repetitions as possible. This is the non-greedy version of the previous quantifier. For example, on the 6-character string `'aaaaaa'`, `a{3,5}` will match 5 `'a'` characters, while `a{3,5}?` will only match 3 characters.

{m,n}+

Causes the resulting RE to match from *m* to *n* repetitions of the preceding RE, attempting to match as many repetitions as possible *without* establishing any backtracking points. This is the possessive version of the quantifier above. For example, on the 6-character string `'aaaaaa'`, `a{3,5}+aa` attempt to match 5 `'a'` characters,

then, requiring 2 more `'a'`s, will need more characters than available and thus fail, while `a{3,5}aa` will match with `a{3,5}` capturing 5, then 4 `'a'`s by backtracking and then the final 2 `'a'`s are matched by the final `aa` in the pattern. `x{m,n}+` is equivalent to `(?>x{m,n})`.

Added in version 3.11.

****

특수 문자를 이스케이프 하거나 (`'*'`, `'?'` 등의 문자를 일치시킬 수 있도록 합니다), 특수 시퀀스를 알립니다; 특수 시퀀스는 아래에서 설명합니다.

날 문자열을 사용하여 패턴을 표현하지 않는다면, 파이썬이 문자열 리터럴에서 이스케이프 시퀀스로 역 슬래시를 사용한다는 것을 기억하십시오; 이스케이프 시퀀스가 파이썬의 구문 분석기에 의해 인식되지 않으면, 역 슬래시와 후속 문자가 결과 문자열에 포함됩니다. 그러나 파이썬이 결과 시퀀스를 인식한다면 역 슬래시는 두 번 반복되어야 합니다. 이것은 복잡하고 이해하기 어렵기 때문에, 가장 단순한 표현 이외에는 날 문자열을 사용하는 것이 좋습니다.

[]

문자 집합을 나타내는 데 사용됩니다. 집합 안에서:

- 문자는 개별적으로 나열 할 수 있습니다, 예를 들어 `[amk]`는 `'a'`, `'m'` 또는 `'k'`와 일치합니다.
- 문자의 범위는 `'-'`로 구분된 두 문자를 주고는 것으로 나타낼 수 있습니다, 예를 들어 `[a-z]`는 모든 소문자 ASCII 문자와 일치하고, `[0-5][0-9]`는 `00`에서 `59`까지의 모든 두 자리 숫자와 일치하며, `[0-9A-Fa-f]`는 모든 16진수와 일치합니다. `-`가 이스케이프 처리되거나 (예를 들어 `[a\-z]`) 첫 번째나 마지막 문자로 배치되면 (예를 들어 `[-a]`나 `[a-]`) 리터럴 `'-'`와 일치합니다.
- Special characters except backslash lose their special meaning inside sets. For example, `[(+*)]` will match any of the literal characters `'('`, `'+'`, `'*'`, or `')'`.
- Backslash either escapes characters which have special meaning in a set such as `'-'`, `']'`, `'^'` and `'\\'` itself or signals a special sequence which represents a single character such as `\xa0` or `\n` or a character class such as `\w` or `\S` (defined below). Note that `\b` represents a single "backspace" character, not a word boundary as outside a set, and numeric escapes such as `\1` are always octal escapes, not group references. Special sequences which do not match a single character such as `\A` and `\z` are not allowed.
- 범위 내에 있지 않은 문자는 여집합(*complementing*)을 만들어 일치할 수 있습니다. 집합의 첫 번째 문자가 `'^'`이면, 집합에 속하지 않은 모든 문자가 일치합니다. 예를 들어, `[^5]`는 `'5'`를 제외한 모든 문자와 일치하며, `[^^]`는 `'^'`를 제외한 모든 문자와 일치합니다. `^`는 집합의 첫 번째 문자가 아닐 때 특별한 의미가 없습니다.
- To match a literal `']'` inside a set, precede it with a backslash, or place it at the beginning of the set. For example, both `[()[\]{}]` and `[]()[{}]` will match a right bracket, as well as left bracket, braces, and parentheses.
- 유니코드 기술 표준#18 에서 정의하는 중첩 집합과 집합 연산의 지원이 다음에 추가될 수 있습니다. 이것은 문법을 변경하므로, 이 변경을 용이하게 하기 위해 당분간 *FutureWarning*이 모호한 경우에 발생합니다. 여기에는 리터럴 `'['`로 시작하거나 리터럴 문자 시퀀스 `'--'`, `'&&'`, `'~~'` 및 `'||'`가 포함된 집합이 포함됩니다. 경고를 피하려면 역 슬래시로 이스케이프 처리하십시오.

버전 3.7에서 변경: 문자 집합이 미래에 의미가 변할 구조를 포함하고 있다면 *FutureWarning*이 발생합니다.

|

`A|B`(여기서 *A*와 *B*는 임의의 RE일 수 있습니다)는 *A*나 *B*와 일치하는 정규식을 만듭니다. 이러한 방식으로 임의의 수의 RE를 `'|'`로 분리 할 수 있습니다. 이것은 그룹(아래를 참조하세요)에서도 사용할 수 있습니다. 대상 문자열이 스캔 될 때 `'|'`로 구분된 RE는 왼쪽에서 오른쪽으로 시도됩니다. 한 패턴이 완전히 일치하면, 해당 분기가 받아들여집니다. 이는 일단 *A*가 일치하면, *B*는 전체적으로 더 긴 일치를 생성하더라도 더 검사되지 않는다는 것을 뜻합니다. 즉, `'|'` 연산자는 절대로 탐욕적이지 않습니다. 리터럴 `'|'`와 일치시키려면, `\|`를 사용하거나, `[|]`처럼 문자 클래스 안에 넣으십시오.

(...)

괄호 안에 있는 정규식과 일치하며, 그룹의 시작과 끝을 나타냅니다; 그룹의 내용은 일치가 수행된 후 조회할 수 있으며, 나중에 문자열에서 `\number` 특수 시퀀스로 일치시킬 수 있습니다 (아래에서 설명됩니다). 리터럴 `'('`나 `')'`를 일치시키려면, `\(`나 `\)`를 사용하거나, 문자 클래스 안에 넣으십시오: `[(], [)]`.

(?...)
이것은 확장 표기법입니다 (그렇지 않으면 '(' 다음에 오는 '?'``는 의미가 없습니다). ``'?' 다음의 첫 번째 문자는 확장의 의미와 이후의 문법을 결정합니다. 확장은 대개 새 그룹을 만들지 않습니다; 이 규칙에 대한 유일한 예외는 (?P<name>...) 입니다. 다음은 현재 지원되는 확장입니다.

(?aiLmsux)
(One or more letters from the set 'a', 'i', 'L', 'm', 's', 'u', 'x'.) The group matches the empty string; the letters set the corresponding flags for the entire regular expression:

- *re.A* (ASCII-only matching)
- *re.I* (ignore case)
- *re.L* (locale dependent)
- *re.M* (multi-line)
- *re.S* (dot matches all)
- *re.U* (Unicode matching)
- *re.X* (verbose)

(The flags are described in 모듈 내용.) This is useful if you wish to include the flags as part of the regular expression, instead of passing a *flag* argument to the *re.compile()* function. Flags should be used first in the expression string.

버전 3.11에서 변경: This construction can only be used at the start of the expression.

(?:...)
일반 괄호의 비 포착 버전. 괄호 안의 정규식과 일치하지만, 그룹과 일치하는 부분 문자열은 일치를 수행한 후 조회하거나 나중에 패턴에서 참조할 수 없습니다.

(?aiLmsux-imsx:...)
(Zero or more letters from the set 'a', 'i', 'L', 'm', 's', 'u', 'x', optionally followed by '-' followed by one or more letters from the 'i', 'm', 's', 'x'.) The letters set or remove the corresponding flags for the part of the expression:

- *re.A* (ASCII-only matching)
- *re.I* (ignore case)
- *re.L* (locale dependent)
- *re.M* (multi-line)
- *re.S* (dot matches all)
- *re.U* (Unicode matching)
- *re.X* (verbose)

(The flags are described in 모듈 내용.)

The letters 'a', 'L' and 'u' are mutually exclusive when used as inline flags, so they can' t be combined or follow '-'. Instead, when one of them appears in an inline group, it overrides the matching mode in the enclosing group. In Unicode patterns (?a:...) switches to ASCII-only matching, and (?u:...) switches to Unicode matching (default). In bytes patterns (?L:...) switches to locale dependent matching, and (?a:...) switches to ASCII-only matching (default). This override is only in effect for the narrow inline group, and the original matching mode is restored outside of the group.

Added in version 3.6.

버전 3.7에서 변경: 문자 'a', 'L' 및 'u'도 그룹에서 사용할 수 있습니다.

(?>...)
Attempts to match ... as if it was a separate regular expression, and if successful, continues to match the rest of the pattern following it. If the subsequent pattern fails to match, the stack can only be unwound to a point *before* the (?>...) because once exited, the expression, known as an *atomic group*, has thrown away all stack points within itself. Thus, (?>.*). would never match anything because first the .* would match

all characters possible, then, having nothing left to match, the final `.` would fail to match. Since there are no stack points saved in the Atomic Group, and there is no stack point before it, the entire expression would thus fail to match.

Added in version 3.11.

(?P<name>...)

Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name *name*. Group names must be valid Python identifiers, and in `bytes` patterns they can only contain bytes in the ASCII range. Each group name must be defined only once within a regular expression. A symbolic group is also a numbered group, just as if the group were not named.

이름 있는 그룹은 세 가지 문맥에서 참조될 수 있습니다. 패턴이 `(?P<quote>['"]).*?(?P=quote)` 면 (즉, 작은따옴표나 큰따옴표로 인용된 문자열과 일치):

그룹 "quote" 에 대한 참조 문맥	참조하는 방법
같은 패턴 자체에서	• `(?P=quote)` (보이는 대로) • `\1`
일치 객체 *m*을 처리할 때	• `m.group('quote')` • `m.end('quote')` (등)
`re.sub()` 의 *repl* 인자로 전달되는 문자열에서	• `\g<quote>` • `\g<1>` • `\1`

버전 3.12에서 변경: In `bytes` patterns, group *name* can only contain bytes in the ASCII range (`b'\x00'`-`b'\x7f'`).

(?P=name)

이름있는 그룹에 대한 역참조; *name*이라는 이름의 앞선 그룹과 일치하는 텍스트와 일치합니다.

(?#...)

주석; 괄호의 내용은 단순히 무시됩니다.

(?=...)

`...`가 다음과 일치하면 일치하지만, 문자열을 소비하지는 않습니다. 이를 미리 보기 어서션 (*lookahead assertion*) 이라고 합니다. 예를 들어, `Isaac (?=Asimov)` 는 `'Asimov'`가 뒤따를 때만 `'Isaac '`과 일치합니다.

(?!...)

`...`가 다음과 일치하지 않으면 일치합니다. 이것은 부정적인 미리 보기 어서션 (*negative lookahead assertion*) 입니다. 예를 들어, `Isaac (?!Asimov)` 는 `'Asimov'`가 뒤따르지 않을 때 만 `'Isaac '`과 일치합니다.

(?<=...)

문자열의 현재 위치 앞에 현재 위치에서 끝나는 `...`와의 일치가 있으면 일치합니다. 이를 긍정적인 되돌아보기 어서션 (*positive lookbehind assertion*) 이라고 합니다. 되돌아보기가 3문자를 백업하고 포함된 패턴이 일치하는지 확인하기 때문에 `(?<=abc)def`는 `'abcdef'`에서 일치를 찾습니다. 포함된 패턴은 고정 길이의 문자열과 일치해야 합니다, 즉, `abc`나 `a|b`는 허용되지만, `a*`와 `a{3,4}`는 허용되지 않습니다. 긍정적인 되돌아보기 어서션으로 시작하는 패턴은 검색되는 문자열의 시작 부분에서 일치하지 않음에 유의하십시오; `match()` 함수보다는 `search()` 함수를 사용하기를 원할 것입니다:

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

이 예에서는 하이픈 다음의 단어를 찾습니다:

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

버전 3.5에서 변경: 고정 길이의 그룹 참조에 대한 지원이 추가되었습니다.

(?<!...)

문자열의 현재 위치 앞에 ...와의 일치가 없으면 일치합니다. 이를 부정적인 뒤돌아보기 어서션 (*negative lookbehind assertion*)이라고 합니다. 긍정적인 뒤돌아보기 어서션과 마찬가지로, 포함된 패턴은 고정 길이의 문자열과 일치해야 합니다. 부정적인 뒤돌아보기 어서션으로 시작하는 패턴은 검색되는 문자열의 시작 부분에서 일치 할 수 있습니다.

(?(id/name)yes-pattern|no-pattern)

주어진 *id*나 *name*의 그룹이 있으면 `yes-pattern`과, 그렇지 않으면 `no-pattern`과 일치하려고 시도합니다. `no-pattern`은 선택적이며 생략될 수 있습니다. 예를 들어, `(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)`는 정교하지 않은 전자 메일 일치 패턴인데, `'<user@host.com>'` 및 `'user@host.com'`과 일치하지만, `'<user@host.com'`이나 `'user@host.com>'`과는 일치하지 않습니다.

버전 3.12에서 변경: Group *id* can only contain ASCII digits. In `bytes` patterns, group *name* can only contain bytes in the ASCII range (`b'\x00'-b'\x7f'`).

특수 시퀀스는 `'\'`와 아래 목록의 문자로 구성됩니다. 일반 문자가 ASCII 숫자나 ASCII 글자가 아니면, 결과 RE는 두 번째 문자와 일치합니다. 예를 들어, `\$`는 문자 `'$'`와 일치합니다.

\number

같은 번호의 그룹 내용과 일치합니다. 그룹은 1부터 번호가 매겨집니다. 예를 들어, `(.+) \1`은 `'the the'`나 `'55 55'`와 일치하지만, `'thethe'`와는 일치하지 않습니다 (그룹 뒤의 공백에 유의하십시오). 이 특수 시퀀스는 첫 99개 그룹 중 하나와 일치하는 데에만 사용될 수 있습니다. *number*의 첫 자릿수가 0이거나, *number*가 3 자릿수면, 그룹 일치로 해석되지 않고, 8진수 값 *number*를 갖는 문자로 해석됩니다. 문자 클래스의 `'['`와 `']'` 안에서는, 모든 숫자 이스케이프가 문자로 처리됩니다.

\A

문자열의 시작 부분에서만 일치합니다.

\b

Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of word characters. Note that formally, `\b` is defined as the boundary between a `\w` and a `\W` character (or vice versa), or between `\w` and the beginning or end of the string. This means that `r'\bat\b'` matches `'at'`, `'at.'`, `'(at)'`, and `'as at ay'` but not `'attempt'` or `'atlas'`.

The default word characters in Unicode (str) patterns are Unicode alphanumerics and the underscore, but this can be changed by using the `ASCII` flag. Word boundaries are determined by the current locale if the `LOCALE` flag is used.

> **참고**
>
> Inside a character range, `\b` represents the backspace character, for compatibility with Python' s string literals.

\B

Matches the empty string, but only when it is *not* at the beginning or end of a word. This means that `r'at\B'` matches `'athens'`, `'atom'`, `'attorney'`, but not `'at'`, `'at.'`, or `'at!'`. `\B` is the opposite of `\b`, so word characters in Unicode (str) patterns are Unicode alphanumerics or the underscore, although this can be changed by using the `ASCII` flag. Word boundaries are determined by the current locale if the `LOCALE` flag is used.

버전 3.14에서 변경: `\B` now matches empty input string.

\d

유니코드 (str) 패턴일 때:
Matches any Unicode decimal digit (that is, any character in Unicode character category [Nd]). This includes `[0-9]`, and also many other digit characters.

Matches `[0-9]` if the *ASCII* flag is used.

8비트 (bytes) 패턴일 때:
Matches any decimal digit in the ASCII character set; this is equivalent to `[0-9]`.

\D

Matches any character which is not a decimal digit. This is the opposite of `\d`.

Matches `[^0-9]` if the *ASCII* flag is used.

\s

유니코드 (str) 패턴일 때:
Matches Unicode whitespace characters (as defined by *str.isspace()*). This includes `[ \t\n\r\f\v]`, and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages.

Matches `[ \t\n\r\f\v]` if the *ASCII* flag is used.

8비트 (bytes) 패턴일 때:
ASCII 문자 집합에서 공백으로 간주하는 문자와 일치합니다; 이것은 `[ \t\n\r\f\v]`와 동등합니다.

\S

Matches any character which is not a whitespace character. This is the opposite of `\s`.

Matches `[^ \t\n\r\f\v]` if the *ASCII* flag is used.

\w

유니코드 (str) 패턴일 때:
Matches Unicode word characters; this includes all Unicode alphanumeric characters (as defined by *str.isalnum()*), as well as the underscore (_).

Matches `[a-zA-Z0-9_]` if the *ASCII* flag is used.

8비트 (bytes) 패턴일 때:
Matches characters considered alphanumeric in the ASCII character set; this is equivalent to `[a-zA-Z0-9_]`. If the *LOCALE* flag is used, matches characters considered alphanumeric in the current locale and the underscore.

\W

Matches any character which is not a word character. This is the opposite of `\w`. By default, matches non-underscore (_) characters for which *str.isalnum()* returns `False`.

Matches `[^a-zA-Z0-9_]` if the *ASCII* flag is used.

If the *LOCALE* flag is used, matches characters which are neither alphanumeric in the current locale nor the underscore.

\z

문자열 끝에만 일치합니다.

Added in version 3.14.

\Z

The same as `\z`. For compatibility with old Python versions.

Most of the escape sequences supported by Python string literals are also accepted by the regular expression parser:

```
\a      \b      \f      \n
\N      \r      \t      \u
\U      \v      \x      \\
```

(`\b`는 단어 경계를 나타내는 데 사용되며, 문자 클래스 내에서만 "백스페이스"를 의미함에 유의하십시오.)

`'\u'`, `'\U'`, and `'\N'` escape sequences are only recognized in Unicode (str) patterns. In bytes patterns they are errors. Unknown escapes of ASCII letters are reserved for future use and treated as errors.

8진수 이스케이프는 제한된 형식으로 포함됩니다. 첫 번째 숫자가 0이거나, 3개의 8진수가 있으면, 8진수 이스케이프로 간주합니다. 그렇지 않으면, 그룹 참조입니다. 문자열 리터럴과 마찬가지로, 8진수 이스케이프 길이는 항상 최대 3자리입니다.

버전 3.3에서 변경: `'\u'`와 `'\U'` 이스케이프 시퀀스가 추가되었습니다.

버전 3.6에서 변경: `'\'`와 ASCII 글자로 구성된 알 수 없는 이스케이프는 이제 에러입니다.

버전 3.8에서 변경: The `'\N{name}'` escape sequence has been added. As in string literals, it expands to the named Unicode character (e.g. `'\N{EM DASH}'`).

6.3.2 모듈 내용

모듈은 몇 가지 함수, 상수 및 예외를 정의합니다. 함수 중 일부는 컴파일된 정규식의 모든 기능을 갖춘 메서드의 단순화된 버전입니다. 대부분의 사소하지 않은 응용 프로그램은 항상 컴파일된 형식을 사용합니다.

Flags

버전 3.6에서 변경: 플래그 상수는 이제 *enum.IntFlag*의 서브 클래스인 *RegexFlag*의 인스턴스입니다.

class re.**RegexFlag**

An *enum.IntFlag* class containing the regex options listed below.

Added in version 3.11: - added to `__all__`

re.**A**

re.**ASCII**

Make `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` and `\S` perform ASCII-only matching instead of full Unicode matching. This is only meaningful for Unicode (str) patterns, and is ignored for bytes patterns.

Corresponds to the inline flag `(?a)`.

> **참고**
>
> The *U* flag still exists for backward compatibility, but is redundant in Python 3 since matches are Unicode by default for `str` patterns, and Unicode matching isn't allowed for bytes patterns. *UNICODE* and the inline flag `(?u)` are similarly redundant.

re.**DEBUG**

Display debug information about compiled expression.

No corresponding inline flag.

re.**I**

re.**IGNORECASE**

Perform case-insensitive matching; expressions like `[A-Z]` will also match lowercase letters. Full Unicode matching (such as `Ü` matching `ü`) also works unless the *ASCII* flag is used to disable non-ASCII matches. The current locale does not change the effect of this flag unless the *LOCALE* flag is also used.

Corresponds to the inline flag `(?i)`.

Note that when the Unicode patterns `[a-z]` or `[A-Z]` are used in combination with the *IGNORECASE* flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters: 'İ' (U+0130, Latin capital letter I with dot above), 'ı' (U+0131, Latin small letter dotless i), 'ſ' (U+017F, Latin small letter long s) and 'K' (U+212A, Kelvin sign). If the *ASCII* flag is used, only letters 'a' to 'z' and 'A' to 'Z' are matched.

re.**L**

re.**LOCALE**

Make \w, \W, \b, \B and case-insensitive matching dependent on the current locale. This flag can be used only with bytes patterns.

Corresponds to the inline flag (?L).

> **경고**
>
> This flag is discouraged; consider Unicode matching instead. The locale mechanism is very unreliable as it only handles one "culture" at a time and only works with 8-bit locales. Unicode matching is enabled by default for Unicode (str) patterns and it is able to handle different locales and languages.

버전 3.6에서 변경: *LOCALE* can be used only with bytes patterns and is not compatible with *ASCII*.

버전 3.7에서 변경: Compiled regular expression objects with the *LOCALE* flag no longer depend on the locale at compile time. Only the locale at matching time affects the result of matching.

re.**M**

re.**MULTILINE**

When specified, the pattern character '^' matches at the beginning of the string and at the beginning of each line (immediately following each newline); and the pattern character '$' matches at the end of the string and at the end of each line (immediately preceding each newline). By default, '^' matches only at the beginning of the string, and '$' only at the end of the string and immediately before the newline (if any) at the end of the string.

Corresponds to the inline flag (?m).

re.**NOFLAG**

Indicates no flag being applied, the value is 0. This flag may be used as a default value for a function keyword argument or as a base value that will be conditionally ORed with other flags. Example of use as a default value:

```
def myfunc(text, flag=re.NOFLAG):
    return re.match(text, flag)
```

Added in version 3.11.

re.**S**

re.**DOTALL**

Make the '.' special character match any character at all, including a newline; without this flag, '.' will match anything *except* a newline.

Corresponds to the inline flag (?s).

re.**U**

re.**UNICODE**

In Python 3, Unicode characters are matched by default for str patterns. This flag is therefore redundant with **no effect** and is only kept for backward compatibility.

See *ASCII* to restrict matching to ASCII characters instead.

re.**X**

re.**VERBOSE**

This flag allows you to write regular expressions that look nicer and are more readable by allowing you to visually separate logical sections of the pattern and add comments. Whitespace within the pattern is ignored, except when in a character class, or when preceded by an unescaped backslash, or within tokens like *?, (?: or (?P<...>. For example, (? : and * ? are not allowed. When a line contains a # that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such # through the end of the line are ignored.

이것은 십진수와 일치하는 다음 두 정규식 객체는 기능적으로 같음을 뜻합니다:

```
a = re.compile(r"""\d +  # the integral part
                  \.    # the decimal point
                  \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

인라인 플래그 `(?x)` 에 해당합니다.

Functions

`re.compile`(*pattern*, *flags=0*)

정규식 패턴을 정규식 객체로 컴파일합니다. 정규식 객체는 아래에 설명되는 `match()`, `search()` 및 기타 메서드를 일치시키는 데 사용할 수 있습니다.

The expression' s behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

시퀀스

```
prog = re.compile(pattern)
result = prog.match(string)
```

는 다음과 동등합니다

```
result = re.match(pattern, string)
```

하지만 정규식이 단일 프로그램에서 여러 번 사용될 때, `re.compile()`을 사용하고 결과 정규식 객체를 저장하여 재사용하는 것이 더 효율적입니다.

> **참고**
>
> `re.compile()`과 모듈 수준 일치 함수에 전달된 가장 최근 패턴의 컴파일된 버전이 캐시 되므로, 한 번에 몇 가지 정규식만 사용하는 프로그램은 정규식 컴파일에 대해 신경 쓸 필요가 없습니다.

`re.search`(*pattern*, *string*, *flags=0*)

Scan through *string* looking for the first location where the regular expression *pattern* produces a match, and return a corresponding `Match`. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

The expression' s behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

`re.match`(*pattern*, *string*, *flags=0*)

If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding `Match`. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

`MULTILINE` 모드에서도, `re.match()`는 각 줄의 시작 부분이 아니라 문자열의 시작 부분에서만 일치함에 유의하십시오.

*string*의 모든 위치에서 일치를 찾으려면, 대신 `search()`를 사용하십시오 (*search*() 대 *match*()도 참조하십시오).

The expression' s behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

`re.fullmatch`(*pattern*, *string*, *flags=0*)

If the whole *string* matches the regular expression *pattern*, return a corresponding `Match`. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

The expression' s behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

Added in version 3.4.

re.**split**(*pattern*, *string*, *maxsplit=0*, *flags=0*)

*string*을 *pattern*으로 나눕니다. *pattern*에서 포착하는 괄호가 사용되면 패턴의 모든 그룹 텍스트도 결과 리스트의 일부로 반환됩니다. *maxsplit*이 0이 아니면, 최대 *maxsplit* 분할이 발생하고, 나머지 문자열이 리스트의 마지막 요소로 반환됩니다.

```
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', maxsplit=1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

구분자에 포착하는 그룹이 있고 문자열 시작 부분에서 일치하면, 결과는 빈 문자열로 시작됩니다. 문자열의 끝에 대해서도 마찬가지입니다:

```
>>> re.split(r'(\W+)', '...words, words...')
['', '...', 'words', ', ', 'words', '...', '']
```

그런 식으로, 구분자 구성 요소는 항상 결과 리스트 내의 같은 상대 인덱스에서 발견됩니다.

Adjacent empty matches are not possible, but an empty match can occur immediately after a non-empty match.

```
>>> re.split(r'\b', 'Words, words, words.')
['', 'Words', ', ', 'words', ', ', 'words', '.']
>>> re.split(r'\W*', '...words...')
['', '', 'w', 'o', 'r', 'd', 's', '', '']
>>> re.split(r'(\W*)', '...words...')
['', '...', '', '', 'w', '', 'o', '', 'r', '', 'd', '', 's', '...', '', '', '']
```

The expression' s behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the | operator).

버전 3.1에서 변경: 선택적 flags 인자를 추가했습니다.

버전 3.7에서 변경: 빈 문자열과 일치 할 수 있는 패턴으로 분할하는 지원을 추가했습니다.

버전 3.13부터 폐지됨: Passing *maxsplit* and *flags* as positional arguments is deprecated. In future Python versions they will be *keyword-only parameters*.

re.**findall**(*pattern*, *string*, *flags=0*)

Return all non-overlapping matches of *pattern* in *string*, as a list of strings or tuples. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.

The result depends on the number of capturing groups in the pattern. If there are no groups, return a list of strings matching the whole pattern. If there is exactly one group, return a list of strings matching that group. If multiple groups are present, return a list of tuples of strings matching the groups. Non-capturing groups do not affect the form of the result.

```
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.findall(r'(\w+)=(\d+)', 'set width=20 and height=10')
[('width', '20'), ('height', '10')]
```

The expression' s behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the | operator).

버전 3.7에서 변경: 비어 있지 않은 일치는 이제 이전의 비어 있는 일치 직후에 시작할 수 있습니다.

re.**finditer**(*pattern*, *string*, *flags=0*)

Return an *iterator* yielding `Match` objects over all non-overlapping matches for the RE *pattern* in *string*. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.

The expression' s behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

버전 3.7에서 변경: 비어 있지 않은 일치는 이제 이전의 비어 있는 일치 직후에 시작할 수 있습니다.

re.**sub**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

*string*에서 겹치지 않는 *pattern*의 가장 왼쪽 일치를 *repl*로 치환하여 얻은 문자열을 반환합니다. 패턴을 찾지 못하면, *string*이 변경되지 않고 반환됩니다. *repl*은 문자열이나 함수가 될 수 있습니다; 문자열이면 모든 역 슬래시 이스케이프가 처리됩니다. 즉, `\n`은 단일 개행 문자로 변환되고, `\r`는 캐리지 리턴으로 변환되고, 등등. 알 수 없는 ASCII 글자 이스케이프는 나중에 사용하기 위해 예약되어 있으며 에러로 처리됩니다. `\&`와 같은 다른 알려지지 않은 이스케이프는 그대로 있습니다. `\6`과 같은 역참조는 패턴에서 그룹 6과 일치하는 부분 문자열로 치환됩니다. 예를 들면:

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...        r'static PyObject*\npy_\1(void)\n{',
...        'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

If *repl* is a function, it is called for every non-overlapping occurrence of *pattern*. The function takes a single `Match` argument, and returns the replacement string. For example:

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
...
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

The pattern may be a string or a `Pattern`.

The optional argument *count* is the maximum number of pattern occurrences to be replaced; *count* must be a non-negative integer. If omitted or zero, all occurrences will be replaced.

Adjacent empty matches are not possible, but an empty match can occur immediately after a non-empty match. As a result, `sub('x*', '-', 'abxd')` returns `'-a-b--d-'` instead of `'-a-b-d-'`.

문자열형 *repl* 인자에서, 위에 설명된 문자 이스케이프와 역참조 외에, `\g<name>`는 `(?P<name>...)` 문법으로 정의한 `name`이라는 그룹에 일치하는 부분 문자열을 사용합니다. `\g<number>`는 해당 그룹 번호를 사용합니다; 따라서 `\g<2>`는 `\2`와 동등하지만, `\g<2>0`와 같은 치환에서 모호하지 않습니다. `\20`은 그룹 2에 대한 참조에 리터럴 문자 `'0'`이 뒤에 오는 것이 아니라, 그룹 20에 대한 참조로 해석됩니다. 역참조 `\g<0>`은 RE와 일치하는 전체 부분 문자열을 치환합니다.

The expression' s behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the `|` operator).

버전 3.1에서 변경: 선택적 flags 인자를 추가했습니다.

버전 3.5에서 변경: 일치하지 않는 그룹은 빈 문자열로 치환됩니다.

버전 3.6에서 변경: *pattern*의 `'\'`와 ASCII 글자(letter)로 구성된 알 수 없는 이스케이프는 이제 에러입니다.

버전 3.7에서 변경: Unknown escapes in *repl* consisting of `'\'` and an ASCII letter now are errors. An empty match can occur immediately after a non-empty match.

버전 3.12에서 변경: Group *id* can only contain ASCII digits. In `bytes` replacement strings, group *name* can only contain bytes in the ASCII range (`b'\x00'-b'\x7f'`).

버전 3.13부터 폐지됨: Passing *count* and *flags* as positional arguments is deprecated. In future Python versions they will be *keyword-only parameters*.

re.**subn**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

*sub()*와 같은 연산을 수행하지만, 튜플 (new_string, number_of_subs_made)를 반환합니다.

The expression' s behaviour can be modified by specifying a *flags* value. Values can be any of the *flags* variables, combined using bitwise OR (the | operator).

re.**escape**(*pattern*)

*pattern*에서 특수 문자를 이스케이프 처리합니다. 이것은 정규식 메타 문자가 포함되어있을 수 있는 임의의 리터럴 문자열을 일치시키려는 경우에 유용합니다. 예를 들면:

```
>>> print(re.escape('https://www.python.org'))
https://www\.python\.org

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789!\#\$%\&'\*\+\-\.\^_`\|\~:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print('|'.join(map(re.escape, sorted(operators, reverse=True))))
/|\-|\+|\*\*|\*
```

이 함수는 *sub()*와 *subn()*의 치환 문자열에 사용하면 안 되며, 역 슬래시만 이스케이프 해야 합니다. 예를 들면:

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'), sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

버전 3.3에서 변경: '_' 문자는 더는 이스케이프 되지 않습니다.

버전 3.7에서 변경: 정규식에서 특별한 의미를 가질 수 있는 문자만 이스케이프 됩니다. 결과적으로, '!', '"', '%', "'", ',', '/', ':', ';', '<', '=', '>', '@' 및 "'"는 더는 이스케이프 되지 않습니다.

re.**purge**()

정규식 캐시를 지웁니다.

Exceptions

exception re.**PatternError**(*msg*, *pattern=None*, *pos=None*)

Exception raised when a string passed to one of the functions here is not a valid regular expression (for example, it might contain unmatched parentheses) or when some other error occurs during compilation or matching. It is never an error if a string contains no match for a pattern. The PatternError instance has the following additional attributes:

msg

포맷되지 않은 에러 메시지.

pattern

정규식 패턴.

pos

컴파일이 실패한 위치를 가리키는 *pattern*의 인덱스 (None일 수 있습니다).

lineno

*pos*에 해당하는 줄 (None일 수 있습니다).

colno

*pos*에 해당하는 열 (`None`일 수 있습니다).

버전 3.5에서 변경: 추가 어트리뷰트가 추가되었습니다.

버전 3.13에서 변경: `PatternError` was originally named `error`; the latter is kept as an alias for backward compatibility.

6.3.3 정규식 객체

class re.**Pattern**

Compiled regular expression object returned by `re.compile()`.

버전 3.9에서 변경: `re.Pattern` supports `[]` to indicate a Unicode (str) or bytes pattern. See 제네릭 에일리어스 형.

Pattern.**search**(*string*[, *pos*[, *endpos*]])

Scan through *string* looking for the first location where this regular expression produces a match, and return a corresponding `Match`. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

선택적 두 번째 매개 변수 *pos*는 검색을 시작할 문자열의 인덱스를 제공합니다; 기본값은 `0`입니다. 이것은 문자열을 슬라이싱하는 것과 완전히 동등하지는 않습니다; `'^'` 패턴 문자는 문자열의 실제 시작 부분과 개행 직후의 위치에서 일치하지만, 검색을 시작할 색인에서 반드시 일치하지는 않습니다.

선택적 매개 변수 *endpos*는 문자열을 어디까지 검색할지를 제한합니다; 문자열이 *endpos* 문자 길이인 것처럼 취급되어, 일치를 찾기 위해 *pos*에서 `endpos - 1`까지의 문자만 검색됩니다. *endpos*가 *pos*보다 작으면 일치는 없습니다; 그렇지 않으면, *rx*가 컴파일된 정규식 객체일 때, `rx.search(string, 0, 50)`는 `rx.search(string[:50], 0)`와 동등합니다.

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match at index 0
<re.Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)  # No match; search doesn't include the "d"
```

Pattern.**match**(*string*[, *pos*[, *endpos*]])

If zero or more characters at the *beginning* of *string* match this regular expression, return a corresponding `Match`. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

선택적 *pos*와 *endpos* 매개 변수는 `search()` 메서드에서와 같은 의미입니다.

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd character of "dog".
<re.Match object; span=(1, 2), match='o'>
```

*string*의 임의 위치에서 일치를 찾으려면, 대신 `search()`를 사용하십시오 (*search*() 대 *match*()도 참조하십시오).

Pattern.**fullmatch**(*string*[, *pos*[, *endpos*]])

If the whole *string* matches this regular expression, return a corresponding `Match`. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

선택적 *pos*와 *endpos* 매개 변수는 `search()` 메서드에서와 같은 의미입니다.

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")      # No match as "o" is not at the start of "dog
↪".
>>> pattern.fullmatch("ogre")     # No match as not the full string matches.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> pattern.fullmatch("doggie", 1, 3)   # Matches within given limits.
<re.Match object; span=(1, 3), match='og'>
```

Added in version 3.4.

`Pattern.split(string, maxsplit=0)`

`split()` 함수와 같은데, 컴파일된 패턴을 사용합니다.

`Pattern.findall(string[, pos[, endpos]])`

`findall()` 함수와 유사한데, 컴파일된 패턴을 사용합니다. 하지만, `search()`처럼 검색 영역을 제한하는 선택적 *pos*와 *endpos* 매개 변수도 받아들입니다.

`Pattern.finditer(string[, pos[, endpos]])`

`finditer()` 함수와 유사한데, 컴파일된 패턴을 사용합니다. 하지만, `search()`처럼 검색 영역을 제한하는 선택적 *pos*와 *endpos* 매개 변수도 받아들입니다.

`Pattern.sub(repl, string, count=0)`

`sub()` 함수와 같은데, 컴파일된 패턴을 사용합니다.

`Pattern.subn(repl, string, count=0)`

`subn()` 함수와 같은데, 컴파일된 패턴을 사용합니다.

`Pattern.flags`

The regex matching flags. This is a combination of the flags given to `compile()`, any `(?...)` inline flags in the pattern, and implicit flags such as `UNICODE` if the pattern is a Unicode string.

`Pattern.groups`

패턴에 있는 포착 그룹 수.

`Pattern.groupindex`

`(?P<id>)`로 정의된 기호 그룹 이름을 그룹 번호에 매핑하는 딕셔너리. 패턴에 기호 그룹이 사용되지 않으면 딕셔너리는 비어 있습니다.

`Pattern.pattern`

패턴 객체가 컴파일된 패턴 문자열.

버전 3.7에서 변경: `copy.copy()`와 `copy.deepcopy()` 지원이 추가되었습니다. 컴파일된 정규식 객체는 원자적이라고 간주합니다.

6.3.4 일치 객체

일치 객체는 항상 불리언 값 `True`를 가집니다. `match()`와 `search()`는 일치가 없을 때 `None`을 반환하기 때문에, 간단한 `if` 문으로 일치가 있는지 검사할 수 있습니다:

```
match = re.search(pattern, string)
if match:
    process(match)
```

class `re.Match`

Match object returned by successful `match`es and `search`es.

버전 3.9에서 변경: `re.Match` supports `[]` to indicate a Unicode (str) or bytes match. See 제네릭 에일리어스 형.

`Match.expand(template)`

Return the string obtained by doing backslash substitution on the template string *template*, as done by the `sub()` method. Escapes such as `\n` are converted to the appropriate characters, and numeric backreferences (`\1`, `\2`) and named backreferences (`\g<1>`, `\g<name>`) are replaced by the contents of the corresponding group. The backreference `\g<0>` will be replaced by the entire match.

버전 3.5에서 변경: 일치하지 않는 그룹은 빈 문자열로 치환됩니다.

Match.**group**([*group1*, ...])

일치의 하나 이상의 서브 그룹을 반환합니다. 단일 인자가 있으면, 결과는 단일 문자열입니다; 인자가 여러 개면, 결과는 인자당 하나의 항목이 있는 튜플입니다. 인자가 없으면, *group1*의 기본값은 0입니다 (전체 일치가 반환됩니다). *groupN* 인자가 0이면, 해당 반환 값은 전체 일치 문자열입니다; 경계를 포함하는 범위 [1..99]에 있으면, 해당 괄호로 묶은 그룹과 일치하는 문자열입니다. 그룹 번호가 음수이거나 패턴에 정의된 그룹 수보다 크면, *IndexError* 예외가 발생합니다. 패턴이 일치하지 않는 부분에 그룹이 포함되어 있으면, 해당 결과는 None입니다. 그룹이 여러 번 일치하는 패턴의 일부에 포함되어 있으면, 마지막 일치가 반환됩니다.

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)       # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)    # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

정규식이 (?P<name>...) 문법을 사용하면, *groupN* 인자는 그룹 이름으로 그룹을 식별하는 문자열일 수도 있습니다. 문자열 인자가 패턴의 그룹 이름으로 사용되지 않으면, *IndexError* 예외가 발생합니다.

적당히 복잡한 예:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

이름있는 그룹은 인덱스로 참조할 수도 있습니다:

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

그룹이 여러 번 일치하면, 마지막 일치만 액세스 할 수 있습니다:

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

Match.**__getitem__**(*g*)

이것은 m.group(g) 와 같습니다. 일치에서 개별 그룹에 더 쉽게 액세스 할 수 있게 합니다:

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]       # The entire match
'Isaac Newton'
>>> m[1]       # The first parenthesized subgroup.
'Isaac'
>>> m[2]       # The second parenthesized subgroup.
'Newton'
```

Named groups are supported as well:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Isaac Newton")
>>> m['first_name']
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
'Isaac'
>>> m['last_name']
'Newton'
```

Added in version 3.6.

`Match.groups(default=None)`

1에서 패턴에 있는 그룹의 수까지, 일치의 모든 서브 그룹을 포함하는 튜플을 반환합니다. *default* 인자는 일치에 참여하지 않은 그룹에 사용됩니다; 기본값은 `None` 입니다.

예를 들면:

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

우리가 소수점과 그 이후의 모든 것을 선택적으로 만들면, 모든 그룹이 일치에 참여하지 않을 수 있습니다. 이 그룹은 *default* 인자가 주어지지 않는 한 기본값 `None` 이 됩니다:

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # Second group defaults to None.
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

`Match.groupdict(default=None)`

일치의 모든 이름 있는 서브 그룹을 포함하고, 서브 그룹의 이름을 키로 사용하는 딕셔너리를 반환합니다. *default* 인자는 일치에 참여하지 않은 그룹에 사용됩니다; 기본값은 `None` 입니다. 예를 들면:

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

`Match.start([group])`

`Match.end([group])`

*group*과 일치하는 부분 문자열의 시작과 끝 인덱스를 반환합니다; *group*의 기본값은 0입니다 (전체 일치 문자열을 뜻합니다). *group*이 있지만, 일치에 기여하지 않으면, `-1`을 반환합니다. 일치 객체 *m*과 일치에 기여한 그룹 *g*에서, 그룹 *g*와 일치하는 부분 문자열(`m.group(g)` 와 동등합니다)은 다음과 같습니다

```
m.string[m.start(g):m.end(g)]
```

*group*이 널 문자열과 일치하면 `m.start(group)` 은 `m.end(group)` 와 같음에 유의하십시오. 예를 들어, `m = re.search('b(c?)', 'cba')` 이후에, `m.start(0)` 은 1이고, `m.end(0)` 은 2이며, `m.start(1)` 과 `m.end(1)` 은 모두 2이고, `m.start(2)` 는 *IndexError* 예외를 발생시킵니다.

전자 메일 주소에서 *remove_this*를 제거하는 예:

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

`Match.span([group])`

일치가 *m*일 때, 2-튜플 `(m.start(group), m.end(group))` 를 반환합니다. *group*이 일치에 기여하지 않으면, 이것은 `(-1, -1)` 임에 유의하십시오. *group*의 기본값은 0으로, 전체 일치입니다.

Match.**pos**

정규식 객체의 *search()* 나 *match()* 메서드에 전달된 *pos* 값. 이것은 RE 엔진이 일치를 찾기 시작한 string에 대한 인덱스입니다.

Match.**endpos**

정규식 객체의 *search()* 나 *match()* 메서드에 전달된 *endpos* 값. 이것은 RE 엔진이 넘어가지 않을 string에 대한 인덱스입니다.

Match.**lastindex**

마지막으로 일치하는 포착 그룹의 정수 인덱스, 또는 그룹이 전혀 일치하지 않으면 `None`. 예를 들어, 정규식 `(a)b`, `((a)(b))` 및 `((ab))` 는 문자열 `'ab'` 에 적용될 경우 `lastindex == 1` 이 되지만, `(a)(b)` 정규식은 같은 문자열에 적용될 때 `lastindex == 2` 가 됩니다.

Match.**lastgroup**

마지막으로 일치하는 포착 그룹의 이름, 또는 그룹에 이름이 없거나, 그룹이 전혀 일치하지 않으면 `None`.

Match.**re**

match() 나 *search()* 메서드가 이 일치 인스턴스를 생성한 정규식 객체.

Match.**string**

match() 나 *search()* 에 전달된 문자열.

버전 3.7에서 변경: *copy.copy()* 와 *copy.deepcopy()* 지원이 추가되었습니다. 일치 객체는 원자적이라고 간주합니다.

6.3.5 정규식 예제

쌍 검사하기

이 예제에서는, 다음과 같은 도우미 함수를 사용하여 좀 더 세련되게 일치 객체를 표시합니다:

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

플레이어의 패를 5문자 문자열로 나타내는 포커 프로그램을 작성하고 있다고 가정해봅시다. "a" 는 에이스, "k" 는 킹, "q" 는 퀸, "j" 는 잭, "t" 는 10, "2" 에서 "9" 는 그 값의 카드를 나타냅니다.

주어진 문자열이 유효한 패인지 보려면, 다음과 같이 할 수 있습니다:

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e"))  # Invalid.
>>> displaymatch(valid.match("akt"))    # Invalid.
>>> displaymatch(valid.match("727ak"))  # Valid.
"<Match: '727ak', groups=()>"
```

마지막 패 `"727ak"` 는 페어, 즉 같은 값의 카드 두 장을 포함합니다. 이것을 정규식과 일치시키려면, 역참조를 다음과 같이 사용할 수 있습니다:

```
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak"))     # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak"))     # No pairs.
>>> displaymatch(pair.match("354aa"))     # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

페어가 어떤 카드로 구성되어 있는지 알아내려면, 다음과 같이 일치 객체의 *group()* 메서드를 사용할 수 있습니다:

```
>>> pair = re.compile(r".*(.).*\1")
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'

>>> pair.match("354aa").group(1)
'a'
```

scanf() 시뮬레이션

Python does not currently have an equivalent to `scanf()`. Regular expressions are generally more powerful, though also more verbose, than `scanf()` format strings. The table below offers some more-or-less equivalent mappings between `scanf()` format tokens and regular expressions.

`scanf()` Token	정규식
`%c`	`.`
`%5c`	`.{5}`
`%d`	`[-+]?\d+`
`%e, %E, %f, %g`	`[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?`
`%i`	`[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)`
`%o`	`[-+]?[0-7]+`
`%s`	`\S+`
`%u`	`\d+`
`%x, %X`	`[-+]?(0[xX])?[\dA-Fa-f]+`

다음과 같은 문자열에서 파일명과 숫자를 추출하려면

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

you would use a `scanf()` format like

```
%s - %d errors, %d warnings
```

동등한 정규식은 다음과 같습니다

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() 대 match()

Python offers different primitive operations based on regular expressions:

- *`re.match()`* checks for a match only at the beginning of the string
- *`re.search()`* checks for a match anywhere in the string (this is what Perl does by default)
- *`re.fullmatch()`* checks for entire string to be a match

예를 들면:

```
>>> re.match("c", "abcdef")     # No match
>>> re.search("c", "abcdef")    # Match
<re.Match object; span=(2, 3), match='c'>
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> re.fullmatch("p.*n", "python") # Match
<re.Match object; span=(0, 6), match='python'>
>>> re.fullmatch("r.*n", "python") # No match
```

'^'로 시작하는 정규식은 *search()*와 함께 사용하여 문자열 시작 부분의 일치로 제한 할 수 있습니다:

```
>>> re.match("c", "abcdef")     # No match
>>> re.search("^c", "abcdef")   # No match
>>> re.search("^a", "abcdef")   # Match
<re.Match object; span=(0, 1), match='a'>
```

그러나 *MULTILINE* 모드에서 *match()*는 문자열 시작 부분에서만 일치하지만, '^'로 시작하는 정규식을 *search()*에 사용하면 각 줄의 시작 부분에서 일치합니다.

```
>>> re.match("X", "A\nB\nX", re.MULTILINE)  # No match
>>> re.search("^X", "A\nB\nX", re.MULTILINE)  # Match
<re.Match object; span=(4, 5), match='X'>
```

전화번호부 만들기

*split()*는 문자열을, 전달된 패턴으로 구분된 리스트로 분할합니다. 이 메서드는 전화번호부를 만드는 다음 예제에서 보이듯이 텍스트 데이터를 파이썬에서 쉽게 읽고 수정할 수 있는 데이터 구조로 변환하는 데 매우 중요합니다.

먼저, 여기 입력이 있습니다. 보통 파일에서 올 수 있습니다만, 여기서는 삼중 따옴표로 묶인 문자열 문법을 사용합니다.

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

항목은 하나 이상의 개행으로 구분됩니다. 이제 비어있지 않은 각 줄이 항목이 되도록 문자열을 리스트로 변환합니다:

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

마지막으로, 각 항목을 이름, 성, 전화번호 및 주소로 구성된 리스트로 분할합니다. 주소에 우리의 분할 패턴인 스페이스가 들어있기 때문에, *split()*의 `maxsplit` 매개 변수를 사용합니다:

```
>>> [re.split(":? ", entry, maxsplit=3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

`:?` 패턴은 결과 리스트에 나타나지 않도록, 성 뒤의 콜론과 일치합니다. `maxsplit`로 `4`를 사용하면, 번지수를 거리 이름과 분리 할 수 있습니다:

```
>>> [re.split(":? ", entry, maxsplit=4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

텍스트 뒤섞기

*sub()*는 패턴의 모든 일치를 문자열이나 함수의 결과로 치환합니다. 이 예제는 *sub()*에 텍스트를 "뒤섞는", 즉 문장의 각 단어에서 첫 번째 문자와 마지막 문자를 제외한 모든 문자의 순서를 무작위로 바꾸는 함수를 사용하는 방법을 보여줍니다:

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
...
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

모든 부사 찾기

*findall()*은 *search()*처럼 첫 번째 등장뿐만 아니라, 패턴의 모든 등장과 일치합니다. 예를 들어, 작가가 어떤 텍스트에서 부사를 모두 찾고 싶으면, 다음과 같은 방식으로 *findall()*을 사용할 수 있습니다:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly\b", text)
['carefully', 'quickly']
```

모든 부사와 그 위치 찾기

If one wants more information about all matches of a pattern than the matched text, *finditer()* is useful as it provides *Match* objects instead of strings. Continuing with the previous example, if a writer wanted to find all of the adverbs *and their positions* in some text, they would use *finditer()* in the following manner:

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly\b", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

날 문자열 표기법

날 문자열 표기법(`r"text"`)은 정규식을 합리적인 상태로 유지합니다. 이것 없이는, 정규식의 모든 역 슬래시(`'\'`)를 이스케이프 하기 위해 그 앞에 또 하나의 역 슬래시를 붙여야 합니다. 예를 들어, 다음 두 코드 줄은 기능상으로 같습니다:

```
>>> re.match(r"\W(.)\1\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
```

리터럴 역 슬래시와 일치시키려면, 정규식에서 이스케이프 되어야 합니다. 날 문자열 표기법을 사용하면, `r"\\"`이 됩니다. 날 문자열 표기법을 사용하지 않으면, `"\\\\"`를 사용해야 하는데, 다음 코드 줄들은 기능적으로 같습니다:

```
>>> re.match(r"\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
```

토크나이저 작성하기

토크나이저나 스캐너는 문자열을 분석하여 문자 그룹을 분류합니다. 이것은 컴파일러나 인터프리터를 작성하는 데 유용한 첫 번째 단계입니다.

텍스트 범주는 정규식으로 지정됩니다. 이 기법은 이들을 하나의 마스터 정규식으로 결합하고 연속적인 일치를 반복하는 것입니다:

```
from typing import NamedTuple
import re

class Token(NamedTuple):
    type: str
    value: str
    line: int
    column: int

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('ASSIGN',   r':='),           # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),    # Identifiers
        ('OP',       r'[+\-*/]'),      # Arithmetic operators
        ('NEWLINE',  r'\n'),           # Line endings
        ('SKIP',     r'[ \t]+'),       # Skip over spaces and tabs
        ('MISMATCH', r'.'),            # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
        elif kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
            continue
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected on line {line_num}')
        yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
'''

for token in tokenize(statements):
    print(token)
```

토크나이저는 다음과 같은 출력을 생성합니다:

```
Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
Token(type='ID', value='price', line=3, column=25)
Token(type='OP', value='*', line=3, column=31)
Token(type='ID', value='quantity', line=3, column=33)
Token(type='END', value=';', line=3, column=41)
Token(type='ID', value='tax', line=4, column=8)
Token(type='ASSIGN', value=':=', line=4, column=12)
Token(type='ID', value='price', line=4, column=15)
Token(type='OP', value='*', line=4, column=21)
Token(type='NUMBER', value=0.05, line=4, column=23)
Token(type='END', value=';', line=4, column=27)
Token(type='ENDIF', value='ENDIF', line=5, column=4)
Token(type='END', value=';', line=5, column=9)
```

6.4 difflib — 델타 계산을 위한 도우미

소스 코드: Lib/difflib.py

이 모듈은 시퀀스 비교를 위한 클래스와 함수를 제공합니다. 예를 들어 파일을 비교하는 데 사용할 수 있으며, HTML 및 문맥(context)과 통합(unified) diff를 비롯한 다양한 형식의 파일 차이에 관한 정보를 생성할 수 있습니다. 디렉터리와 파일을 비교하려면, *filecmp* 모듈을 참조하십시오.

class difflib.**SequenceMatcher**

이것은 시퀀스 요소가 해시 가능이기만 하다면, 모든 형의 시퀀스 쌍을 비교할 수 있는 유연한 클래스입니다. 기본 알고리즘은 1980년대 후반에 Ratcliff와 Obershelp가 '게슈탈트 패턴 매칭(gestalt pattern matching)' 이라는 과장된 이름으로 발표한 알고리즘까지 거슬러 올라가는데, 그보다는 약간 더 공을 들였습니다. 아이디어는 "정크" 요소가 없는 가장 긴 연속적으로 일치하는 서브 시퀀스를 찾는 것입니다; 이러한 "정크" 요소는 빈 줄이나 공백과 같은 어떤 의미에서는 흥미롭지 않은 요소들입니다. (정크 처리는 Ratcliff와 Obershelp 알고리즘의 확장입니다.) 그런 다음 같은 아이디어를 일치하는 서브 시퀀스의 왼쪽과 오른쪽에 있는 시퀀스 조각에 재귀적으로 적용합니다. 이것이 최소 편집 시퀀스를 산출하지는 않지만, 사람들에게 "그럴듯해 보이는" 일치를 산출하는 경향이 있습니다.

타이밍: 기본 Ratcliff-Obershelp 알고리즘은 최악의 상황(worst case)에 세제곱 시간이고, 평균적으로(expected case) 제곱 시간입니다. *SequenceMatcher*는 최악의 상황에 제곱 시간이며, 평균적인 동작은 시퀀스에 공통으로 포함된 요소의 수에 따라 복잡한 방식으로 달라집니다; 최상의 경우(best cast)는 선형 시간입니다.

자동 정크 휴리스틱: *SequenceMatcher*는 특정 시퀀스 항목을 자동으로 정크로 처리하는 경험적 방법을 지원합니다. 경험적 방법은 개별 항목이 시퀀스에 나타나는 횟수를 계산합니다. (첫 번째 항목 이후의) 중복된 항목이 시퀀스의 1% 이상을 차지하고 시퀀스의 길이가 최소 200항목 이상이

면, 이 항목은 "흔한" 것으로 표시되고 시퀀스 일치를 위해 정크로 처리됩니다. 이 경험적 방법은 `SequenceMatcher`를 만들 때 `autojunk` 인자를 `False`로 설정하여 끌 수 있습니다.

버전 3.2에서 변경: *autojunk* 매개 변수를 추가했습니다.

class difflib.**Differ**

이것은 텍스트 줄의 시퀀스를 비교하고, 사람이 읽을 수 있는 차이 또는 델타를 생성하는 클래스입니다. Differ는 줄의 시퀀스를 비교하고, 유사한 (거의 일치하는) 줄 내의 문자 시퀀스를 비교하는데 `SequenceMatcher`를 사용합니다.

`Differ` 델타의 각 줄은 2자 코드로 시작합니다:

코드	뜻
`'- '`	시퀀스 1에만 있는 줄
`'+ '`	시퀀스 2에만 있는 줄
`' '`	두 시퀀스에 공통인 줄
`'? '`	두 입력 시퀀스에 없는 줄

'?'로 시작하는 줄은, 시선을 줄 내의 차이로 유도하려고 시도하며, 두 입력 시퀀스 어디에도 나타나지 않습니다. 이 줄은 시퀀스에 스페이스, 탭 또는 줄 넘김과 같은 공백 문자가 포함되면 혼동을 줄 수 있습니다.

class difflib.**HtmlDiff**

이 클래스는 HTML 표를 (또는 표를 포함하는 완전한 HTML 파일을) 만드는 데 사용할 수 있습니다. 이 HTML은 줄 간과 줄 내의 변경을 강조하면서, 텍스트를 나란히 줄 단위로 비교하여 보여줍니다. 표는 전체 또는 문맥 차이 모드로 생성될 수 있습니다.

이 클래스의 생성자는 다음과 같습니다:

__init__(*tabsize=8, wrapcolumn=None, linejunk=None, charjunk=IS_CHARACTER_JUNK*)

`HtmlDiff`의 인스턴스를 초기화합니다.

*tabsize*는 탭 간격을 지정하는 선택적 키워드 인자이며 기본값은 `8`입니다.

*wrapcolumn*는 줄이 자동 줄 넘김 되는 열 번호를 지정하는 선택적 키워드로, 줄을 자동 줄 넘김 하지 않는 `None`이 기본값입니다.

linejunk 와 *charjunk*는 `ndiff()`(`HtmlDiff`가 나란히 배치된 HTML 차이를 만드는 데 사용됩니다)로 전달되는 선택적 키워드 인자입니다. 인자 기본값과 설명은 `ndiff()` 설명서를 참조하십시오.

다음과 같은 메서드가 공개됩니다:

make_file(*fromlines, tolines, fromdesc='', todesc='', context=False, numlines=5, *, charset='utf-8'*)

fromlines 와 *tolines*(문자열의 리스트)를 비교하고, 줄 간 및 줄 내부의 변경을 강조하면서, 줄 단위로 차이를 보여주는 표를 포함하는 완전한 HTML 파일을 문자열로 반환합니다.

fromdesc 와 *todesc*는 from/to 파일 열 헤더 문자열을 지정하는 선택적 키워드 인자입니다 (기본값은 모두 빈 문자열입니다).

context 와 *numlines*는 모두 선택적 키워드 인자입니다. 문맥 차이를 표시하려면 *context*를 `True`로 설정하십시오, 그렇지 않으면 기본값은 전체 파일을 표시하는 `False`입니다. *numlines*의 기본값은 `5`입니다. *context*가 `True` 일 때, *numlines*는 차이 하이라이트를 둘러싸는 문맥 줄의 수를 제어합니다. *context*가 `False`면 *numlines*는 "next" 하이퍼 링크를 사용할 때 차이 하이라이트 앞에 표시되는 줄 수를 제어합니다 (0으로 설정하면 "next" 하이퍼 링크가 다음 차이 하이라이트를 아무런 선행 문맥 줄 없이 브라우저의 맨 위에 놓도록 합니다).

> **참고**
>
> *fromdesc*와 *todesc*는 이스케이프 되지 않은 HTML로 해석되며 신뢰할 수 없는 소스로부터 입력을 받는 동안 적절히 이스케이프 되어야 합니다.

버전 3.5에서 변경: *charset* 키워드 전용 인자가 추가되었습니다. HTML 문서의 기본 문자 집합이 `'ISO-8859-1'`에서 `'utf-8'`로 변경되었습니다.

make_table(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*)

fromlines 와 *tolines*(문자열의 리스트)를 비교하고, 줄 간 및 줄 내부의 변경을 강조하면서, 줄 단위로 차이를 보여주는 완전한 HTML 표를 문자열로 반환합니다.

이 메서드의 인자는 `make_file()` 메서드의 인자와 같습니다.

difflib.**context_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

*a*와 *b*(문자열의 리스트)를 비교합니다; 델타(델타 줄을 생성하는 제너레이터)를 문맥 diff 형식으로 반환합니다.

문맥 diff는 단지 변경된 줄과 몇 줄의 문맥만을 더해서 표시하는 간결한 방법입니다. 변경 사항은 이전/이후 스타일로 표시됩니다. 문맥 줄의 수는 *n*에 의해 설정되며 기본값은 3입니다.

기본적으로, diff 제어 줄(`***`나 `---`가 포함된 것)은 끝에 줄 넘김을 붙여 만들어집니다. 이것은 `io.IOBase.readlines()`로 만들어진 입력이 `io.IOBase.writelines()`와 함께 사용하기에 적합한 diff를 생성하도록 하는 데 유용합니다. 왜냐하면, 입력과 출력 모두 끝에 줄 넘김이 있기 때문입니다.

끝에 줄 넘김이 없는 입력이면, *lineterm* 인자를 `""`로 설정해서 출력에 일관되게 줄 넘김이 포함되지 않게 하십시오.

문맥 diff 형식에는 일반적으로 파일명과 수정 시간에 대한 헤더가 있습니다. 이들 중 일부 또는 전부는 *fromfile*, *tofile*, *fromfiledate* 및 *tofiledate*에 문자열을 사용하여 지정될 수 있습니다. 수정 시간은 일반적으로 ISO 8601 형식으로 표현됩니다. 지정하지 않으면, 문자열들의 기본값은 빈 문자열입니다.

```
>>> import sys
>>> from difflib import *
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(context_diff(s1, s2, fromfile='before.py',
...                        tofile='after.py'))
*** before.py
--- after.py
***************
*** 1,4 ****
! bacon
! eggs
! ham
  guido
--- 1,4 ----
! python
! eggy
! hamster
  guido
```

더욱 자세한 예제는 *difflib*의 명령 줄 인터페이스를 참조하십시오.

difflib.**get_close_matches**(*word*, *possibilities*, *n=3*, *cutoff=0.6*)

최상의 "충분히 좋은" 일치의 리스트를 반환합니다. *word*는 근접 일치가 목표로 하는 시퀀스(일반적으로 문자열)며, *possibilities*는 *word*와 일치시킬 시퀀스의 리스트입니다 (일반적으로 문자열의 리스트).

선택적 인자 *n*(기본값 `3`)은 반환할 근접 일치의 최대 개수입니다; *n*는 `0`보다 커야 합니다.

선택적 인자 *cutoff*(기본값 `0.6`)는 [0, 1] 범위의 float입니다. *word*와의 유사성 점수가 이 값보다 적은 possibilities는 무시됩니다.

possibilities 중에서 가장 좋은 (최대 *n* 개의) 일치가 리스트로 반환되는데, 유사성 점수로 정렬되어 있고 가장 유사한 것이 먼저 나옵니다.

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('pineapple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

difflib.**ndiff**(*a*, *b*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

*a*와 *b*(문자열의 리스트)를 비교합니다; *Differ*-스타일 델타(델타 줄을 생성하는 제너레이터)를 반환합니다.

선택적 키워드 매개 변수 *linejunk* 와 *charjunk*는 필터링 함수(또는 None)입니다:

linejunk: 단일 문자열 인자를 받아들이고 문자열이 정크면 참을 반환하고, 그렇지 않으면 거짓을 반환하는 함수입니다. 기본값은 None 입니다. 모듈 수준의 함수 *IS_LINE_JUNK()*도 있는데, 최대로 한 개의 파운드 문자('#')를 제외하고 눈에 보이는 문자가 없는 줄을 걸러냅니다 – 하지만 하부 *SequenceMatcher* 클래스는 어떤 줄이 잡음으로 볼만큼 자주 등장하는지 동적으로 분석하고, 이것이 보통 이 함수를 사용하는 것보다 효과적입니다.

charjunk: 문자(길이 1의 문자열)를 받아들이고, 문자가 정크면 참을 반환하고, 그렇지 않으면 거짓을 반환하는 함수입니다. 기본값은 모듈 수준의 함수 *IS_CHARACTER_JUNK()* 인데, 공백 문자(스페이스나 탭; 줄 넘김 문자를 포함하는 것은 좋은 생각이 아닙니다)를 걸러냅니다.

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> print(''.join(diff), end="")
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

difflib.**restore**(*sequence*, *which*)

델타를 만든 두 시퀀스 중 하나를 반환합니다.

Differ.compare() 나 *ndiff()*로 만들어진 *sequence*가 주어지면, 파일 1이나 2(매개 변수 *which*)에서 원래 제공되었던 줄을 추출하고, 줄 접두어를 제거합니다.

예:

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print(''.join(restore(diff, 1)), end="")
one
two
three
>>> print(''.join(restore(diff, 2)), end="")
ore
tree
emu
```

difflib.**unified_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

*a*와 *b*(문자열의 리스트)를 비교합니다; 델타(델타 줄을 생성하는 제너레이터)를 통합 diff 형식으로 반환합니다.

통합(unified) diff는 단지 변경된 줄과 몇 줄의 문맥만을 더해서 표시하는 간결한 방법입니다. 변경 사항은 (별도의 이전/이후 블록 대신) 인라인 스타일로 표시됩니다. 문맥 줄의 수는 *n*에 의해 설정되며 기본값은 3입니다.

기본적으로, diff 제어 줄(`---`, `+++` 또는 `@@`가 포함된 것)은 끝에 줄 넘김을 붙여 만들어집니다. 이것은 `io.IOBase.readlines()`로 만들어진 입력이 `io.IOBase.writelines()`와 함께 사용하기에 적합한 diff를 생성하도록 하는 데 유용합니다. 왜냐하면, 입력과 출력 모두 끝에 줄 넘김이 있기 때문입니다.

끝에 줄 넘김이 없는 입력이면, *lineterm* 인자를 `""`로 설정해서 출력에 일관되게 줄 넘김이 포함되지 않게 하십시오.

통합(unified) diff 형식에는 일반적으로 파일명과 수정 시간에 대한 헤더가 있습니다. 이들 중 일부 또는 전부는 *fromfile*, *tofile*, *fromfiledate* 및 *tofiledate*에 문자열을 사용하여 지정될 수 있습니다. 수정 시간은 일반적으로 ISO 8601 형식으로 표현됩니다. 지정하지 않으면, 문자열들의 기본값은 빈 문자열입니다.

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(unified_diff(s1, s2, fromfile='before.py', tofile=
↪'after.py'))
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
 guido
```

더욱 자세한 예제는 *difflib의 명령 줄 인터페이스*를 참조하십시오.

difflib.**diff_bytes**(*dfunc*, *a*, *b*, *fromfile=b''*, *tofile=b''*, *fromfiledate=b''*, *tofiledate=b''*, *n=3*, *lineterm=b'\n'*)

*a*와 *b*(바이트열 객체의 리스트)를 *dfunc*를 사용하여 비교합니다; *dfunc*가 반환하는 형식으로 델타 줄(역시 바이트열)의 시퀀스를 산출합니다. *dfunc*는 콜러블이어야하며, 보통 `unified_diff()` 나 `context_diff()`입니다.

알 수 없거나 일관성 없는 인코딩의 데이터를 비교할 수 있게 합니다. *n*를 제외한 모든 입력은 바이트열 객체여야 합니다, str이 아닙니다. 모든 입력(*n* 제외)을 str로 무손실 변환하고, `dfunc(a, b, fromfile, tofile, fromfiledate, tofiledate, n, lineterm)`를 호출하는 방식으로 작동합니다. *dfunc*의 출력은 다시 바이트로 변환되므로, 여러분이 얻는 델타 줄은 *a*와 *b* 처럼 알 수 없고/일관성 없는 인코딩을 갖습니다.

Added in version 3.5.

difflib.**IS_LINE_JUNK**(*line*)

무시할 수 있는 줄이면 `True`를 반환합니다. *line*이 빈 줄이거나 하나의 `'#'`를 포함하면, 줄 *line*은 무시할 수 있습니다, 그렇지 않으면 무시할 수 없습니다. 이전 버전의 `ndiff()`에서 매개 변수 *linejunk*의 기본값으로 사용되었습니다.

difflib.**IS_CHARACTER_JUNK**(*ch*)

무시할 수 있는 문자면 `True`를 반환합니다. *ch*가 스페이스나 탭이면 문자 *ch*는 무시할 수 있습니다, 그렇지 않으면 무시할 수 없습니다. `ndiff()`에서 매개 변수 *charjunk*의 기본값으로 사용됩니다.

더 보기

Pattern Matching: The Gestalt Approach
Discussion of a similar algorithm by John W. Ratcliff and D. E. Metzener. This was published in Dr. Dobb' s Journal in July, 1988.

6.4.1 SequenceMatcher 객체

`SequenceMatcher` 클래스는 다음과 같은 생성자를 갖습니다:

class difflib.**SequenceMatcher**(*isjunk=None, a='', b='', autojunk=True*)

선택 인자 *isjunk*는 `None`(기본값)이거나, 시퀀스 요소를 받아서 요소가 "정크" 이고, 무시되어야 하는 경우에만 참을 반환하는 하나의 인자 함수여야 합니다. *isjunk*에 `None`을 전달하는 것은, `lambda x: False`를 전달하는 것과 같습니다; 즉, 아무 요소도 무시하지 않습니다. 예를 들어,:

```
lambda x: x in " \t"
```

줄을 문자의 시퀀스로 비교하고, 스페이스와 탭을 무시하고 싶으면, 위와 같은 것을 전달하면 됩니다.

선택적 인자 *a* 와 *b*는 비교할 시퀀스입니다; 둘 다 빈 문자열이 기본값입니다. 두 시퀀스의 요소는 모두 해시 가능해야 합니다.

선택적 인자 *autojunk*는 자동 정크 휴리스틱을 비활성화하는 데 사용할 수 있습니다.

버전 3.2에서 변경: *autojunk* 매개 변수를 추가했습니다.

SequenceMatcher 객체는 세 개의 데이터 어트리뷰트를 갖습니다: *bjunk*는 *isjunk*가 `True` 인 *b* 요소의 집합입니다; *bpopular*는 휴리스틱(비활성화하지 않았다면)에서 흔하다고 판단되는 정크가 아닌 요소의 집합입니다; *b2j*는 *b*의 나머지 요소를 그들이 나타난 위치의 리스트로 매핑하는 dict입니다. *b*가 `set_seqs()` 나 `set_seq2()`로 재설정 될 때마다 세 개 모두 재설정됩니다.

Added in version 3.2: *bjunk* 및 *bpopular* 어트리뷰트

`SequenceMatcher` 객체에는 다음과 같은 메서드가 있습니다:

set_seqs(*a, b*)

비교할 두 시퀀스를 설정합니다.

`SequenceMatcher`는 두 번째 시퀀스에 대한 자세한 정보를 계산하고 캐시 하므로, 많은 시퀀스에 대해 하나의 시퀀스를 비교하려면, `set_seq2()`를 사용하여 자주 사용되는 시퀀스를 한 번 설정하고, `set_seq1()`를 다른 시퀀스 각각에 대해 한 번 반복적으로 호출하십시오.

set_seq1(*a*)

비교할 첫 번째 시퀀스를 설정합니다. 비교할 두 번째 시퀀스는 변경되지 않습니다.

set_seq2(*b*)

비교할 두 번째 시퀀스를 설정합니다. 비교할 첫 번째 시퀀스는 변경되지 않습니다.

find_longest_match(*alo=0, ahi=None, blo=0, bhi=None*)

`a[alo:ahi]` 와 `b[blo:bhi]`에서 가장 긴 일치 블록을 찾습니다.

*isjunk*가 생략되거나 `None` 이면, `find_longest_match()`는 `a[i:i+k]` 가 `b[j:j+k]`와 같은 `(i, j, k)`를 반환하는데, 여기서 `alo <= i <= i+k <= ahi` 이고 `blo <= j <= j+k <= bhi` 입니다. 이 조건을 만족시키는 모든 `(i', j', k')`에 대해, 추가 조건 `k >= k'`, `i <= i'` 와 `i == i'` 면 `j <= j'` 도 만족합니다. 즉, 모든 최대 일치 블록 중에서 *a*에서 가장 먼저 시작하는 블록을 반환하고, *a*에서 가장 먼저 시작하는 모든 최대 일치 블록 중에서 *b*에서 가장 먼저 시작하는 블록을 반환합니다.

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

*isjunk*가 제공되면, 먼저 가장 긴 일치 블록이 상기와 같이 결정되지만, 정크 요소가 블록에 나타나지 않아야 한다는 추가 제약이 있습니다. 그런 다음 그 블록의 좌우에서 정크 요소만 일치시켜 가능한 한 최대로 확장합니다. 그래서 결과 블록은 흥미로운 일치와 인접하게 같은 정크가 등장할 때를 제외하고는, 정크와 일치하지 않습니다.

여기에 이전과 같은 예가 있지만, 스페이스를 정크로 간주합니다. 이렇게 하면 `' abcd'`가 두 번째 시퀀스의 끝에 있는 `' abcd'`와 직접 일치하지 않게 됩니다. 대신 `'abcd'` 만 일치 할 수 있으며, 두 번째 시퀀스에서 가장 왼쪽의 `'abcd'`와 일치합니다:

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

일치하는 블록이 없으면 `(alo, blo, 0)`를 반환합니다.

이 메서드는 네임드 튜플 `Match(a, b, size)`를 반환합니다.

버전 3.9에서 변경: 인자 기본값이 추가되었습니다.

get_matching_blocks()

중첩하지 않는 일치하는 서브 시퀀스를 기술하는 3-튜플의 리스트를 반환합니다. 각 3-튜플은 `(i, j, n)` 형식이며, `a[i:i+n] == b[j:j+n]`를 뜻합니다. 3-튜플은 *i*와 *j*에 대해 단조 증가합니다.

마지막 3-튜플은 더미이며, `(len(a), len(b), 0)` 값을 가집니다. `n == 0` 인 유일한 3-튜플입니다. `(i, j, n)`와 `(i', j', n')`가 리스트에서 인접한 3-튜플이고, 두 번째가 리스트의 마지막 3-튜플이 아니면 `i+n < i'` 또는 `j+n < j'`입니다; 즉, 인접 3-튜플은 항상 인접하지 않은 같은 블록을 나타냅니다.

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes()

*a*를 *b*로 변환하는 방법을 설명하는 5-튜플의 리스트를 반환합니다. 각 튜플은 `(tag, i1, i2, j1, j2)` 형식입니다. 첫 번째 튜플은 `i1 == j1 == 0` 이고, 나머지 튜플에서는 *i1*이 이전 튜플의 *i2*와 같고, 마찬가지로 *j1*은 이전 *j2*와 같습니다.

tag 값은 문자열이고, 이런 의미입니다:

값	뜻
`'replace'`	`a[i1:i2]`를 `b[j1:j2]`로 치환해야 합니다.
`'delete'`	`a[i1:i2]`를 삭제해야 합니다. 이때 `j1 == j2` 임을 유의하십시오.
`'insert'`	`b[j1:j2]`을 `a[i1:i1]`에 삽입해야 합니다. 이때 `i1 == i2` 임을 유의하십시오.
`'equal'`	`a[i1:i2] == b[j1:j2]` (서브 시퀀스가 같습니다).

예를 들면:

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print('{:7}   a[{}:{}] --> b[{}:{}] {!r:>8} --> {!r}'.format(
...         tag, i1, i2, j1, j2, a[i1:i2], b[j1:j2]))
delete    a[0:1] --> b[0:0]      'q' --> ''
equal     a[1:3] --> b[0:2]     'ab' --> 'ab'
replace   a[3:4] --> b[2:3]      'x' --> 'y'
equal     a[4:6] --> b[3:5]     'cd' --> 'cd'
insert    a[6:6] --> b[5:6]       '' --> 'f'
```

get_grouped_opcodes(*n=3*)

최대 *n* 줄의 문맥을 갖는 그룹의 제너레이터를 반환합니다.

get_opcodes() 에서 반환된 그룹으로 출발해서, 이 메서드는 더 작은 변경 클러스터로 나누고, 변경 사항이 없는 중간 범위를 제거합니다.

그룹은 *get_opcodes()* 와 같은 형식으로 반환됩니다.

ratio()

[0, 1] 의 범위의 float로 시퀀스 유사성 척도를 돌려줍니다.

T가 두 시퀀스의 요소의 총 개수이고, M은 일치 개수일 때, 척도는 2.0*M / T입니다. 시퀀스가 같으면 `1.0` 이고, 공통 요소가 없으면 `0.0` 입니다.

get_matching_blocks() 나 *get_opcodes()* 가 아직 호출되지 않았으면, 계산하는 데 비용이 많이 듭니다. 이럴 때, *quick_ratio()* 나 *real_quick_ratio()* 를 먼저 시도하여 상한값을 얻을 수 있습니다.

> **참고**
>
> 주의: *ratio()* 호출의 결과는 인자의 순서에 따라 달라질 수 있습니다. 예를 들어:
>
> ```
> >>> SequenceMatcher(None, 'tide', 'diet').ratio()
> 0.25
> >>> SequenceMatcher(None, 'diet', 'tide').ratio()
> 0.5
> ```

quick_ratio()

비교적 빨리 *ratio()* 의 상한을 반환합니다.

real_quick_ratio()

아주 빨리 *ratio()* 의 상한을 반환합니다.

총 문자 수에 대한 일치 비율을 반환하는 세 가지 메서드는 서로 다른 수준의 근삿값 때문에 다른 결과를 줄 수 있습니다. 하지만 *quick_ratio()* 와 *real_quick_ratio()* 는 항상 최소한 *ratio()* 만큼 큰 값을 줍니다:

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

6.4.2 SequenceMatcher 예제

이 예제에서는 공백을 "정크" 로 간주하여, 두 문자열을 비교합니다:

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                     "private Thread currentThread;",
...                     "private volatile Thread currentThread;")
```

ratio() 는 [0, 1] 범위의 float를 반환하여, 시퀀스의 유사성을 측정합니다. 경험적으로, *ratio()* 값이 0.6 이상이면 시퀀스가 근접하게 일치함을 뜻합니다:

```
>>> print(round(s.ratio(), 3))
0.866
```

시퀀스가 일치하는 부분에만 관심이 있다면, *get_matching_blocks()* 가 유용합니다:

```
>>> for block in s.get_matching_blocks():
...     print("a[%d] and b[%d] match for %d elements" % block)
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

*get_matching_blocks()*에 의해 반환된 마지막 튜플은 항상 더미인 `(len(a), len(b), 0)`이며, 이는 마지막 튜플 요소(일치하는 요소의 수)가 `0` 인 유일한 경우입니다.

첫 번째 시퀀스를 두 번째 시퀀스로 변경하는 방법을 알고 싶다면, *get_opcodes()*를 사용하십시오:

```
>>> for opcode in s.get_opcodes():
...     print("%6s a[%d:%d] b[%d:%d]" % opcode)
 equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
 equal a[8:29] b[17:38]
```

더 보기

- 이 모듈의 *get_close_matches()* 함수는 *SequenceMatcher*를 사용한 간단한 코드 작성을 통해 유용한 작업을 수행하는 방법을 보여줍니다.
- *SequenceMatcher*로 만들어진 작은 응용 프로그램을 위한 간단한 버전 관리 조리법.

6.4.3 Differ 객체

*Differ*가 만든 델타는 **최소** diff라고 주장하지 않음에 유의하십시오. 반대로, 최소 diff는 종종 반 직관적인데, 가능한 모든 곳에서 일치를 취하기 때문입니다. 때로 우발적으로 100페이지가 떨어진 곳에서 일치시키기도 합니다. 동기화 지점을 인접한 일치로 제한하면 가끔 더 긴 diff를 만드는 대신 일종의 지역성을 보존합니다.

Differ 클래스에는 다음과 같은 생성자가 있습니다:

class difflib.**Differ**(*linejunk=None*, *charjunk=None*)

선택적 키워드 매개 변수 *linejunk* 와 *charjunk*는 필터 함수(또는 `None`)를 위한 것입니다:

linejunk: 단일 문자열 인자를 받아들이고 문자열이 정크면 참을 반환하는 함수입니다. 기본값은 `None`이며, 이는 어떤 줄도 정크로 간주하지 않음을 의미합니다.

charjunk: 문자(길이 1의 문자열)를 받아들이고, 문자가 정크면 참을 반환하는 함수입니다. 기본값은 `None`이며, 이는 어떤 문자도 정크로 간주하지 않음을 의미합니다.

이러한 정크 필터링 함수는 차이점을 찾기 위한 일치 속도를 높이고 차이가 나는 줄이나 문자를 무시하지 않습니다. 설명이 필요하면 *find_longest_match()* 메서드의 *isjunk* 매개 변수에 대한 설명을 읽으십시오.

Differ 객체는 단일 메서드를 통해 사용됩니다 (델타가 만들어집니다):

compare(*a*, *b*)

줄의 시퀀스 두 개를 비교하고, 델타(줄의 시퀀스)를 만듭니다.

각 시퀀스는 줄 넘김으로 끝나는 개별 단일 줄 문자열을 포함해야 합니다. 이러한 시퀀스는 파일류 객체의 *readlines()* 메서드로 얻을 수 있습니다. 생성된 델타 역시 파일류 객체의 *writelines()* 메서드를 통해 그대로 인쇄될 수 있도록 줄 넘김으로 끝나는 문자열로 구성됩니다.

6.4.4 Differ 예제

이 예제는 두 개의 텍스트를 비교합니다. 먼저 텍스트를 설정하는데, 줄 넘김 문자로 끝나는 개별 단일 줄 문자열의 시퀀스입니다 (이러한 시퀀스는 파일류 객체의 *readlines()* 메서드로도 얻을 수 있습니다):

```
>>> text1 = '''  1. Beautiful is better than ugly.
...   2. Explicit is better than implicit.
...   3. Simple is better than complex.
...   4. Complex is better than complicated.
... '''.splitlines(keepends=True)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = '''  1. Beautiful is better than ugly.
...   3.   Simple is better than complex.
...   4. Complicated is better than complex.
...   5. Flat is better than nested.
... '''.splitlines(keepends=True)
```

다음으로 Differ 객체의 인스턴스를 만듭니다:

```
>>> d = Differ()
```

Differ 객체의 인스턴스를 만들 때, 줄과 문자 "정크"를 필터링하는 함수를 전달할 수 있음에 유의하십시오. 자세한 내용은 *Differ()* 생성자를 참조하십시오.

마지막으로, 두 개를 비교합니다:

```
>>> result = list(d.compare(text1, text2))
```

`result`는 문자열의 리스트이므로, 예쁜 인쇄를 해봅시다:

```
>>> from pprint import pprint
>>> pprint(result)
['    1. Beautiful is better than ugly.\n',
 '-   2. Explicit is better than implicit.\n',
 '-   3. Simple is better than complex.\n',
 '+   3.   Simple is better than complex.\n',
 '?     ++\n',
 '-   4. Complex is better than complicated.\n',
 '?            ^                     ---- ^\n',
 '+   4. Complicated is better than complex.\n',
 '?           ++++ ^                     ^\n',
 '+   5. Flat is better than nested.\n']
```

여러 줄이 포함된 하나의 문자열로 만들면 이렇게 보입니다:

```
>>> import sys
>>> sys.stdout.writelines(result)
    1. Beautiful is better than ugly.
-   2. Explicit is better than implicit.
-   3. Simple is better than complex.
+   3.   Simple is better than complex.
?     ++
-   4. Complex is better than complicated.
?            ^                     ---- ^
+   4. Complicated is better than complex.
?           ++++ ^                     ^
+   5. Flat is better than nested.
```

6.4.5 difflib의 명령 줄 인터페이스

이 예제는 difflib를 사용하여 `diff`와 유사한 유틸리티를 만드는 방법을 보여줍니다.

```
""" 네 가지 형식의 diff를 제공하는 diffflib.py의 명령줄 인터페이스입니다:

* ndiff:    모든 줄을 나열하고 줄 간 변경 사항을 강조 표시합니다.
* context:  변경 사항의 뭉치를 이전/이후 형식으로 강조 표시합니다.
* unified:  변경 사항의 뭉치를 인라인 형식으로 강조 표시합니다.
* html:     좌우 비교를 변경 사항 강조 표시와 함께 생성합니다.

"""

import sys, os, difflib, argparse
from datetime import datetime, timezone

def file_mtime(path):
    t = datetime.fromtimestamp(os.stat(path).st_mtime,
                               timezone.utc)
    return t.astimezone().isoformat()

def main():

    parser = argparse.ArgumentParser()
    parser.add_argument('-c', action='store_true', default=False,
                        help='Produce a context format diff (default)')
    parser.add_argument('-u', action='store_true', default=False,
                        help='Produce a unified format diff')
    parser.add_argument('-m', action='store_true', default=False,
                        help='Produce HTML side by side diff '
                             '(can use -c and -l in conjunction)')
    parser.add_argument('-n', action='store_true', default=False,
                        help='Produce a ndiff format diff')
    parser.add_argument('-l', '--lines', type=int, default=3,
                        help='Set number of context lines (default 3)')
    parser.add_argument('fromfile')
    parser.add_argument('tofile')
    options = parser.parse_args()

    n = options.lines
    fromfile = options.fromfile
    tofile = options.tofile

    fromdate = file_mtime(fromfile)
    todate = file_mtime(tofile)
    with open(fromfile) as ff:
        fromlines = ff.readlines()
    with open(tofile) as tf:
        tolines = tf.readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile, fromdate,
↪ todate, n=n)
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
    elif options.m:
        diff = difflib.HtmlDiff().make_file(fromlines,tolines,fromfile,tofile,
↪context=options.c,numlines=n)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    else:
        diff = difflib.context_diff(fromlines, tolines, fromfile, tofile, fromdate,
↪ todate, n=n)

    sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
```

6.4.6 ndiff 예제

d이 예제는 *difflib.ndiff()*를 사용하는 방법을 보여줍니다.

```
"""ndiff [-q] file1 file2
    or
ndiff (-r1 | -r2) < ndiff_output > file1_or_file2

사람이 읽기 쉬운 파일 차이 보고서를 표준 출력에 인쇄합니다.  줄 간 및 줄 내 차이점이 모두␣
↪표시됩니다.
두 번째 형식에서는, 표준 입력의 ndiff 보고서로부터
 표준 출력으로 file1 (-r1) 이나 file2 (-r2) 를 다시 생성합니다.

첫 번째 형식에서는, -q ("quiet") 가 지정되지 않으면, 출력의 처음
두 줄은 다음과 같습니다

-: file1
+: file2

나머지 각 줄은 두 글자 코드로 시작합니다:

    "- "    file1 에 고유한 줄
    "+ "    file2 에 고유한 줄
    "  "    두 파일에 공통된 줄
    "? "    두 입력 파일 어디에도 없는 줄

"? " 로 시작하는 줄은 줄 내 차이로 눈을 안내하려고 시도하고,
두 입력 파일 어디에도 존재하지 않습니다.  이 줄들은 소스 파일에
탭 문자가 있을 경우 혼란스러울 수 있습니다.

첫 번째 파일은 "  " 나 "- " 로 시작하는 줄만 유지하고, 두 문자 접두사를
삭제해서 복구할 수 있습니다; ndiff 를 -r1 과 함께 사용하세요.

두 번째 파일도 비슷하게 복구할 수 있지만, "  " 과
"+ " 줄만 유지합니다; ndiff 를 -r2 와 함께 사용하세요; 또는, 유닉스에서, 두 번째 파일은
다음과 같이 출력을 파이핑하여 복구할 수 있습니다

    sed -n '/^[+ ] /s/^..//p'
"""

__version__ = 1, 7, 0

import difflib, sys

def fail(msg):
    out = sys.stderr.write
    out(msg + "\n\n")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    out(__doc__)
    return 0

# 파일을 열고 파일 객체를 반환합니다; 열 수 없으면
# 메시지를 출력하고 0을 반환합니다
def fopen(fname):
    try:
        return open(fname)
    except IOError as detail:
        return fail("couldn't open " + fname + ": " + str(detail))

# 두 파일을 열고 표준 출력으로 diff 를 뿌립니다; 문제가 있으면 거짓을 반환합니다
def fcompare(f1name, f2name):
    f1 = fopen(f1name)
    f2 = fopen(f2name)
    if not f1 or not f2:
        return 0

    a = f1.readlines(); f1.close()
    b = f2.readlines(); f2.close()
    for line in difflib.ndiff(a, b):
        print(line, end=' ')

    return 1

# args (보통 sys.argv[1:]) 를 쪼개고 비교합니다;
# 문제가 있으면 & 문제가 있을 때만 거짓을 반환합니다

def main(args):
    import getopt
    try:
        opts, args = getopt.getopt(args, "qr:")
    except getopt.error as detail:
        return fail(str(detail))
    noisy = 1
    qseen = rseen = 0
    for opt, val in opts:
        if opt == "-q":
            qseen = 1
            noisy = 0
        elif opt == "-r":
            rseen = 1
            whichfile = val
    if qseen and rseen:
        return fail("can't specify both -q and -r")
    if rseen:
        if args:
            return fail("no args allowed with -r option")
        if whichfile in ("1", "2"):
            restore(whichfile)
            return 1
        return fail("-r value must be 1 or 2")
    if len(args) != 2:
        return fail("need 2 filename args")
    f1name, f2name = args
    if noisy:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        print('-:', f1name)
        print('+:', f2name)
    return fcompare(f1name, f2name)

# 표준 입력에서 ndiff 출력을 읽고, file1 (which=='1') 이나
# file2 (which=='2') 를 표준 출력에 인쇄합니다

def restore(which):
    restored = difflib.restore(sys.stdin.readlines(), which)
    sys.stdout.writelines(restored)

if __name__ == '__main__':
    main(sys.argv[1:])
```

6.5 textwrap — 텍스트 래핑과 채우기

소스 코드: Lib/textwrap.py

---

textwrap 모듈은 모든 작업을 수행하는 클래스인 *TextWrapper*뿐만 아니라 몇 가지 편리 함수도 제공합니다. 한두 개의 텍스트 문자열을 래핑(wrapping)하거나 채운(filling)다면, 편리 함수로도 충분해야 합니다; 그렇지 않으면 효율을 위해 *TextWrapper* 인스턴스를 사용해야 합니다.

textwrap.**wrap**(*text, width=70, *, initial_indent=’’, subsequent_indent=’’, expand_tabs=True, replace_whitespace=True, fix_sentence_endings=False, break_long_words=True, drop_whitespace=True, break_on_hyphens=True, tabsize=8, max_lines=None, placeholder=’ [...]’*)

text(문자열)에 있는 단일 문단을 래핑해서 모든 줄의 길이가 최대 *width* 자가 되도록 합니다. 최종 줄 바꿈이 없는 출력 줄의 리스트를 반환합니다.

선택적 키워드 인자는 아래에 설명된 *TextWrapper*의 인스턴스 어트리뷰트에 해당합니다.

wrap() 작동 방식에 대한 자세한 내용은 *TextWrapper.wrap()* 메서드를 참조하십시오.

textwrap.**fill**(*text, width=70, *, initial_indent=’’, subsequent_indent=’’, expand_tabs=True, replace_whitespace=True, fix_sentence_endings=False, break_long_words=True, drop_whitespace=True, break_on_hyphens=True, tabsize=8, max_lines=None, placeholder=’ [...]’*)

*text*에 있는 단일 문단을 래핑하고, 래핑 된 문단을 포함하는 단일 문자열을 반환합니다. *fill()*은 다음의 줄임 표현입니다

```
"\n".join(wrap(text, ...))
```

특히, *fill()*은 *wrap()*과 같은 키워드 인자를 받아들입니다.

textwrap.**shorten**(*text, width, *, fix_sentence_endings=False, break_long_words=True, break_on_hyphens=True, placeholder=’ [...]’*)

주어진 *width*에 맞게 주어진 *text*를 축약하거나 자릅니다.

먼저 *text*에 있는 공백이 축약됩니다 (모든 공백이 단일 스페이스로 치환됩니다). 결과가 *width*에 맞으면 반환됩니다. 그렇지 않으면, 나머지 단어와 *placeholder*가 *width* 내에 맞도록 충분한 단어가 끝에서 삭제됩니다:

```
>>> textwrap.shorten("Hello  world!", width=12)
'Hello world!'
>>> textwrap.shorten("Hello  world!", width=11)
'Hello [...]'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> textwrap.shorten("Hello world", width=10, placeholder="...")
'Hello...'
```

선택적 키워드 인자는 아래에 설명된 *TextWrapper*의 인스턴스 어트리뷰트에 해당합니다. 텍스트가 *TextWrapper fill()* 함수에 전달되기 전에 공백이 축약되므로, *tabsize*, *expand_tabs*, *drop_whitespace* 및 *replace_whitespace* 값을 변경하는 것은 아무 효과가 없음에 유의하십시오.

Added in version 3.4.

textwrap.**dedent**(*text*)

*text*의 모든 줄에서 같은 선행 공백을 제거합니다.

이것은 삼중 따옴표로 묶은 문자열을 소스 코드에서 여전히 들여쓰기 된 형태로 제시하면서, 디스플레이의 왼쪽 가장자리에 맞추는 데 사용할 수 있습니다.

탭과 공백은 모두 공백으로 처리되지만, 이들이 같지 않음에 유의하십시오: `" hello"`와 `"\thello"` 줄에는 공통 선행 공백이 없는 것으로 간주합니다.

공백만 포함하는 줄은 입력에서 무시되고 출력에서 단일 개행 문자로 정규화됩니다.

예를 들면:

```
def test():
    # 빈 줄을 피하려고 첫 줄을 \ 로 끝냅니다!
    s = '''\
    hello
      world
    '''
    print(repr(s))           # '    hello\n      world\n    ' 를 인쇄합니다
    print(repr(dedent(s)))   # 'hello\n  world\n' 를 인쇄합니다
```

textwrap.**indent**(*text*, *prefix*, *predicate=None*)

*text*에서 선택된 줄의 시작 부분에 *prefix*를 추가합니다.

`text.splitlines(True)`를 호출하여 줄을 분할합니다.

기본적으로, *prefix*는 공백으로만 구성되지 않는 모든 줄(마지막 줄 포함)에 추가됩니다.

예를 들면:

```
>>> s = 'hello\n\n \nworld'
>>> indent(s, '  ')
'  hello\n\n \n  world'
```

선택적 *predicate* 인자는 어떤 줄을 들여쓰기할지 제어하는 데 사용될 수 있습니다. 예를 들어, 빈 줄과 공백만 있는 줄에도 *prefix*를 추가하기는 쉽습니다:

```
>>> print(indent(s, '+ ', lambda line: True))
+ hello
+
+
+ world
```

Added in version 3.3.

wrap(), *fill()* 및 *shorten()*은 *TextWrapper* 인스턴스를 만들고 그것의 단일 메서드를 호출하여 작동합니다. 이 인스턴스는 재사용되지 않기 때문에, *wrap()* 및/또는 *fill()*을 사용하여 많은 텍스트 문자열을 처리하는 응용 프로그램의 경우, 여러분 자신의 *TextWrapper* 객체를 만드는 것이 더 효율적일 수 있습니다.

텍스트는 공백과 하이픈이 있는 단어의 하이픈 바로 뒤에서 래핑하는 것을 선호합니다; *TextWrapper.break_long_words*가 거짓으로 설정되어 있지 않으면 그 후에만 긴 단어를 분할합니다.

class textwrap.**TextWrapper**(***kwargs*)

TextWrapper 생성자는 여러 개의 선택적 키워드 인자를 받아들입니다. 각 키워드 인자는 인스턴스 어트리뷰트에 해당합니다, 그래서 예를 들면

```
wrapper = TextWrapper(initial_indent="* ")
```

는 다음과 같습니다

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

같은 *TextWrapper* 객체를 여러 번 재사용 할 수 있으며, 사용 도중 인스턴스 어트리뷰트에 직접 대입하여 옵션을 변경할 수 있습니다.

TextWrapper 인스턴스 어트리뷰트(와 생성자에 대한 키워드 인자)는 다음과 같습니다:

width

(기본값: `70`) 래핑 된 줄의 최대 길이. 입력 텍스트에 *width*보다 긴 개별 단어가 없는 한, *TextWrapper*는 *width* 문자보다 긴 출력 줄이 없음을 보장합니다.

expand_tabs

(기본값: `True`) 참이면, *text*의 모든 탭 문자가 *text*의 *expandtabs()* 메서드를 사용하여 스페이스로 확장됩니다.

tabsize

(기본값: `8`) *expand_tabs*가 참이면, *text*의 모든 탭 문자는 현재 열과 주어진 탭 크기에 따라 0개 이상의 스페이스로 확장됩니다.

Added in version 3.3.

replace_whitespace

(기본값: `True`) 참이면, 탭 확장 후 래핑 전에, *wrap()* 메서드는 각 공백 문자를 단일 스페이스로 치환합니다. 치환되는 공백 문자는 다음과 같습니다: 탭, 줄 바꿈, 세로 탭, 폼 피드 및 캐리지 리턴 (`'\t\n\v\f\r'`).

> **참고**
>
> *expand_tabs*가 거짓이고 *replace_whitespace*가 참이면, 각 탭 문자는 단일 스페이스로 치환되는데, 탭 확장과는 다릅니다.

> **참고**
>
> *replace_whitespace*가 거짓이면, 줄 중간에 줄 바꿈이 나타나서 이상한 결과가 발생할 수 있습니다. 이러한 이유로, 텍스트는 (*str.splitlines()*나 유사한 것을 사용해서) 문단으로 분할한 후에 별도로 래핑해야 합니다.

drop_whitespace

(기본값: `True`) 참이면, 모든 줄의 처음과 끝의 공백(래핑 이후 들여쓰기 전)이 삭제됩니다. 문단 시작 부분의 공백은 공백이 아닌 것이 뒤에 오면 삭제되지 않습니다. 삭제되는 공백이 줄 전체를 차지하면, 줄 전체가 삭제됩니다.

initial_indent

(기본값: `''`) 래핑 된 출력의 첫 번째 줄 앞에 추가될 문자열입니다. 첫 번째 줄의 길이 계산에 포함됩니다. 빈 문자열은 들여 쓰지 않습니다.

subsequent_indent

(기본값: `''`) 첫 줄을 제외한 래핑 된 출력의 모든 줄 앞에 추가될 문자열입니다. 첫 번째 줄을 제외한 각 줄의 길이 계산에 포함됩니다.

fix_sentence_endings

(기본값: `False`) 참이면, *TextWrapper*는 문장의 끝을 감지하고 문장이 항상 정확히 두 개의 스페이스로 분리되도록 만들려고 합니다. 이것은 일반적으로 고정 폭 글꼴의 텍스트에 적합합니다. 그러나, 문장 감지 알고리즘은 불완전합니다: 문장 끝은 `'.'`, `'!'` 또는 `'?'` 중 하나가 뒤에 오고, `'"'`나 `"'"` 중 하나가 뒤따르는 것도 가능, 그 뒤에 스페이스가 오는 소문자로 구성된다고 가정합니다. 이 알고리즘의 한가지 문제는 다음에 나오는 "Dr." 와

```
[...] Dr. Frankenstein's monster [...]
```

다음에 나오는 "Spot." 사이의 차이점을 탐지할 수 없다는 것입니다

```
[...] See Spot. See Spot run [...]
```

*fix_sentence_endings*는 기본적으로 거짓입니다.

문장 감지 알고리즘은 "소문자" 의 정의에 `string.lowercase`에 의존하고, 같은 줄에서 문장을 분리하기 위해 마침표 뒤에 두 개의 스페이스를 사용하는 규칙을 따르므로, 영어 텍스트에만 적용됩니다.

break_long_words

(기본값: `True`) 참이면, *width*보다 긴 줄이 없도록 하기 위해, *width*보다 긴 단어를 분할합니다. 거짓이면, 긴 단어가 깨지지 않으며, 일부 줄이 *width*보다 길 수 있습니다. (*width*를 초과하는 양을 최소화하기 위해 긴 단어는 독립된 줄에 넣습니다.)

break_on_hyphens

(기본값: `True`) 참이면, 래핑이, 영어에서의 관례대로, 공백과 복합 단어의 하이픈 바로 뒤에서 발생합니다. 거짓이면, 공백만을 줄 바꿈을 위한 좋은 장소로 간주하지만, 진정한 분할되지 않는 단어를 원한다면 *break_long_words*를 거짓으로 설정해야 합니다. 이전 버전의 기본 동작은 항상 하이픈으로 연결된 단어를 분리 할 수 있게 하는 것이었습니다.

max_lines

(기본값: `None`) `None`이 아니면, 출력은 최대 *max_lines* 줄을 포함하고, *placeholder*가 출력 끝에 나타납니다.

Added in version 3.4.

placeholder

(기본값: `' [...]'`) 잘렸을 때 출력 텍스트의 끝에 표시할 문자열.

Added in version 3.4.

*TextWrapper*는 모듈 수준 편리 함수와 유사한 몇 가지 공용 메서드도 제공합니다:

wrap(*text*)

text(문자열)에 있는 한 문단을 모든 줄의 길이가 최대 *width* 자가 되도록 래핑합니다. 모든 래핑 옵션은 *TextWrapper* 인스턴스의 인스턴스 어트리뷰트에서 가져옵니다. 최종 줄 바꿈이 없는 출력 줄의 리스트를 반환합니다. 래핑 된 출력에 내용이 없으면 반환된 리스트는 비어 있습니다.

fill(*text*)

*text*에 있는 단일 문단을 래핑하고, 래핑 된 문단을 포함하는 단일 문자열을 반환합니다.

6.6 unicodedata — 유니코드 데이터베이스

This module provides access to the Unicode Character Database (UCD) which defines character properties for all Unicode characters. The data contained in this database is compiled from the UCD version 16.0.0.

모듈은 유니코드 표준 부속서 #44, "유니코드 문자 데이터베이스"에 정의된 것과 같은 이름과 기호를 사용합니다. 다음과 같은 함수를 정의합니다:

더 보기

The unicode-howto for more information about Unicode and how to use this module.

`unicodedata.lookup(name)`

Look up character by name. If a character with the given name is found, return the corresponding character. If not found, `KeyError` is raised. For example:

```
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{'
```

The characters returned by this function are the same as those produced by `\N` escape sequence in string literals. For example:

```
>>> unicodedata.lookup('MIDDLE DOT') == '\N{MIDDLE DOT}'
True
```

버전 3.3에서 변경: 이름 별칭[1] 과 명명된 시퀀스[2] 가 추가되었습니다.

`unicodedata.name(chr, default=None, /)`

Returns the name assigned to the character *chr* as a string. If no name is defined, *default* is returned, or, if not given, `ValueError` is raised. For example:

```
>>> unicodedata.name('½')
'VULGAR FRACTION ONE HALF'
>>> unicodedata.name('\uFFFF', 'fallback')
'fallback'
```

`unicodedata.decimal(chr, default=None, /)`

Returns the decimal value assigned to the character *chr* as integer. If no such value is defined, *default* is returned, or, if not given, `ValueError` is raised. For example:

```
>>> unicodedata.decimal('\N{ARABIC-INDIC DIGIT NINE}')
9
>>> unicodedata.decimal('\N{SUPERSCRIPT NINE}', -1)
-1
```

`unicodedata.digit(chr, default=None, /)`

Returns the digit value assigned to the character *chr* as integer. If no such value is defined, *default* is returned, or, if not given, `ValueError` is raised:

```
>>> unicodedata.digit('\N{SUPERSCRIPT NINE}')
9
```

`unicodedata.numeric(chr, default=None, /)`

Returns the numeric value assigned to the character *chr* as float. If no such value is defined, *default* is returned, or, if not given, `ValueError` is raised:

```
>>> unicodedata.numeric('½')
0.5
```

`unicodedata.category(chr)`

Returns the general category assigned to the character *chr* as string. General category names consist of two letters. See the General Category Values section of the Unicode Character Database documentation for a list of category codes. For example:

[1] https://www.unicode.org/Public/16.0.0/ucd/NameAliases.txt
[2] https://www.unicode.org/Public/16.0.0/ucd/NamedSequences.txt

```
>>> unicodedata.category('A')  # 'L'etter, 'u'ppercase
'Lu'
```

unicodedata.**bidirectional**(*chr*)

Returns the bidirectional class assigned to the character *chr* as string. If no such value is defined, an empty string is returned. See the Bidirectional Class Values section of the Unicode Character Database documentation for a list of bidirectional codes. For example:

```
>>> unicodedata.bidirectional('\N{ARABIC-INDIC DIGIT SEVEN}') # 'A'rabic, 'N
↪'umber
'AN'
```

unicodedata.**combining**(*chr*)

Returns the canonical combining class assigned to the character *chr* as integer. Returns `0` if no combining class is defined. See the Canonical Combining Class Values section of the Unicode Character Database for more information.

unicodedata.**east_asian_width**(*chr*)

Returns the east asian width assigned to the character *chr* as string. For a list of widths and or more information, see the Unicode Standard Annex #11.

unicodedata.**mirrored**(*chr*)

Returns the mirrored property assigned to the character *chr* as integer. Returns `1` if the character has been identified as a "mirrored" character in bidirectional text, `0` otherwise. For example:

```
>>> unicodedata.mirrored('>')
1
```

unicodedata.**decomposition**(*chr*)

Returns the character decomposition mapping assigned to the character *chr* as string. An empty string is returned in case no such mapping is defined. For example:

```
>>> unicodedata.decomposition('Ã')
'0041 0303'
```

unicodedata.**normalize**(*form*, *unistr*)

유니코드 문자열 *unistr*에 대한 정규화 형식(normal form) *form*을 반환합니다. *form*의 유효한 값은 'NFC', 'NFKC', 'NFD' 및 'NFKD' 입니다.

유니코드 표준은 정준 동등성(canonical equivalence) 및 호환 동등성(compatibility equivalence)의 정의를 기반으로, 유니코드 문자열의 다양한 정규화 형식을 정의합니다. 유니코드에서, 여러 문자를 다양한 방법으로 표현할 수 있습니다. 예를 들어, U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA)은 시퀀스 U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA) 로도 표현할 수 있습니다.

각 문자에는, 두 개의 정규화 형식이 있습니다: 정규화 형식 C와 정규화 형식 D. 정규화 형식 D(NFD)는 정준 분해라고도 하며, 각 문자를 분해된 형식으로 변환합니다. 정규화 형식 C(NFC)는 먼저 정준 분해를 적용한 다음, 미리 결합한 문자로 다시 조합합니다.

In addition to these two forms, there are two additional normal forms based on compatibility equivalence. In Unicode, certain characters are supported which normally would be unified with other characters. For example, U+2160 (ROMAN NUMERAL ONE) is really the same thing as U+0049 (LATIN CAPITAL LETTER I). However, it is supported in Unicode for compatibility with existing character sets (for example, gb2312).

The normal form KD (NFKD) will apply the compatibility decomposition, that is, replace all compatibility characters with their equivalents. The normal form KC (NFKC) first applies the compatibility decomposition, followed by the canonical composition.

두 개의 유니코드 문자열이 정규화되고, 사람이 보기에 같아 보여도, 하나가 결합한 문자를 갖고 다른 것은 그렇지 않으면, 같다고 비교되지 않을 수 있습니다.

unicodedata.**is_normalized**(*form*, *unistr*)

유니코드 문자열 *unistr*이 정규화 형식 *form*인지를 반환합니다. *form*의 유효한 값은 'NFC', 'NFKC', 'NFD' 및 'NFKD' 입니다.

Added in version 3.8.

또한, 이 모듈은 다음 상수를 노출합니다:

unicodedata.**unidata_version**

이 모듈에 사용된 유니코드 데이터베이스의 버전.

unicodedata.**ucd_3_2_0**

이것은 전체 모듈과 같은 메서드를 가지고 있는 객체이지만, 유니코드 데이터베이스 버전 3.2를 대신 사용합니다. 이 특정 버전의 유니코드 데이터베이스가 필요한 응용 프로그램(가령 IDNA)을 위한 것입니다.

6.7 stringprep — 인터넷 문자열 준비

소스 코드: Lib/stringprep.py

인터넷에서 무언가(가령 호스트 이름)를 식별할 때, 종종 그러한 식별에 대해 "동등" 비교할 필요가 있습니다. 이 비교가 실행되는 정확한 방법은 응용 프로그램 도메인에 따라 달라질 수 있습니다, 예를 들어 대/소문자를 구분하는지 그렇지 않은지. 또한 "인쇄 가능" 문자로만 구성된 식별만 허용하기 위해, 가능한 식별을 제한해야 할 수도 있습니다.

RFC 3454는 인터넷 프로토콜에서 유니코드 문자열을 "준비" 하는 절차를 정의합니다. 문자열을 전선에 전달하기 전에, 준비 절차를 통해 문자열을 처리해서 어떤 정규화된 형식을 갖도록 만듭니다. RFC는 프로파일로 결합할 수 있는 테이블 집합을 정의합니다. 각 프로파일은 사용하는 테이블과 stringprep 절차의 어떤 선택적 부분이 프로파일 일부인지 정의해야 합니다. stringprep 프로파일의 한 가지 예는 국제화된 도메인 이름에 사용되는 nameprep 입니다.

stringprep 모듈은 **RFC 3454**의 테이블만 노출합니다. 이러한 테이블은 딕셔너리나 리스트로 표현하기에 매우 크기 때문에, 모듈은 내부적으로 유니코드 문자 데이터베이스를 사용합니다. 모듈 소스 코드 자체는 mkstringprep.py 유틸리티를 사용하여 생성되었습니다.

결과적으로, 이러한 테이블은 데이터 구조가 아닌 함수로 노출됩니다. RFC에는 두 종류의 테이블이 있습니다: 집합과 매핑. 집합의 경우, *stringprep*는 "특성 함수", 즉 매개 변수가 집합 일부면 True를 반환하는 함수를 제공합니다. 매핑의 경우, 매핑 함수를 제공합니다: 주어진 키에 대해, 연관된 값을 반환합니다. 다음은 모듈에서 사용할 수 있는 모든 함수의 목록입니다.

stringprep.**in_table_a1**(*code*)

*code*가 tableA.1(유니코드 3.2에서 지정되지 않은 코드 포인트)에 있는지 판별합니다.

stringprep.**in_table_b1**(*code*)

*code*가 tableB.1(일반적으로 아무것도 매핑되지 않습니다)에 있는지 판별합니다.

stringprep.**map_table_b2**(*code*)

tableB.2(NFKC와 함께 사용되는 케이스 폴딩용 매핑)에 따라 *code*의 매핑 된 값을 반환합니다.

stringprep.**map_table_b3**(*code*)

tableB.3(정규화가 없는 케이스 폴딩용 매핑)에 따라 *code*의 매핑 된 값을 반환합니다.

stringprep.**in_table_c11**(*code*)

*code*가 tableC.1.1(ASCII 스페이스 문자)에 있는지 판별합니다.

stringprep.**in_table_c12**(*code*)

*code*가 tableC.1.2(비 ASCII 스페이스 문자)에 있는지 판별합니다.

stringprep.**in_table_c11_c12**(*code*)

*code*가 tableC.1(스페이스 문자, C.1.1과 C.1.2의 합집합)에 있는지 판별합니다.

stringprep.**in_table_c21**(*code*)

*code*가 tableC.2.1(ASCII 제어 문자)에 있는지 판별합니다.

stringprep.**in_table_c22**(*code*)

*code*가 tableC.2.2(비 ASCII 제어 문자)에 있는지 판별합니다.

stringprep.**in_table_c21_c22**(*code*)

*code*가 tableC.2(제어 문자, C.2.1과 C.2.2의 합집합)에 있는지 판별합니다.

stringprep.**in_table_c3**(*code*)

*code*가 tableC.3(개인 사용)에 있는지 판별합니다.

stringprep.**in_table_c4**(*code*)

*code*가 tableC.4(비문자 코드 포인트)에 있는지 판별합니다.

stringprep.**in_table_c5**(*code*)

*code*가 tableC.5(대리 코드)에 있는지 판별합니다.

stringprep.**in_table_c6**(*code*)

*code*가 tableC.6(일반 텍스트에는 부적절)에 있는지 판별합니다.

stringprep.**in_table_c7**(*code*)

*code*가 tableC.7(규범적 표현에는 부적절)에 있는지 판별합니다.

stringprep.**in_table_c8**(*code*)

*code*가 tableC.8(표시 특성 변경 또는 폐지)에 있는지 판별합니다.

stringprep.**in_table_c9**(*code*)

*code*가 tableC.9(문자 태깅)에 있는지 판별합니다.

stringprep.**in_table_d1**(*code*)

*code*가 tableD.1(양방향 특성이 "R" 이나 "AL" 인 문자)에 있는지 판별합니다.

stringprep.**in_table_d2**(*code*)

*code*가 tableD.2(양방향 특성이 "L" 인 문자)에 있는지 판별합니다.

6.8 readline — GNU readline 인터페이스

readline 모듈은 파이썬 인터프리터에서 완성(completion)과 히스토리 파일의 읽기/쓰기를 용이하게 하는 여러 함수를 정의합니다. 이 모듈은 직접 사용하거나, 대화식 프롬프트에서 파이썬 식별자 완성을 지원하는 *rlcompleter* 모듈을 통해 사용할 수 있습니다. 이 모듈을 사용하여 설정한 내용은 인터프리터의 대화식 프롬프트와 내장 *input()* 함수가 제공하는 프롬프트의 동작에 영향을 줍니다.

Readline 키 바인딩은 초기화 파일을 통해 구성할 수 있습니다, 일반적으로 홈 디렉터리의 `.inputrc`. 이 파일의 형식과 허용되는 구성 및 Readline 라이브러리의 기능에 대한 일반적인 정보는 GNU Readline 매뉴얼의 Readline Init File을 참조하십시오.

가용성: not Android, not iOS, not WASI.

이 모듈은 모바일 플랫폼이나 웹어셈블리 플랫폼에서 지원되지 않습니다.

참고

하부 Readline 라이브러리 API는 GNU readline 대신 `editline` (`libedit`) 라이브러리로 구현될 수 있습니다. macOS에서 *readline* 모듈은 실행 시간에 사용 중인 라이브러리를 감지합니다.

`editline`의 구성 파일은 GNU readline의 구성 파일과 다릅니다. 프로그래밍 방식으로 구성 문자열을 로드하는 경우 어떤 라이브러리가 사용되고 있는지 *backend*를 사용하여 확인할 수 있습니다.

macOS에서 `editline/libedit` readline 에뮬레이션을 사용하는 경우, 홈 디렉터리에 있는 초기화 파일의 이름은 `.editrc`입니다. 예를 들어, `~/.editrc`의 다음 내용은 *vi* 키 바인딩과 TAB 완성을 켭니다:

```
python:bind -v
python:bind ^I rl_complete
```

Also note that different libraries may use different history file formats. When switching the underlying library, existing history files may become unusable.

`readline.`**`backend`**

The name of the underlying Readline library being used, either `"readline"` or `"editline"`.

Added in version 3.13.

6.8.1 초기화 파일

다음 함수는 초기화 파일 및 사용자 구성과 관련이 있습니다:

`readline.`**`parse_and_bind`**(*string*)

string 인자에 제공된 초기화 줄을 실행합니다. 하부 라이브러리에서 `rl_parse_and_bind()`를 호출합니다.

`readline.`**`read_init_file`**([*filename*])

Execute a readline initialization file. The default filename is the last filename used. This calls `rl_read_init_file()` in the underlying library. It raises an *auditing event* `open` with the file name if given, and `"<readline_init_file>"` otherwise, regardless of which file the library resolves.

버전 3.14에서 변경: The auditing event was added.

6.8.2 줄 버퍼

다음 함수는 라인 버퍼에 대해 작용합니다:

`readline.`**`get_line_buffer`**()

줄 버퍼의 현재 내용(하부 라이브러리의 `rl_line_buffer`)을 반환합니다.

`readline.`**`insert_text`**(*string*)

줄 버퍼의 커서 위치에 텍스트를 삽입합니다. 하부 라이브러리에서 `rl_insert_text()`를 호출하지만, 반환 값은 무시합니다.

`readline.`**`redisplay`**()

줄 버퍼의 현재 내용을 반영하도록 화면에 표시되는 내용을 변경합니다. 하부 라이브러리에서 `rl_redisplay()`를 호출합니다.

6.8.3 히스토리 파일

다음 함수는 히스토리 파일에 대해 작용합니다:

`readline.`**`read_history_file`**([*filename*])

Load a readline history file, and append it to the history list. The default filename is `~/.history`. This calls `read_history()` in the underlying library and raises an *auditing event* `open` with the file name if given and `"~/.history"` otherwise.

버전 3.14에서 변경: The auditing event was added.

`readline.`**`write_history_file`**([*filename*])

Save the history list to a readline history file, overwriting any existing file. The default filename is `~/.history`. This calls `write_history()` in the underlying library and raises an *auditing event* `open` with the file name if given and `"~/.history"` otherwise.

버전 3.14에서 변경: The auditing event was added.

readline.**append_history_file**(*nelements*[, *filename*])

Append the last *nelements* items of history to a file. The default filename is `~/.history`. The file must already exist. This calls `append_history()` in the underlying library. This function only exists if Python was compiled for a version of the library that supports it. It raises an *auditing event* `open` with the file name if given and `"~/.history"` otherwise.

Added in version 3.5.

버전 3.14에서 변경: The auditing event was added.

readline.**get_history_length**()

readline.**set_history_length**(*length*)

히스토리 파일에 저장하기 원하는 줄 수를 설정하거나 반환합니다. *write_history_file()* 함수는 이 값을 사용하여, 하부 라이브러리에서 `history_truncate_file()` 을 호출하여 히스토리 파일을 자릅니다. 음수 값은 제한 없는 히스토리 파일 크기를 의미합니다.

6.8.4 히스토리 목록

다음 함수는 전역 히스토리 목록에 대해 작용합니다:

readline.**clear_history**()

현재 히스토리를 지웁니다. 하부 라이브러리에서 `clear_history()` 를 호출합니다. 파이썬 함수는 파이썬이 이를 지원하는 라이브러리 버전으로 컴파일된 경우에만 존재합니다.

readline.**get_current_history_length**()

현재 히스토리에 있는 항목 수를 반환합니다. (이것은 히스토리 파일에 기록될 최대 줄 수를 반환하는 *get_history_length()* 와 다릅니다.)

readline.**get_history_item**(*index*)

index 에 있는 히스토리 항목의 현재 내용을 반환합니다. 항목 인덱스는 1부터 시작합니다. 하부 라이브러리에서 `history_get()` 을 호출합니다.

readline.**remove_history_item**(*pos*)

히스토리에서 위치(pos)로 지정된 히스토리 항목을 제거합니다. 위치는 0부터 시작합니다. 하부 라이브러리에서 `remove_history()` 를 호출합니다.

readline.**replace_history_item**(*pos*, *line*)

위치(pos)로 지정된 히스토리 항목을 *line* 으로 교체합니다. 위치는 0부터 시작합니다. 하부 라이브러리에서 `replace_history_entry()` 를 호출합니다.

readline.**add_history**(*line*)

마지막 줄이 입력된 것처럼 히스토리 버퍼에 *line* 을 추가합니다. 하부 라이브러리에서 `add_history()` 를 호출합니다.

readline.**set_auto_history**(*enabled*)

readline 을 통해 입력을 읽을 때 `add_history()` 에 대한 자동 호출을 활성화 또는 비활성화합니다. *enabled* 인자는 참일 때 자동 히스토리를 활성화하고, 거짓일 때 자동 기록을 비활성화하는 불리언 값이어야 합니다.

Added in version 3.6.

CPython 구현 상세: Auto history is enabled by default, and changes to this do not persist across multiple sessions.

6.8.5 시동 훅

readline.**set_startup_hook**([*function*])

하부 라이브러리의 `rl_startup_hook` 콜백에 의해 호출되는 함수를 설정하거나 제거합니다. *function* 이 지정되면 새 훅(hook) 함수로 사용됩니다; 생략되거나 `None` 이면, 이미 설치된 함수가 제거됩니다. 이 훅은 readline 이 첫 번째 프롬프트를 인쇄하기 직전에 인자 없이 호출됩니다.

readline.**set_pre_input_hook**([*function*])

하부 라이브러리의 rl_pre_input_hook 콜백에 의해 호출되는 함수를 설정하거나 제거합니다. *function*이 지정되면, 새 훅 함수로 사용됩니다; 생략되거나 None이면, 이미 설치된 함수가 제거됩니다. 이 훅은 첫 번째 프롬프트가 인쇄된 후 readline이 입력 문자를 읽기 시작하기 직전에 인자 없이 호출됩니다. 이 함수는 파이썬이 이를 지원하는 라이브러리 버전으로 컴파일된 경우에만 존재합니다.

6.8.6 완성

다음 함수는 사용자 정의 단어 완성 기능 구현과 관련이 있습니다. 이것은 일반적으로 Tab 키로 작동하며, 입력되는 단어를 제안하고 자동으로 완성할 수 있습니다. 기본적으로, Readline은 대화식 인터프리터를 위해 파이썬 식별자를 완성하는 *rlcompleter*에서 사용하도록 설정되어 있습니다. *readline* 모듈을 사용자 정의 완성기와 함께 사용하려면, 다른 단어 구분자 집합을 설정해야 합니다.

readline.**set_completer**([*function*])

완성 함수를 설정하거나 제거합니다. *function*이 지정되면 새 완성 함수로 사용됩니다; 생략하거나 None이면, 이미 설치된 완성 함수가 제거됩니다. 완성 함수는 문자열이 아닌 값을 반환할 때까지 0, 1, 2 등의 *state*에 대해 function(text, state)로 호출됩니다. *text*로 시작하는 다음으로 가능한 완성을 반환해야 합니다.

설치된 완성 함수는 하부 라이브러리의 rl_completion_matches()로 전달된 *entry_func* 콜백에 의해 호출됩니다. *text* 문자열은 하부 라이브러리의 rl_attempted_completion_function 콜백의 첫 번째 매개 변수로부터 옵니다.

readline.**get_completer**()

완성 함수나, 완성 함수가 설정되지 않았으면 None을 얻습니다.

readline.**get_completion_type**()

시도 중인 완성 유형을 가져옵니다. 하부 라이브러리의 rl_completion_type 변수를 정수로 반환합니다.

readline.**get_begidx**()

readline.**get_endidx**()

완성 범위(completion scope)의 시작이나 끝 인덱스를 가져옵니다. 이 인덱스는 하부 라이브러리의 rl_attempted_completion_function 콜백에 전달된 *start*와 *end* 인자입니다. 같은 입력 편집 시나리오에서 하부 C readline 구현에 따라 값이 다를 수 있습니다. 예: libedit는 libreadline과 다르게 동작하는 것으로 알려져 있습니다.

readline.**set_completer_delims**(*string*)

readline.**get_completer_delims**()

완성을 위한 단어 구분자를 설정하거나 가져옵니다. 이것들은 완성을 위해 고려할 단어의 시작(완성 범위)을 결정합니다. 이 함수는 하부 라이브러리의 rl_completer_word_break_characters 변수를 액세스합니다.

readline.**set_completion_display_matches_hook**([*function*])

완성 표시 함수를 설정하거나 제거합니다. *function*이 지정되면, 새로운 완성 표시 함수로 사용됩니다; 생략하거나 None이면, 이미 설치된 완성 표시 함수가 제거됩니다. 하부 라이브러리에서 rl_completion_display_matches_hook 콜백을 설정하거나 지웁니다. 완성 표시 함수는 일치를 표시해야 할 때마다 한 번 function(substitution, [matches], longest_match_length)로 호출됩니다.

6.8.7 예제

다음 예는 *readline* 모듈의 히스토리 읽기와 쓰기 함수를 사용하여 사용자의 홈 디렉터리에서 .python_history라는 이름의 히스토리 파일을 자동으로 로드하고 저장하는 방법을 보여줍니다. 아래 코드는 일반적으로 사용자의 PYTHONSTARTUP 파일에서 대화식 세션 중에 자동으로 실행됩니다.

```
import atexit
import os
import readline
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
histfile = os.path.join(os.path.expanduser("~"), ".python_history")
try:
    readline.read_history_file(histfile)
    # 기본 히스토리 길이는 -1(무한)이며, 제약 없이 커질 수 있습니다
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)
```

이 코드는 실제로 파이썬이 대화형 모드로 실행될 때 자동으로 실행됩니다 (*Readline 구성*을 참조하십시오).

다음 예는 같은 목표를 달성하지만 새 히스토리를 덧붙이기만 해서 동시적인(concurrent) 대화형 세션을 지원합니다.

```
import atexit
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".python_history")

try:
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
    open(histfile, 'wb').close()
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
    readline.append_history_file(new_h_len - prev_h_len, histfile)
atexit.register(save, h_len, histfile)
```

다음 예는 히스토리 저장/복원을 지원하도록 `code.InteractiveConsole` 클래스를 확장합니다.

```
import atexit
import code
import os
import readline

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

6.9 rlcompleter — GNU readline을 위한 완성 함수

소스 코드: Lib/rlcompleter.py

---

rlcompleter 모듈은 *readline* 모듈의 *set_completer()*로 전달하기에 적합한 완성 함수를 정의합니다.

readline 모듈을 사용할 수 있는 유닉스 플랫폼에서 이 모듈이 임포트될 때, *Completer* 클래스의 인스턴스가 자동으로 만들어지고, *complete()* 메서드가 *readline 완성기*로 설정됩니다. 이 메서드는 유효한 파이썬 식별자와 키워드의 완성을 제공합니다.

예제:

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__          readline.get_line_buffer(  readline.read_init_file(
readline.__file__         readline.insert_text(      readline.set_completer(
readline.__name__         readline.parse_and_bind(
>>> readline.
```

rlcompleter 모듈은 파이썬의 대화형 모드와 함께 사용하도록 설계되었습니다. 파이썬이 -S 옵션으로 실행되지 않는 한, 모듈은 자동으로 임포트되고 구성됩니다 (*Readline 구성*를 보세요).

*readline*이 없는 플랫폼에서, 이 모듈이 정의하는 *Completer* 클래스는 여전히 사용자 정의 목적에 사용될 수 있습니다.

class rlcompleter.**Completer**

Completer 객체는 다음과 같은 메서드를 가집니다:

complete(*text*, *state*)

*text*에 대한 가능한 다음 완성을 반환합니다.

When called by the *readline* module, this method is called successively with `state == 0, 1, 2, ...` until the method returns `None`.

마침표('.')가 포함되지 않은 *text*로 호출되면, *__main__*, *builtins* 및 키워드(*keyword* 모듈에서 정의한 대로)에 현재 정의된 이름으로 완성됩니다.

점으로 구분된 이름으로 호출하면, 명백한 부작용(함수는 평가되지 않지만 __getattr__() 에 대한 호출을 만들 수 있습니다)없이 마지막 부분까지 평가하려고 시도하고, 나머지는 *dir()* 함수를 통해 일치하는 것을 찾습니다. 표현식을 평가하는 동안 발생하는 모든 예외는 잡히고, 억제하며 *None*을 반환합니다.

CHAPTER 7

바이너리 데이터 서비스

이 장에서 설명하는 모듈은 바이너리 데이터를 다루기 위한 기본 서비스 연산을 제공합니다. 파일 포맷, 네트워크 프로토콜과 관련 있는 바이너리 데이터에 대한 연산은 관련 절에서 설명합니다.

텍스트 처리 서비스에서 설명한 일부 라이브러리는 ASCII 호환 바이너리 형식 (예를 들면, `re`) 또는 모든 바이너리 데이터 (예를 들면, `difflib`)에 사용할 수 있습니다.

또한, 파이썬 내장 바이너리 데이터형에 대한 설명은 바이너리 시퀀스 형 — *bytes, bytearray, memoryview* 문서를 참고하세요.

7.1 `struct` — 패킹 된 바이너리 데이터로 바이트열을 해석

소스 코드: Lib/struct.py

This module converts between Python values and C structs represented as Python `bytes` objects. Compact *format strings* describe the intended conversions to/from Python values. The module' s functions and objects can be used for two largely distinct applications, data exchange with external sources (files or network connections), or data transfer between the Python application and the C layer.

> **참고**
>
> When no prefix character is given, native mode is the default. It packs or unpacks data based on the platform and compiler on which the Python interpreter was built. The result of packing a given C struct includes pad bytes which maintain proper alignment for the C types involved; similarly, alignment is taken into account when unpacking. In contrast, when communicating data between external sources, the programmer is responsible for defining byte ordering and padding between elements. See 바이트 순서, 크기 및 정렬 for details.

여러 `struct` 함수(그리고 `Struct` 의 메서드)는 *buffer* 인자를 취합니다. 이는 bufferobjects을 구현하고 읽을 수 있거나 읽고 쓸 수 있는 버퍼를 제공하는 객체를 나타냅니다. 이 목적으로 사용되는 가장 일반적인 형은 `bytes`와 `bytearray`지만, 바이트 배열로 볼 수 있는 많은 다른 형이 버퍼 프로토콜을 구현하므로, `bytes` 객체에서 추가로 복사하지 않고도 읽고 채울 수 있습니다.

7.1.1 함수와 예외

이 모듈은 다음과 같은 예외와 함수를 정의합니다:

exception struct.**error**

여러 상황에서 발생하는 예외; 인자는 무엇이 잘못되었는지 설명하는 문자열입니다.

struct.**pack**(*format, v1, v2, ...*)

v1, v2, … 값을 포함하고 포맷 문자열 *format*에 따라 패킹 된 바이트열 객체를 반환합니다. 인자는 포맷이 요구하는 값과 정확히 일치해야 합니다.

struct.**pack_into**(*format, buffer, offset, v1, v2, ...*)

포맷 문자열 *format*에 따라 값 *v1, v2, …* 를 패킹하고 패킹 된 바이트열을 쓰기 가능한 버퍼 *buffer*에 *offset* 위치에서부터 씁니다. *offset*은 필수 인자임에 유의하십시오.

struct.**unpack**(*format, buffer*)

포맷 문자열 *format*에 따라 버퍼 *buffer*(아마도 `pack(format, ...)`으로 패킹 된)에서 언 패킹 합니다. 정확히 하나의 항목을 포함하더라도 결과는 튜플입니다. 바이트 단위의 버퍼 크기는 (`calcsize()`에 의해 반영되는) 포맷이 요구하는 크기와 일치해야 합니다.

struct.**unpack_from**(*format, /, buffer, offset=0*)

포맷 문자열 *format*에 따라, *offset* 위치에서 시작하여 *buffer*에서 언 패킹 합니다. 정확히 하나의 항목을 포함하더라도 결과는 튜플입니다. *offset* 위치에서 시작하여 바이트 단위로 측정한 버퍼 크기는 (`calcsize()`에 의해 반영되는) 포맷이 요구하는 크기 이상이어야 합니다.

struct.**iter_unpack**(*format, buffer*)

포맷 문자열 *format*에 따라 버퍼 *buffer*에서 이터레이션을 통해 언 패킹 합니다. 이 함수는 모든 내용이 소비될 때까지 버퍼에서 같은 크기의 청크를 읽는 이터레이터를 반환합니다. 바이트 단위의 버퍼 크기는 (`calcsize()`에 의해 반영되는) 포맷이 요구하는 크기의 배수여야 합니다.

각 이터레이션은 포맷 문자열에 지정된 대로 튜플을 산출합니다.

Added in version 3.4.

struct.**calcsize**(*format*)

포맷 문자열 *format*에 해당하는 구조체(`pack(format, ...)`에 의해 생성되는 바이트열 객체)의 크기를 반환합니다.

7.1.2 포맷 문자열

포맷 문자열은 데이터를 패킹과 언 패킹할 때 데이터 배치를 설명합니다. 이들은 패킹/언 패킹 될 데이터형을 지정하는 포맷 문자로 구축됩니다. 또한, 특수 문자가 바이트 순서, 크기 및 정렬을 제어합니다. 각 포맷 문자열은 데이터의 전체 속성을 설명하는 선택적 접두사 문자와 실제 데이터 값과 패딩을 설명하는 하나 이상의 포맷 문자로 구성됩니다.

바이트 순서, 크기 및 정렬

기본적으로, C형은 기계의 네이티브 형식과 바이트 순서로 표현되며, 필요하면 (C 컴파일러에서 사용하는 규칙에 따라) 패드 바이트로 건너뛰어 적절하게 정렬됩니다. 이 동작은 패킹된 구조체의 바이트가 해당 C 구조체의 메모리 배치와 정확히 일치하도록 선택되었습니다. 네이티브 바이트 순서와 패딩을 사용할지 아니면 표준 포맷을 사용할지는 응용 프로그램에 달려 있습니다.

또는, 다음 표에 따라, 포맷 문자열의 첫 번째 문자를 사용하여 패킹 된 데이터의 바이트 순서, 크기 및 정렬을 표시할 수 있습니다:

문자	바이트 순서	크기	정렬
@	네이티브	네이티브	네이티브
=	네이티브	표준	none
<	리틀 엔디안	표준	none
>	빅 엔디안	표준	none
!	네트워크 (= 빅 엔디안)	표준	none

첫 번째 문자가 이들 중 하나가 아니면, `'@'`로 가정합니다.

> **참고**
>
> The number 1023 (`0x3ff` in hexadecimal) has the following byte representations:
>
> - `03 ff` in big-endian (>)
> - `ff 03` in little-endian (<)
>
> 파이썬 예제:
>
> ```
> >>> import struct
> >>> struct.pack('>h', 1023)
> b'\x03\xff'
> >>> struct.pack('<h', 1023)
> b'\xff\x03'
> ```

네이티브 바이트 순서는 호스트 시스템에 따라 빅 엔디안이나 리틀 엔디안입니다. 예를 들어, 인텔 x86, AMD64 (x86-64) 및 애플 M1은 리틀 엔디안입니다; IBM z 와 많은 레거시 아키텍처는 빅 엔디안입니다. 시스템의 엔디안을 확인하려면 *sys.byteorder*를 사용하십시오.

네이티브 크기와 정렬은 C 컴파일러의 `sizeof` 표현식을 사용하여 결정됩니다. 이것은 항상 네이티브 바이트 순서와 결합합니다.

표준 크기는 포맷 문자에만 의존합니다; 포맷 문자 섹션의 표를 참조하십시오.

`'@'`과 `'='`의 차이점에 유의하십시오; 둘 다 네이티브 바이트 순서를 사용하지만, 후자는 크기와 정렬이 표준화됩니다.

`'!'` 형식은 IETF RFC 1700에 정의된 대로 항상 빅 엔디안인 네트워크 바이트 순서를 나타냅니다.

네이티브가 아닌 바이트 순서(강제 바이트 스와핑)를 표시하는 방법은 없습니다; `'<'`나 `'>'`를 적절히 선택하십시오.

노트:

(1) 패딩은 연속되는 구조체 멤버 간에만 자동으로 추가됩니다. 인코딩된 구조체의 시작이나 끝에는 패딩이 추가되지 않습니다.

(2) 네이티브가 아닌 크기와 정렬을 사용할 때는 패딩이 추가되지 않습니다, 예를 들어 '<', '>', '=' 및 '!' 에서.

(3) 구조체의 끝을 특정 형의 정렬 요구 사항에 맞추려면, 반복 횟수가 0인 해당 형의 코드로 포맷을 끝내십시오. 예를 참조하십시오.

포맷 문자

포맷 문자는 다음과 같은 의미가 있습니다; C와 파이썬 값 사이의 변환은 형을 주면 분명해야 합니다. '표준 크기' 열은 표준 크기를 사용할 때 패킹 된 값의 크기를 바이트 단위로 나타냅니다; 즉, 포맷 문자열이 `'<'`, `'>'`, `'!'` 또는 `'='` 중 하나로 시작하는 경우입니다. 네이티브 크기를 사용할 때, 패킹 된 값의 크기는 플랫폼에 따라 다릅니다.

포맷	C형	파이썬 형	표준 크기	노트
x	패드 바이트	값이 없습니다		(7)
c	char	길이가 1 인 bytes	1	
b	signed char	정수	1	(1), (2)
B	unsigned char	정수	1	(2)
?	_Bool	bool	1	(1)
h	short	정수	2	(2)
H	unsigned short	정수	2	(2)
i	int	정수	4	(2)
I	unsigned int	정수	4	(2)
l	long	정수	4	(2)
L	unsigned long	정수	4	(2)
q	long long	정수	8	(2)
Q	unsigned long long	정수	8	(2)
n	ssize_t	정수		(3)
N	size_t	정수		(3)
e	(6)	float	2	(4)
f	float	float	4	(4)
d	double	float	8	(4)
F	float complex	complex	8	(10)
D	double complex	complex	16	(10)
s	char[]	bytes		(9)
p	char[]	bytes		(8)
P	void*	정수		(5)

버전 3.3에서 변경: `'n'`과 `'N'` 포맷에 대한 지원이 추가되었습니다.

버전 3.6에서 변경: `'e'` 포맷에 대한 지원이 추가되었습니다.

버전 3.14에서 변경: Added support for the `'F'` and `'D'` formats.

노트:

(1) `'?'` 변환 코드는 C99 이후의 C 표준이 정의한 `_Bool` 형에 해당합니다. 표준 모드에서는, 1 바이트로 표현됩니다.

(2) 정수 변환 코드 중 하나를 사용하여 정수가 아닌 값을 패킹하려고 할 때, 정수가 아닌 값에 `__index__()` 메서드가 있으면 패킹 전에 해당 메서드가 호출되어 인자를 정수로 변환합니다.

버전 3.2에서 변경: 정수가 아닌 값에서 `__index__()` 메서드를 사용하는 것을 추가했습니다.

(3) `'n'`과 `'N'` 변환 코드는 (기본값이나 `'@'` 바이트 순서 문자로 선택된) 네이티브 크기에만 사용할 수 있습니다. 표준 크기의 경우, 응용 프로그램에 맞는 다른 정수 포맷을 사용할 수 있습니다.

(4) `'f'`, `'d'` 및 `'e'` 변환 코드의 경우, 패킹 된 표현은 플랫폼에서 사용하는 부동 소수점 형식과 관계없이 IEEE 754 binary32, binary64 또는 binary16 형식을 사용합니다 (각각 `'f'`, `'d'` 또는 `'e'`).

(5) `'P'` 포맷 문자는 (기본값이나 `'@'` 바이트 순서 문자로 선택된) 네이티브 바이트 순서에만 사용할 수 있습니다. 바이트 순서 문자 `'='`는 호스트 시스템에 따라 리틀이나 빅 엔디안 순서를 사용하도록 선택합니다. struct 모듈은 이를 네이티브 순서로 해석하지 않아서, `'P'` 형식을 사용할 수 없습니다.

(6) IEEE 754 binary16 "반 정밀도" 형은 2008 년 IEEE 754 표준 개정판에서 도입되었습니다. 부호 비트, 5 비트 지수 및 11 비트 정밀도(10 비트가 명시적으로 저장됩니다)를 가지며, 전체 정밀도에서 대략 `6.1e-05`와 `6.5e+04` 사이의 숫자를 나타낼 수 있습니다. 이 형은 C 컴파일러에서 널리 지원되지 않습니다: 일반적인 기계에서는, unsigned short를 저장에 사용할 수 있지만, 수학 연산에는 사용할 수 없습니다. 자세한 내용은 half-precision floating-point format의 Wikipedia 페이지를 참조하십시오.

(7) When packing, `'x'` inserts one NUL byte.

(8) `'p'` 포맷 문자는 "파스칼 문자열"을 인코딩하는데, 이는 카운트에 의해 주어진 고정된 바이트 수에 저장된 짧은 가변 길이 문자열을 의미합니다. 저장된 첫 번째 바이트는 문자열의 길이나 255 중 작은 값입니다. 문자열의 바이트가 그 뒤에 옵니다. *pack()*에 전달된 문자열이 너무 길면 (횟수 빼기 1 보다

길면), 문자열의 선행 count-1 바이트만 저장됩니다. 문자열이 count-1보다 짧으면, 전부 정확한 바이트 수가 되도록 널 바이트로 채워집니다. *unpack()*의 경우, 'p' 포맷 문자는 count 바이트를 소비하지만, 반환된 문자열은 255바이트를 초과할 수 없음에 유의하십시오.

(9) 's' 포맷 문자의 경우, 횟수는 다른 포맷 문자와 같은 반복 횟수가 아닌 바이트열의 길이로 해석됩니다; 예를 들어, '10s'는 단일 파이썬 바이트 문자열과 양 방향 매핑되는 단일 10바이트 문자열을 의미하는 반면에, '10c'는 10개의 서로 다른 파이썬 바이트 객체와 양 방향 매핑되는 10개의 개별 바이트 문자 요소를 의미합니다 (예를 들어, cccccccccc). (차이점에 대한 구체적인 시연은 예를 참조하세요.) 횟수를 지정하지 않으면, 기본값은 1입니다. 패킹의 경우, 맞도록 문자열이 잘리거나 널 바이트로 채워집니다. 언 패킹의 경우, 결과 바이트열 객체는 항상 지정된 바이트 수를 갖습니다. 특별한 경우로, '0s'는 하나의 빈 문자열을 의미합니다(반면에 '0c'는 0문자를 의미합니다).

(10) For the 'F' and 'D' format characters, the packed representation uses the IEEE 754 binary32 and binary64 format for components of the complex number, regardless of the floating-point format used by the platform. Note that complex types (F and D) are available unconditionally, despite complex types being an optional feature in C. As specified in the C11 standard, each complex type is represented by a two-element C array containing, respectively, the real and imaginary parts.

포맷 문자 앞에는 정수 반복 횟수가 올 수 있습니다. 예를 들어, 포맷 문자열 '4h'는 'hhhh'와 정확히 같습니다.

포맷 사이의 공백 문자는 무시됩니다; 횟수와 형식 사이에는 공백이 없어야 합니다.

정수 형식 ('b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q') 중 하나를 사용하여 값 x를 패킹할 때, x가 해당 포맷의 유효한 범위를 벗어나면 *struct.error*가 발생합니다.

버전 3.1에서 변경: 이전에는, 일부 정수 포맷은 범위를 벗어난 값을 래핑하고 *struct.error* 대신 *DeprecationWarning*을 발생시켰습니다.

'?' 포맷 문자의 경우, 반환 값은 *True*나 *False*입니다. 패킹할 때, 인자 객체의 논리값이 사용됩니다. 네이티브나 표준 bool 표현에서 0이나 1이 패킹 되고, 언 패킹할 때 모든 0이 아닌 값은 True가 됩니다.

예

참고

Native byte order examples (designated by the '@' format prefix or lack of any prefix character) may not match what the reader' s machine produces as that depends on the platform and compiler.

Pack and unpack integers of three different sizes, using big endian ordering:

```
>>> from struct import *
>>> pack(">bhl", 1, 2, 3)
b'\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('>bhl', b'\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('>bhl')
7
```

Attempt to pack an integer which is too large for the defined field:

```
>>> pack(">h", 99999)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
struct.error: 'h' format requires -32768 <= number <= 32767
```

Demonstrate the difference between 's' and 'c' format characters:

```
>>> pack("@ccc", b'1', b'2', b'3')
b'123'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> pack("@3s", b'123')
b'123'
```

언 패킹 된 필드는 변수에 대입하거나 결과를 네임드 튜플로 감싸서 이름을 붙일 수 있습니다:

```
>>> record = b'raymond   \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name=b'raymond   ', serialnum=4658, school=264, gradelevel=8)
```

The ordering of format characters may have an impact on size in native mode since padding is implicit. In standard mode, the user is responsible for inserting any desired padding. Note in the first `pack` call below that three NUL bytes were added after the packed `'#'` to align the following integer on a four-byte boundary. In this example, the output was produced on a little endian machine:

```
>>> pack('@ci', b'#', 0x12131415)
b'#\x00\x00\x00\x15\x14\x13\x12'
>>> pack('@ic', 0x12131415, b'#')
b'\x15\x14\x13\x12#'
>>> calcsize('@ci')
8
>>> calcsize('@ic')
5
```

다음 포맷 `'llh0l'`는 플랫폼의 long이 4바이트 경계에 정렬된다고 가정할 때 끝에 2개의 패드 바이트를 추가합니다:

```
>>> pack('@llh0l', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

더 보기

모듈 *array*
: 동종 데이터의 패킹 된 바이너리 저장소.

모듈 *json*
: JSON 인코더와 디코더.

모듈 *pickle*
: 파이썬 객체 직렬화.

7.1.3 응용

Two main applications for the *struct* module exist, data interchange between Python and C code within an application or another application compiled using the same compiler (*native formats*), and data interchange between applications using agreed upon data layout (*standard formats*). Generally speaking, the format strings constructed for these two domains are distinct.

네이티브 포맷

When constructing format strings which mimic native layouts, the compiler and machine architecture determine byte ordering and padding. In such cases, the `@` format character should be used to specify native byte ordering and data sizes. Internal pad bytes are normally inserted automatically. It is possible that a zero-repeat format code will be

needed at the end of a format string to round up to the correct byte boundary for proper alignment of consecutive chunks of data.

Consider these two simple examples (on a 64-bit, little-endian machine):

```
>>> calcsize('@lhl')
24
>>> calcsize('@llh')
18
```

Data is not padded to an 8-byte boundary at the end of the second format string without the use of extra padding. A zero-repeat format code solves that problem:

```
>>> calcsize('@llh0l')
24
```

The `'x'` format code can be used to specify the repeat, but for native formats it is better to use a zero-repeat format like `'0l'`.

By default, native byte ordering and alignment is used, but it is better to be explicit and use the `'@'` prefix character.

표준 포맷

When exchanging data beyond your process such as networking or storage, be precise. Specify the exact byte order, size, and alignment. Do not assume they match the native order of a particular machine. For example, network byte order is big-endian, while many popular CPUs are little-endian. By defining this explicitly, the user need not care about the specifics of the platform their code is running on. The first character should typically be `<` or `>` (or `!`). Padding is the responsibility of the programmer. The zero-repeat format character won' t work. Instead, the user must explicitly add `'x'` pad bytes where needed. Revisiting the examples from the previous section, we have:

```
>>> calcsize('<qh6xq')
24
>>> pack('<qh6xq', 1, 2, 3) == pack('@lhl', 1, 2, 3)
True
>>> calcsize('@llh')
18
>>> pack('@llh', 1, 2, 3) == pack('<qqh', 1, 2, 3)
True
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
24
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
True
```

The above results (executed on a 64-bit machine) aren' t guaranteed to match when executed on different machines. For example, the examples below were executed on a 32-bit machine:

```
>>> calcsize('<qqh6x')
24
>>> calcsize('@llh0l')
12
>>> pack('@llh0l', 1, 2, 3) == pack('<qqh6x', 1, 2, 3)
False
```

7.1.4 클래스

struct 모듈은 또한 다음 형을 정의합니다:

class struct.**Struct**(*format*)

포맷 문자열 *format*에 따라 바이너리 데이터를 쓰고 읽는 새 Struct 객체를 반환합니다. Struct 객체를 한 번 만들고 메서드를 호출하는 것은 포맷 문자열이 한 번만 컴파일 되기 때문에 같은 포맷으로 모듈 수준 함수를 호출하는 것보다 효율적입니다.

> **참고**
>
> 모듈 수준 함수에 전달된 최신 포맷 문자열의 컴파일된 버전이 캐시 되므로, 몇 가지 포맷 문자열만 사용하는 프로그램은 단일 *Struct* 인스턴스 재사용에 대해 신경 쓸 필요가 없습니다.

컴파일된 Struct 객체는 다음 메서드와 어트리뷰트를 지원합니다:

pack(*v1*, *v2*, ...)

pack() 함수와 동일하고, 컴파일된 포맷을 사용합니다. (len(result) 는 *size*와 같게 됩니다.)

pack_into(*buffer*, *offset*, *v1*, *v2*, ...)

pack_into() 함수와 동일하고, 컴파일된 포맷을 사용합니다.

unpack(*buffer*)

unpack() 함수와 동일하고, 컴파일된 포맷을 사용합니다. 바이트 단위의 버퍼 크기는 *size*와 같아야 합니다.

unpack_from(*buffer*, *offset=0*)

unpack_from() 함수와 동일하고, 컴파일된 포맷을 사용합니다. *offset* 위치에서 시작하는 바이트 단위의 버퍼 크기는 *size* 이상이어야 합니다.

iter_unpack(*buffer*)

iter_unpack() 함수와 동일하고, 컴파일된 포맷을 사용합니다. 바이트 단위의 버퍼 크기는 *size*의 배수이어야 합니다.

Added in version 3.4.

format

이 Struct 객체를 구성하는 데 사용된 포맷 문자열.

버전 3.7에서 변경: 포맷 문자열형은 이제 *bytes* 대신 *str*입니다.

size

*format*에 해당하는 구조체(*pack()* 메서드에 의해 생성된 바이트열 객체)의 계산된 크기.

버전 3.13에서 변경: The *repr*() of structs has changed. It is now:

```
>>> Struct('i')
Struct('i')
```

7.2 codecs — 코덱 레지스트리와 베이스 클래스

소스 코드: Lib/codecs.py

이 모듈은 표준 파이썬 코덱(인코더와 디코더)의 베이스 클래스를 정의하고, 코덱과 에러 처리 조회 프로세스를 관리하는 내부 파이썬 코덱 레지스트리에 대한 액세스를 제공합니다. 대부분의 표준 코덱은 텍스트를 바이트열로 인코딩하는 (그리고 바이트열을 텍스트로 디코딩하는) 텍스트 인코딩이지만, 텍스트를 텍스트로 인코딩하고 바이트열을 바이트열로 인코딩하는 코덱도 제공됩니다. 사용자 정의 코덱은 임의 형 간에 인코딩과 디코딩 할 수 있지만, 일부 모듈 기능은 텍스트 인코딩이나 *bytes*로 인코딩하는 코덱과 함께 사용하도록 특별히 제한됩니다.

이 모듈은 임의의 코덱으로 인코딩과 디코딩하는 다음 함수를 정의합니다:

codecs.**encode**(*obj, encoding='utf-8', errors='strict'*)

*encoding*을 위해 등록된 코덱을 사용하여 *obj*를 인코딩합니다.

원하는 에러 처리 방식을 설정하기 위해 *errors*가 주어질 수 있습니다. 기본 에러 처리기는 `'strict'`이며 인코딩 에러가 `ValueError`(또는 `UnicodeEncodeError`와 같은 더 많은 코덱 관련 서브 클래스)를 발생시킨다는 뜻입니다. 코덱 에러 처리에 대한 자세한 내용은 코덱 베이스 클래스를 참조하십시오.

codecs.**decode**(*obj, encoding='utf-8', errors='strict'*)

*encoding*을 위해 등록된 코덱을 사용하여 *obj*를 디코딩합니다.

원하는 에러 처리 방식을 설정하기 위해 *errors*가 주어질 수 있습니다. 기본 에러 처리기는 `'strict'`이며 디코딩 에러가 `ValueError`(또는 `UnicodeDecodeError`와 같은 더 많은 코덱 관련 서브 클래스)를 발생시킨다는 뜻입니다. 코덱 에러 처리에 대한 자세한 내용은 코덱 베이스 클래스를 참조하십시오.

codecs.**charmap_build**(*string*)

Return a mapping suitable for encoding with a custom single-byte encoding. Given a `str` *string* of up to 256 characters representing a decoding table, returns either a compact internal mapping object `EncodingMap` or a `dictionary` mapping character ordinals to byte values. Raises a `TypeError` on invalid input.

각 코덱에 대한 자세한 내용도 직접 확인할 수 있습니다:

codecs.**lookup**(*encoding, /*)

파이썬 코덱 레지스트리에서 코덱 정보를 조회하고 아래 정의된 `CodecInfo` 객체를 반환합니다.

인코딩은 먼저 레지스트리 캐시에서 조회됩니다. 찾을 수 없으면, 등록된 검색 함수 리스트를 탐색합니다. `CodecInfo` 객체가 없으면, `LookupError`가 발생합니다. 그렇지 않으면, `CodecInfo` 객체가 캐시에 저장되고 호출자에게 반환됩니다.

class codecs.**CodecInfo**(*encode, decode, streamreader=None, streamwriter=None, incrementalencoder=None, incrementaldecoder=None, name=None*)

코덱 레지스트리를 조회할 때의 코덱 세부 정보. 생성자 인자는 같은 이름의 어트리뷰트에 저장됩니다:

name

인코딩의 이름.

encode

decode

상태 없는 인코딩과 디코딩 함수. 이들은 코덱 인스턴스의 `encode()`와 `decode()` 메서드와 같은 인터페이스를 갖는 함수나 메서드여야 합니다(코덱 인터페이스를 참조하십시오). 함수나 메서드는 상태 없는 모드로 작동할 것으로 기대됩니다.

incrementalencoder

incrementaldecoder

증분형 인코더와 디코더 클래스 또는 팩토리 함수. 이들은 각각 베이스 클래스 `IncrementalEncoder`와 `IncrementalDecoder`가 정의하는 인터페이스를 제공해야 합니다. 증분 코덱은 상태를 유지할 수 있습니다.

streamwriter

streamreader

스트림 기록기와 판독기 클래스 또는 팩토리 함수. 이들은 각각 베이스 클래스 `StreamWriter`와 `StreamReader`가 정의하는 인터페이스를 제공해야 합니다. 스트림 코덱은 상태를 유지할 수 있습니다.

다양한 코덱 구성 요소에 대한 액세스를 단순화하기 위해, 이 모듈은 코덱 조회에 `lookup()`을 사용하는 다음과 같은 추가 함수를 제공합니다:

codecs.**getencoder**(*encoding*)

주어진 인코딩에 대한 코덱을 찾아서 해당 인코더 함수를 반환합니다.

인코딩을 찾을 수 없는 경우 `LookupError`를 발생시킵니다.

codecs.**getdecoder**(*encoding*)

주어진 인코딩에 대한 코덱을 찾아서 해당 디코더 함수를 반환합니다.

인코딩을 찾을 수 없는 경우 `LookupError`를 발생시킵니다.

codecs.**getincrementalencoder**(*encoding*)

주어진 인코딩에 대한 코덱을 찾아서 증분 인코더 클래스나 팩토리 함수를 반환합니다.

인코딩을 찾을 수 없거나 코덱이 증분 인코더를 지원하지 않는 경우 `LookupError`를 발생시킵니다.

codecs.**getincrementaldecoder**(*encoding*)

주어진 인코딩에 대한 코덱을 찾아서 증분 디코더 클래스나 팩토리 함수를 반환합니다.

인코딩을 찾을 수 없거나 코덱이 증분 디코더를 지원하지 않는 경우 `LookupError`를 발생시킵니다.

codecs.**getreader**(*encoding*)

주어진 인코딩의 코덱을 찾아서 `StreamReader` 클래스나 팩토리 함수를 반환합니다.

인코딩을 찾을 수 없는 경우 `LookupError`를 발생시킵니다.

codecs.**getwriter**(*encoding*)

주어진 인코딩의 코덱을 찾아서 `StreamWriter` 클래스나 팩토리 함수를 반환합니다.

인코딩을 찾을 수 없는 경우 `LookupError`를 발생시킵니다.

적합한 코덱 검색 함수를 등록하여 사용자 정의 코덱을 사용할 수 있도록 합니다:

codecs.**register**(*search_function*, /)

코덱 검색 함수를 등록합니다. 검색 함수는 모두 소문자로 이루어지고 하이픈과 스페이스를 언더스코어로 변환한 인코딩 이름인 하나의 인자를 취하고, `CodecInfo` 객체를 반환해야 합니다. 검색 함수가 주어진 인코딩을 찾지 못하면, `None`을 반환해야 합니다.

버전 3.9에서 변경: Hyphens and spaces are converted to underscore.

codecs.**unregister**(*search_function*, /)

Unregister a codec search function and clear the registry' s cache. If the search function is not registered, do nothing.

Added in version 3.10.

내장 `open()`과 관련 `io` 모듈은 인코딩된 텍스트 파일 작업에 권장되는 접근 방법이지만, 이 모듈은 바이너리 파일로 작업할 때 더 넓은 범위의 코덱을 사용할 수 있도록 하는 추가 유틸리티 함수와 클래스를 제공합니다:

codecs.**open**(*filename*, *mode='r'*, *encoding=None*, *errors='strict'*, *buffering=-1*)

주어진 *mode*를 사용하여 인코딩된 파일을 열고 투명한 인코딩/디코딩을 제공하는 `StreamReaderWriter`의 인스턴스를 반환합니다. 기본 파일 모드는 `'r'`이고, 파일을 읽기 모드로 연다는 뜻입니다.

> **참고**
>
> *encoding*이 `None`이 아니면, 하부 인코딩된 파일은 항상 바이너리 모드로 열립니다. 읽기와 쓰기 시 `'\n'`의 자동 변환이 수행되지 않습니다. *mode* 인자는 내장 `open()` 함수에 허용되는 모든 바이너리 모드일 수 있습니다; `'b'`가 자동으로 추가됩니다.

*encoding*은 파일에 사용될 인코딩을 지정합니다. 바이트열에서 인코딩하고 바이트열로 디코딩하는 모든 인코딩이 허용되며, 파일 메서드가 지원하는 데이터형은 사용된 코덱에 따라 다릅니다.

에러 처리를 정의하기 위해 *errors*가 제공될 수 있습니다. 기본값은 `'strict'`이고 인코딩 에러가 발생하면 `ValueError`가 발생합니다.

*buffering*은 내장 `open()` 함수에서와 같은 의미입니다. 기본값은 -1이며 기본 버퍼 크기가 사용됨을 의미합니다.

버전 3.11에서 변경: The `'U'` mode has been removed.

버전 3.14부터 폐지됨: `codecs.open()` has been superseded by `open()`.

codecs.**EncodedFile**(*file*, *data_encoding*, *file_encoding=None*, *errors='strict'*)

투명한 트랜스코딩을 제공하는 *file*의 래핑 된 버전인 `StreamRecoder` 인스턴스를 반환합니다. 래핑 된 버전이 닫힐 때 원본 파일이 닫힙니다.

래핑 된 파일에 기록된 데이터는 주어진 *data_encoding*에 따라 디코딩된 다음 *file_encoding*을 사용하여 바이트열로 원본 파일에 기록됩니다. 원본 파일에서 읽은 바이트열은 *file_encoding*에 따라 디코딩되며, 결과는 *data_encoding*을 사용하여 인코딩됩니다.

*file_encoding*이 제공되지 않으면, 기본값은 *data_encoding*입니다.

에러 처리를 정의하기 위해 *errors*가 제공될 수 있습니다. 기본값은 `'strict'`이며, 인코딩 에러가 발생하면 `ValueError`가 발생합니다.

codecs.**iterencode**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)

Uses an incremental encoder to iteratively encode the input provided by *iterator*. *iterator* must yield `str` objects. This function is a *generator*. The *errors* argument (as well as any other keyword argument) is passed through to the incremental encoder.

이 함수를 사용하려면 코덱에서 인코딩할 텍스트 `str` 객체를 허용해야 합니다. 따라서 `base64_codec`과 같은 바이트열-바이트열 인코더는 지원하지 않습니다.

codecs.**iterdecode**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)

Uses an incremental decoder to iteratively decode the input provided by *iterator*. *iterator* must yield `bytes` objects. This function is a *generator*. The *errors* argument (as well as any other keyword argument) is passed through to the incremental decoder.

이 함수를 사용하려면 코덱에서 디코딩할 `bytes` 객체를 허용해야 합니다. 따라서 `rot_13`이 `iterencode()`로 동등하게 사용될 수 있지만, `rot_13`과 같은 텍스트-텍스트 인코더는 지원하지 않습니다.

codecs.**readbuffer_encode**(*buffer*, *errors=None*, */*)

Return a `tuple` containing the raw bytes of *buffer*, a buffer-compatible object or `str` (encoded to UTF-8 before processing), and their length in bytes.

The *errors* argument is ignored.

```
>>> codecs.readbuffer_encode(b"Zito")
(b'Zito', 4)
```

이 모듈은 또한 플랫폼 종속 파일을 읽고 쓰는 데 유용한 다음 상수를 제공합니다:

codecs.**BOM**

codecs.**BOM_BE**

codecs.**BOM_LE**

codecs.**BOM_UTF8**

codecs.**BOM_UTF16**

codecs.**BOM_UTF16_BE**

codecs.**BOM_UTF16_LE**

codecs.**BOM_UTF32**

codecs.**BOM_UTF32_BE**

codecs.**BOM_UTF32_LE**

이 상수는 여러 인코딩에서 유니코드 바이트 순서 표시(BOM)인 다양한 바이트 시퀀스를 정의합니다. UTF-16과 UTF-32 데이터 스트림에서 사용된 바이트 순서를 나타내는 데 사용되며, UTF-8에서 유니코드 서명으로 사용됩니다. `BOM_UTF16`은 플랫폼의 네이티브 바이트 순서에 따라 `BOM_UTF16_BE`나 `BOM_UTF16_LE`이며, `BOM`은 `BOM_UTF16`의 별칭, `BOM_LE`는 `BOM_UTF16_LE`의 별칭,

*BOM_BE*는 *BOM_UTF16_BE*의 별칭입니다. 다른 것은 UTF-8과 UTF-32 인코딩에서 BOM을 나타냅니다.

7.2.1 코덱 베이스 클래스

codecs 모듈은 코덱 객체로 작업하기 위한 인터페이스를 정의하는 베이스 클래스 집합을 정의하며, 사용자 정의 코덱 구현의 기초로 사용될 수도 있습니다.

각 코덱은 파이썬에서 코덱으로 사용할 수 있도록 네 가지 인터페이스를 정의해야 합니다: 상태 없는 인코더, 상태 없는 디코더, 스트림 판독기 및 스트림 기록기. 스트림 판독기와 기록기는 일반적으로 상태 없는 인코더/디코더를 재사용하여 파일 프로토콜을 구현합니다. 코덱 작성자는 코덱에서 인코딩과 디코딩 에러를 처리하는 방법도 정의해야 합니다.

에러 처리기

에러 처리를 단순화하고 표준화하기 위해, 코덱은 *errors* 문자열 인자를 받아들여 다른 에러 처리 체계를 구현할 수 있습니다:

```
>>> 'German ß, ☃'.encode(encoding='ascii', errors='backslashreplace')
b'German \\xdf, \\u266c'
>>> 'German ß, ☃'.encode(encoding='ascii', errors='xmlcharrefreplace')
b'German &#223;, &#9836;'
```

다음 에러 처리기는 모든 파이썬 표준 인코딩과 함께 사용할 수 있습니다:

값	의미
`'strict'`	*UnicodeError*(또는 서브 클래스)를 발생시킵니다, 이것이 기본값입니다. *strict_errors()*에서 구현되었습니다.
`'ignore'`	잘못된 데이터를 무시하고 추가 통지 없이 계속 진행합니다. *ignore_errors()*에서 구현되었습니다.
`'replace'`	교체 마커로 교체합니다. 인코딩 시, `?` (ASCII 문자) 를 사용합니다. 디코딩 시, `\ufffd` (U+FFFD, 공식 REPLACEMENT CHARACTER) 를 사용합니다. *replace_errors()*에서 구현되었습니다.
`'backslashreplace'`	Replace with backslashed escape sequences. On encoding, use hexadecimal form of Unicode code point with formats `\xhh` `\uxxxx` `\Uxxxxxxxx`. On decoding, use hexadecimal form of byte value with format `\xhh`. Implemented in *backslashreplace_errors()*.
`'surrogateescape'`	디코딩 시, 바이트를 `U+DC80`에서 `U+DCFF` 범위의 개별 서로게이트 코드 (surrogate code)로 바꿉니다. 이 코드는 데이터를 인코딩할 때 `'surrogateescape'` 에러 처리기가 사용되면 같은 바이트로 다시 변환됩니다. (자세한 내용은 **PEP 383**을 참조하십시오.)

다음 에러 처리기는 인코딩에만 적용됩니다 (텍스트 인코딩 내):

값	의미
`'xmlcharref`	`&#`*num*`;` 포맷의 유니코드 코드 포인트의 십진수 형식인 XML/HTML 숫자 문자 참조로 교체합니다. *xmlcharrefreplace_errors()*에서 구현되었습니다.
`'namereplac`	`\N{...}` 이스케이프 시퀀스로 교체합니다, 중괄호 안에 나타나는 것은 유니코드 문자 데이터베이스의 이름(Name) 속성입니다. *namereplace_errors()*에서 구현되었습니다.

또한, 다음 에러 처리기는 지정된 코덱에만 적용됩니다:

값	코덱	의미
`'surroc`	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	서로게이트 코드 포인트 (`U+D800` - `U+DFFF`) 를 일반 코드 포인트로 인코딩 및 디코딩하도록 허용합니다. 그렇지 않으면 이 코덱들은 *str*에 있는 서로게이트 코드 포인트의 존재를 에러로 취급합니다.

Added in version 3.1: `'surrogateescape'`와 `'surrogatepass'` 에러 처리기.

버전 3.4에서 변경: `'surrogatepass'` 에러 처리기는 이제 utf-16* 와 utf-32* 코덱에서 작동합니다.

Added in version 3.5: `'namereplace'` 에러 처리기.

버전 3.5에서 변경: `'backslashreplace'` 에러 처리기는 이제 디코딩과 변환(translating)에서 작동합니다.

새 이름으로 에러 처리기를 등록하여 허용되는 값 집합을 확장할 수 있습니다:

codecs.**register_error**(*name, error_handler, /*)

에러 처리 함수 *error_handler*를 *name*이라는 이름으로 등록합니다. *error_handler* 인자는 *name*이 errors 매개 변수로 지정될 때, 인코딩과 디코딩 중에 에러가 있으면 호출됩니다.

인코딩의 경우, 에러 위치에 대한 정보가 포함된 `UnicodeEncodeError` 인스턴스와 함께 *error_handler*가 호출됩니다. 에러 처리기는 이 예외나 다른 예외를 발생시키거나, 입력의 인코딩할 수 없는 부분의 대체 값과 인코딩을 계속할 위치를 담은 튜플을 반환해야 합니다. 대체 값은 `str`이나 `bytes`일 수 있습니다. 대체 값이 바이트열이면, 인코더는 이를 단순히 출력 버퍼에 복사합니다. 대체 값이 문자열이면, 인코더는 대체 값을 인코딩합니다. 지정된 위치에서 원래 입력으로 인코딩이 계속됩니다. 음수 위칫값은 입력 문자열의 끝을 기준으로 처리됩니다. 결과 위치가 범위를 벗어나면 `IndexError`가 발생합니다.

`UnicodeDecodeError`나 `UnicodeTranslateError`가 처리기로 전달되고 에러 처리기의 대체 값을 출력에 직접 넣는다는 점을 제외하면, 디코딩과 변환(translating)은 비슷하게 작동합니다.

이전에 등록된 에러 처리기(표준 에러 처리기를 포함하여)는 이름으로 조회할 수 있습니다:

codecs.**lookup_error**(*name, /*)

*name*이라는 이름으로 이전에 등록된 에러 처리기를 반환합니다.

처리기를 찾을 수 없으면 `LookupError`를 발생시킵니다.

다음과 같은 표준 에러 처리기가 모듈 수준 함수로 제공됩니다:

codecs.**strict_errors**(*exception*)

`'strict'` 에러 처리를 구현합니다.

각 인코딩이나 디코딩 에러는 `UnicodeError`를 발생시킵니다.

codecs.**ignore_errors**(*exception*)

`'ignore'` 에러 처리를 구현합니다.

잘못된 형식의 데이터는 무시됩니다; 추가 통지 없이 인코딩이나 디코딩이 계속됩니다.

codecs.**replace_errors**(*exception*)

`'replace'` 에러 처리를 구현합니다.

Substitutes `?` (ASCII character) for encoding errors or `\ufffd` (U+FFFD, the official REPLACEMENT CHARACTER) for decoding errors.

codecs.**backslashreplace_errors**(*exception*)

`'backslashreplace'` 에러 처리를 구현합니다.

Malformed data is replaced by a backslashed escape sequence. On encoding, use the hexadecimal form of Unicode code point with formats `\xhh` `\uxxxx` `\Uxxxxxxxx`. On decoding, use the hexadecimal form of byte value with format `\xhh`.

버전 3.5에서 변경: 디코딩과 변환(translating)에서 작동합니다.

codecs.**xmlcharrefreplace_errors**(*exception*)

`'xmlcharrefreplace'` 에러 처리를 구현합니다 (텍스트 인코딩으로 인코딩하는 경우에만 해당).

인코드할 수 없는 문자는 `&#num;` 포맷의 유니코드 코드 포인트의 십진수 형식인 적절한 XML/HTML 숫자 문자 참조로 대체됩니다.

codecs.**namereplace_errors**(*exception*)

'namereplace' 에러 처리를 구현합니다 (텍스트 인코딩으로 인코딩하는 경우에만 해당).

인코드할 수 없는 문자는 \N{...} 이스케이프 시퀀스로 대체됩니다. 중괄호 안에 나타나는 문자 집합은 유니코드 문자 데이터베이스의 이름(Name) 속성입니다. 예를 들어, 독일어 소문자 'ß'는 바이트 시퀀스 \N{LATIN SMALL LETTER SHARP S}로 변환됩니다.

Added in version 3.5.

상태 없는 인코딩과 디코딩

기본 *Codec* 클래스는 다음과 같은 메서드를 정의하는데, 상태 없는 인코더와 디코더의 함수 인터페이스를 정의하기도 합니다:

class codecs.**Codec**

encode(*input*, *errors*='*strict*')

객체 *input*을 인코딩하고 튜플(출력 객체, 소비한 길이)을 반환합니다. 예를 들어, 텍스트 인코딩은 특정 문자 집합 인코딩 (예를 들어, cp1252나 iso-8859-1)을 사용하여 문자열 객체를 바이트열 객체로 변환합니다.

errors 인자는 적용할 에러 처리를 정의합니다. 기본값은 'strict' 처리입니다.

이 메서드는 *Codec* 인스턴스에 상태를 저장하지 않을 수 있습니다. 인코딩 효율을 높이기 위해 상태를 유지해야 하는 코덱에서는 *StreamWriter*를 사용하십시오.

인코더는 길이가 0인 입력을 처리하고 이 상황에서는 출력 객체 형의 빈 객체를 반환할 수 있어야 합니다.

decode(*input*, *errors*='*strict*')

객체 *input*을 디코딩하고 튜플 (출력 객체, 소비한 길이)를 반환합니다. 예를 들어, 텍스트 인코딩의 경우, 디코딩은 특정 문자 집합 인코딩을 사용하여 인코딩된 바이트열 객체를 문자열 객체로 변환합니다.

텍스트 인코딩과 바이트열-바이트열 코덱의 경우, *input*은 바이트열 객체이거나 읽기 전용 버퍼 인터페이스를 제공하는 객체여야 합니다 – 예를 들어, 버퍼 객체와 및 메모리 맵핑 파일.

errors 인자는 적용할 에러 처리를 정의합니다. 기본값은 'strict' 처리입니다.

이 메서드는 *Codec* 인스턴스에 상태를 저장하지 않을 수 있습니다. 디코딩 효율을 높이기 위해 상태를 유지해야 하는 코덱에서는 *StreamReader*를 사용하십시오.

디코더는 길이가 0인 입력을 처리하고 이 상황에서 출력 객체 형의 빈 객체를 반환할 수 있어야 합니다.

증분 인코딩과 디코딩

*IncrementalEncoder*와 *IncrementalDecoder* 클래스는 증분 인코딩과 디코딩을 위한 기본 인터페이스를 제공합니다. 입력을 인코딩/디코딩하는 것이 상태 없는 인코더/디코더 함수를 한 번 호출하는 것이 아니라, 증분 인코더/디코더의 *encode()*/*decode()* 메서드를 여러 번 호출하여 수행됩니다. 증분 인코더/디코더는 메서드 호출 중에 인코딩/디코딩 프로세스를 추적합니다.

encode()/*decode()* 메서드에 대한 호출의 연결된 출력은 모든 단일 입력을 하나로 결합하여 상태 없는 인코더/디코더로 인코딩/디코딩되는 것과 같습니다.

IncrementalEncoder 객체

IncrementalEncoder 클래스는 여러 단계로 입력을 인코딩하는 데 사용됩니다. 파이썬 코덱 레지스트리와 호환되도록 모든 증분 인코더가 정의해야 하는 다음 메서드를 정의합니다.

class codecs.**IncrementalEncoder**(*errors*='*strict*')

IncrementalEncoder 인스턴스의 생성자.

모든 증분 인코더는 이 생성자 인터페이스를 제공해야 합니다. 추가 키워드 인자를 자유롭게 추가할 수 있지만, 여기에 정의된 키워드 인자만 파이썬 코덱 레지스트리에서 사용됩니다.

`IncrementalEncoder`는 *errors* 키워드 인자를 제공하여 다른 에러 처리 체계를 구현할 수 있습니다. 가능한 값은 에러 처리기를 참조하십시오.

errors 인자는 같은 이름의 어트리뷰트에 대입됩니다. 이 어트리뷰트에 대입하면 `IncrementalEncoder` 객체의 수명 동안 다른 에러 처리 전략 간에 전환할 수 있습니다.

encode(*object*, *final=False*)

*object*를 (인코더의 현재 상태를 고려하여) 인코딩하고 결과 인코딩된 객체를 반환합니다. 이것이 `encode()`에 대한 마지막 호출이면 *final*은 참이어야 합니다 (기본값은 거짓).

reset()

인코더를 초기 상태로 재설정합니다. 출력은 버려집니다: 인코더를 재설정하고 출력을 얻으려면, 필요하면 빈 바이트열이나 텍스트 문자열을 전달하여, `.encode(object, final=True)`를 호출하십시오.

getstate()

인코더의 현재 상태를 반환하는데, 정수여야 합니다. 구현은 `0`이 가장 흔한 상태가 되도록 해야 합니다. (정수보다 복잡한 상태는 상태를 마샬링/피클링하고 결과 문자열의 바이트열을 정수로 인코딩하여 정수로 변환할 수 있습니다.)

setstate(*state*)

인코더 상태를 *state*로 설정합니다. *state*는 `getstate()`가 반환한 인코더 상태여야 합니다.

IncrementalDecoder 객체

`IncrementalDecoder` 클래스는 여러 단계로 입력을 디코딩하는 데 사용됩니다. 파이썬 코덱 레지스트리와 호환되도록 모든 증분 디코더에서 정의해야 하는 다음 메서드를 정의합니다.

class codecs.**IncrementalDecoder**(*errors='strict'*)

`IncrementalDecoder` 인스턴스의 생성자.

모든 증분 디코더는 이 생성자 인터페이스를 제공해야 합니다. 추가 키워드 인자를 자유롭게 추가할 수 있지만, 여기에 정의된 키워드 인자만 파이썬 코덱 레지스트리에서 사용됩니다.

`IncrementalDecoder`는 *errors* 키워드 인자를 제공하여 다른 에러 처리 체계를 구현할 수 있습니다. 가능한 값은 에러 처리기를 참조하십시오.

errors 인자는 같은 이름의 어트리뷰트에 대입됩니다. 이 어트리뷰트에 대입하면 `IncrementalDecoder` 객체의 수명 동안 다른 에러 처리 전략 간에 전환할 수 있습니다.

decode(*object*, *final=False*)

*object*를 디코딩하고 (디코더의 현재 상태를 고려하여) 결과 디코딩된 객체를 반환합니다. 이것이 `decode()`에 대한 마지막 호출이면 *final*은 참이어야 합니다 (기본값은 거짓). *final*이 참이면 디코더는 입력을 완전히 디코딩해야 하며 모든 버퍼를 플러시 해야 합니다. 이것이 가능하지 않으면 (예를 들어 입력 끝의 불완전한 바이트 시퀀스로 인해), 상태 없는 경우와 같이 에러 처리를 시작해야 합니다 (예외가 발생시킬 수 있습니다).

reset()

디코더를 초기 상태로 재설정합니다.

getstate()

디코더의 현재 상태를 반환합니다. 두 항목이 있는 튜플이어야 하며, 첫 번째는 여전히 디코딩되지 않은 입력을 포함하는 버퍼여야 합니다. 두 번째는 정수여야 하며 추가 상태 정보일 수 있습니다. (구현은 `0`이 가장 흔한 추가 상태 정보가 되도록 해야 합니다.) 이 추가 상태 정보가 `0`이면, 입력 버퍼가 없고 추가 상태 정보가 `0`인 상태로 디코더를 설정할 수 있어서, 이전에 버퍼링된 입력을 디코더에 공급하면 출력을 생성하지 않고 이전 상태로 되돌아갈 수 있어야 합니다. (정수보다 복잡한 추가 상태 정보는 정보를 마샬링/피클링하고 결과 문자열의 바이트를 정수로 인코딩하여 정수로 변환할 수 있습니다.)

setstate(*state*)

디코더의 상태를 *state*로 설정합니다. *state*는 `getstate()`가 반환한 디코더 상태여야 합니다.

스트림 인코딩과 디코딩

`StreamWriter`와 `StreamReader` 클래스는 새로운 인코딩 서브 모듈을 매우 쉽게 구현하는 데 사용할 수 있는 범용 작업 인터페이스를 제공합니다. 이를 수행하는 방법에 대한 예는 `encodings.utf_8`을 참조하십시오.

StreamWriter 객체

`StreamWriter` 클래스는 `Codec`의 서브 클래스이며 파이썬 코덱 레지스트리와 호환되도록 모든 스트림 기록기가 정의해야 하는 다음 메서드를 정의합니다.

class codecs.**StreamWriter**(*stream*, *errors='strict'*)

`StreamWriter` 인스턴스의 생성자.

모든 스트림 기록기는 이 생성자 인터페이스를 제공해야 합니다. 추가 키워드 인자를 자유롭게 추가할 수 있지만, 여기에 정의된 키워드 인자만 파이썬 코덱 레지스트리에서 사용됩니다.

stream 인자는 특정 코덱에 적합하도록 텍스트나 바이너리 데이터를 쓰기 위해 열린 파일류 객체여야 합니다.

`StreamWriter`는 *errors* 키워드 인자를 제공하여 다른 에러 처리 체계를 구현할 수 있습니다. 하부 스트림 코덱이 지원할 수 있는 표준 에러 처리기에 대해서는 에러 처리기를 참조하십시오.

errors 인자는 같은 이름의 어트리뷰트에 대입됩니다. 이 어트리뷰트에 대입하면 `StreamWriter` 객체의 수명 동안 다른 에러 처리 전략 간에 전환할 수 있습니다.

write(*object*)

스트림에 인코딩된 객체의 내용을 씁니다.

writelines(*list*)

이어붙인 문자열의 이터러블을 스트림에 씁니다(`write()` 메서드를 재사용할 수 있습니다). 무한하거나 아주 큰 이터러블은 지원되지 않습니다. 표준 바이트열-바이트열 코덱은 이 메서드를 지원하지 않습니다.

reset()

내부 상태를 유지하는 데 사용되는 코덱 버퍼를 재설정합니다.

이 메서드를 호출하면 출력의 데이터가 깨끗한 상태가 되어 상태를 복구하기 위해 전체 스트림을 다시 스캔하지 않고도 새로운 최신 데이터를 추가할 수 있도록 합니다.

위의 메서드 외에도, `StreamWriter`는 하부 스트림에서 온 다른 모든 메서드와 어트리뷰트를 상속해야 합니다.

StreamReader 객체

`StreamReader` 클래스는 `Codec`의 서브 클래스이며 파이썬 코덱 레지스트리와 호환되도록 모든 스트림 판독기가 정의해야 하는 다음 메서드를 정의합니다.

class codecs.**StreamReader**(*stream*, *errors='strict'*)

`StreamReader` 인스턴스의 생성자.

모든 스트림 판독기는 이 생성자 인터페이스를 제공해야 합니다. 추가 키워드 인자를 자유롭게 추가할 수 있지만, 여기에 정의된 키워드 인자만 파이썬 코덱 레지스트리에서 사용됩니다.

stream 인자는 특정 코덱에 적합하게 텍스트나 바이너리 데이터를 읽기 위해 열린 파일류 객체여야 합니다.

`StreamReader`는 *errors* 키워드 인자를 제공하여 다른 에러 처리 체계를 구현할 수 있습니다. 하부 스트림 코덱이 지원할 수 있는 표준 에러 처리기에 대해서는 에러 처리기를 참조하십시오.

errors 인자는 같은 이름의 어트리뷰트에 대입됩니다. 이 어트리뷰트에 대입하면 `StreamReader` 객체의 수명 동안 다른 에러 처리 전략 간에 전환할 수 있습니다.

errors 인자에 허용되는 값 집합은 `register_error()`로 확장될 수 있습니다.

read(*size=-1*, *chars=-1*, *firstline=False*)

스트림에서 데이터를 디코딩하고 결과 객체를 반환합니다.

chars 인자는 반환할 디코딩 된 코드 포인트나 바이트의 수를 나타냅니다. `read()` 메서드는 요청된 것보다 더 많은 데이터를 반환하지 않지만, 사용 가능한 것이 충분하지 않으면 더 적게 반환할 수 있습니다.

size 인자는 디코딩을 위해 읽을 인코딩 된 바이트나 코드 포인트의 대략적인 최대 수를 나타냅니다. 디코더는 이 설정을 적절하게 수정할 수 있습니다. 기본값 -1은 가능한 한 많이 읽고 디코딩함을 나타냅니다. 이 매개 변수는 커다란 파일을 한 번에 디코딩하지 않도록 하기 위한 것입니다.

firstline 플래그는 이후 줄에 디코딩 에러가 있으면 첫 번째 줄만 반환해도 충분함을 나타냅니다.

이 메서드는 탐욕스러운(greedy) 읽기 전략을 사용해야 합니다. 즉, 인코딩 정의와 주어진 size 내에서 허용되는 만큼 많은 데이터를 읽어야 합니다. 예를 들어 스트림에 선택적 인코딩 종료나 상태 마커가 있으면, 이것도 읽어야 합니다.

readline(*size=None*, *keepends=True*)

입력 스트림에서 한 줄을 읽고 디코딩된 데이터를 반환합니다.

주어지면, *size*는 스트림의 `read()` 메서드에 size 인자로 전달됩니다.

*keepends*가 거짓이면 줄 종료가 반환된 줄에서 제거됩니다.

readlines(*sizehint=None*, *keepends=True*)

입력 스트림에서 사용 가능한 모든 줄을 읽고 줄의 리스트로 반환합니다.

줄 종료는 코덱의 `decode()` 메서드를 사용하여 구현되며 *keepends*가 참이면 리스트 항목에 포함됩니다.

주어지면, *sizehint*는 스트림의 `read()` 메서드에 *size* 인자로 전달됩니다.

reset()

내부 상태를 유지하는 데 사용되는 코덱 버퍼를 재설정합니다.

스트림 위치 변경이 발생하지 않아야 함에 유의하십시오. 이 메서드는 주로 디코딩 에러에서 복구할 수 있도록 하기 위한 것입니다.

위의 메서드 외에도 `StreamReader`는 하부 스트림에서 다른 모든 메서드와 어트리뷰트를 상속해야 합니다.

StreamReaderWriter 객체

`StreamReaderWriter`는 읽기와 쓰기 모드 모두에서 작동하는 스트림을 래핑하도록 하는 편의 클래스입니다.

`lookup()` 함수가 반환한 팩토리 함수를 사용하여 인스턴스를 구성할 수 있도록 설계되었습니다.

class codecs.**StreamReaderWriter**(*stream*, *Reader*, *Writer*, *errors='strict'*)

`StreamReaderWriter` 인스턴스를 만듭니다. *stream*은 파일류 객체여야 합니다. *Reader*와 *Writer*는 각각 `StreamReader`와 `StreamWriter` 인터페이스를 제공하는 팩토리 함수나 클래스여야 합니다. 에러 처리는 스트림 판독기와 기록기에 정의된 것과 같은 방식으로 수행됩니다.

`StreamReaderWriter` 인스턴스는 `StreamReader`와 `StreamWriter` 클래스가 결합한 인터페이스를 정의합니다. 하부 스트림에서 다른 모든 메서드와 어트리뷰트를 상속합니다.

StreamRecoder 객체

`StreamRecoder`는 한 인코딩에서 다른 인코딩으로 데이터를 변환하는데, 이는 때때로 다른 인코딩 환경을 다룰 때 유용합니다.

`lookup()` 함수가 반환한 팩토리 함수를 사용하여 인스턴스를 구성할 수 있도록 설계되었습니다.

class codecs.**StreamRecoder**(*stream*, *encode*, *decode*, *Reader*, *Writer*, *errors='strict'*)

양방향 변환을 구현하는 `StreamRecoder` 인스턴스를 만듭니다: *encode*와 *decode*는 프런트 엔드에 작동합니다 - `read()`와 `write()`를 호출하는 코드가 보는 데이터, 반면에 *Reader*와 *Writer*는 백 엔드에 작동합니다 - *stream*의 데이터.

이러한 객체를 사용하여 투명한 트랜스코딩을 수행 할 수 있습니다, 예를 들어, Latin-1 에서 UTF-8로 또는 그 반대로.

stream 인자는 파일류 객체여야 합니다.

*encode*와 *decode* 인자는 `Codec` 인터페이스를 준수해야 합니다. *Reader*와 *Writer*는 각각 `StreamReader`와 `StreamWriter` 인터페이스의 객체를 제공하는 팩토리 함수나 클래스여야 합니다.

에러 처리는 스트림 판독기와 기록기에 정의된 것과 같은 방식으로 수행됩니다.

`StreamRecoder` 인스턴스는 `StreamReader`와 `StreamWriter` 클래스가 결합한 인터페이스를 정의합니다. 하부 스트림에서 다른 모든 메서드와 어트리뷰트를 상속합니다.

7.2.2 인코딩과 유니코드

문자열은 `U+0000-U+10FFFF` 범위의 코드 포인트 시퀀스로 내부적으로 저장됩니다. (구현에 대한 자세한 내용은 **PEP 393**을 참조하십시오.) 일단 문자열 객체가 CPU와 메모리 외부에서 사용되면, 엔디안(endianness)과 이러한 배열이 바이트열로 저장되는 방식이 문제가 됩니다. 다른 코덱과 마찬가지로, 문자열을 바이트 시퀀스로 직렬화하는 것을 인코딩이라고 하며, 바이트 시퀀스에서 문자열을 다시 만드는 것을 디코딩이라고 합니다. 다양한 텍스트 직렬화 코덱이 있으며, 이를 집합적으로 텍스트 인코딩이라고 합니다.

가장 간단한 텍스트 인코딩(`'latin-1'` 또는 `'iso-8859-1'`이라고 합니다)은 코드 포인트 0–255를 바이트 `0x0-0xff`로 매핑합니다. 이것은 `U+00FF` 위의 코드 포인트를 포함하는 문자열 객체는 이 코덱으로 인코딩할 수 없음을 뜻합니다. 그렇게 하면 다음과 유사한 `UnicodeEncodeError` 가 발생합니다 (에러 메시지의 세부 사항은 다를 수 있습니다): `UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256).`

모든 유니코드 코드 포인트의 다른 부분 집합과 이러한 코드 포인트가 바이트 `0x0-0xff`에 매핑되는 방식을 선택하는 또 다른 인코딩 그룹(소위 charmap 인코딩)이 있습니다. 이 작업을 수행하는 방법을 보려면 간단히 예를 들어 `encodings/cp1252.py`(윈도우에서 주로 사용되는 인코딩)를 열어보십시오. 어떤 문자가 어떤 바이트 값에 매핑되는지를 나타내는 256개의 문자로 구성된 문자열 상수가 있습니다.

이러한 모든 인코딩은 유니코드로 정의된 1114112 코드 포인트 중 256개만 인코딩 할 수 있습니다. 각 유니코드 코드 포인트를 저장할 수 있는 간단하고 간단한 방법은 각 코드 포인트를 4개의 연속 바이트로 저장하는 것입니다. 두 가지 가능성이 있습니다: 바이트를 빅 엔디안이나 리틀 엔디안 순서로 저장합니다. 이 두 가지 인코딩을 각각 `UTF-32-BE`와 `UTF-32-LE`라고 합니다. 단점은, 예를 들어 리틀 엔디안 기계에서 `UTF-32-BE`를 사용하면 인코딩과 디코딩 시 항상 바이트를 스와프해야 한다는 것입니다. `UTF-32`는 이 문제를 피합니다: 바이트는 항상 자연 엔디안입니다. 이 바이트를 엔디안이 다른 CPU에서 읽을 때는, 바이트를 스와프해야 합니다. `UTF-16`이나 `UTF-32` 바이트 시퀀스의 엔디안을 감지할 수 있도록, BOM("Byte Order Mark – 바이트 순서 마크")이 있습니다. 이것은 유니코드 문자 `U+FEFF`입니다. 이 문자는 모든 `UTF-16`이나 `UTF-32` 바이트 시퀀스 앞에 붙일 수 있습니다. 이 문자의 바이트 스와프된 버전(`0xFFFE`)은 유니코드 텍스트에 나타날 수 없는 잘못된 문자입니다. 따라서 `UTF-16`이나 `UTF-32` 바이트 시퀀스의 첫 번째 문자가 `U+FFFE`이면, 디코딩 시 바이트를 스와프해야 합니다. 불행히도 `U+FEFF` 문자는 `ZERO WIDTH NO-BREAK SPACE`라는 두 번째 목적을 가지고 있었습니다: 너비가 없고 단어를 나눌 수 없도록 하는 문자입니다. 예를 들어 합자(ligature) 알고리즘에 힌트를 주기 위해 사용될 수 있습니다. 유니코드 4.0에서는 `U+FEFF`를 `ZERO WIDTH NO-BREAK SPACE`로 사용하는 것이 폐지되었습니다 (`U+2060`(`WORD JOINER`)이 이 역할을 맡습니다). 그런데도 유니코드 소프트웨어는 여전히 두 가지 역할 모두로 `U+FEFF`를 처리할 수 있어야 합니다: 인코딩된 바이트의 스토리지 배치를 결정하는 장치이며, 바이트 시퀀스가 문자열로 디코딩되면 사라지는 BOM의 역할, 다른 문자처럼 디코딩되는 일반 문자 `ZERO WIDTH NO-BREAK SPACE`의 역할.

유니코드 문자의 전체 범위를 인코딩 할 수 있는 또 다른 인코딩이 있습니다: UTF-8. UTF-8은 8비트 인코딩입니다. UTF-8에서는 바이트 순서에 관한 문제가 없음을 의미합니다. UTF-8 바이트 시퀀스의 각 바이트는 두 부분으로 구성됩니다: 마커 비트(최상위 비트)와 페이로드 비트. 마커 비트는 0에서 4개의 `1` 비트와 그 뒤에 `0`비트가 오는 시퀀스입니다. 유니코드 문자는 다음과 같이 인코딩됩니다 (x는 페이로드 비트이며, 이어 붙이면 유니코드 문자가 됩니다):

범위	인코딩
U-00000000 … U-0000007F	0xxxxxxx
U-00000080 … U-000007FF	110xxxxx 10xxxxxx
U-00000800 … U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 … U-0010FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

유니코드 문자의 최하위 비트는 가장 오른쪽에 있는 x 비트입니다.

UTF-8은 8비트 인코딩이라서 BOM이 필요하지 않으며 디코딩된 문자열의 모든 U+FEFF 문자(첫 번째 문자라 할지라도)는 ZERO WIDTH NO-BREAK SPACE로 처리됩니다.

외부 정보 없이 문자열 인코딩에 사용된 인코딩을 신뢰성 있게 결정하는 것은 불가능합니다. 각 charmap 인코딩은 모든 임의의 바이트 시퀀스를 디코딩 할 수 있습니다. 그러나 UTF-8에서는 그렇지 않습니다, UTF-8 바이트 시퀀스는 임의의 바이트 시퀀스를 허용하지 않는 구조를 갖기 때문입니다. UTF-8 인코딩 감지의 신뢰성을 높이기 위해, Microsoft는 메모장(Notepad) 프로그램을 위해 UTF-8의 변형을 발명했습니다 (파이썬에서 "utf-8-sig"라고 부릅니다): 유니코드 문자를 파일에 쓰기 전에, UTF-8 인코딩된 BOM(다음과 같은 바이트 시퀀스로 표시됩니다: 0xef, 0xbb, 0xbf)이 기록됩니다. 모든 charmap 인코딩된 파일이 이러한 바이트 값으로 시작한다는 것은 다소 불가능하기 때문에 (예를 들어 iso-8859-1 에서 다음과 같은 것으로 매핑됩니다

> LATIN SMALL LETTER I WITH DIAERESIS
> RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
> INVERTED QUESTION MARK

), 바이트 시퀀스에서 utf-8-sig 인코딩을 정확하게 추측할 수 있는 가능성을 높입니다. 따라서 여기서 BOM은 바이트 시퀀스를 생성하는 데 사용되는 바이트 순서를 결정할 수 있도록 하는데 사용되지는 않지만, 인코딩을 추측하는 데 도움이 되는 서명으로 사용됩니다. 인코딩할 때 utf-8-sig 코덱은 0xef, 0xbb, 0xbf를 파일의 처음 3바이트로 기록합니다. 디코딩할 때 utf-8-sig는 파일에서 처음 3바이트에 등장하면 이 3바이트를 건너뜁니다. UTF-8에서는, BOM 사용을 권장하지 않으며 일반적으로 피해야 합니다.

7.2.3 표준 인코딩

Python comes with a number of codecs built-in, either implemented as C functions or with dictionaries as mapping tables. The following table lists the codecs by name, together with a few common aliases, and the languages for which the encoding is likely used. Neither the list of aliases nor the list of languages is meant to be exhaustive. Notice that spelling alternatives that only differ in case or use a hyphen instead of an underscore are also valid aliases because they are equivalent when normalized by *normalize_encoding()*. For example, 'utf-8' is a valid alias for the 'utf_8' codec.

> **참고**
>
> The below table lists the most common aliases, for a complete list refer to the source aliases.py file.

On Windows, cpXXX codecs are available for all code pages. But only codecs listed in the following table are guarantead to exist on other platforms.

일부 공통 인코딩은 코덱 조회 메커니즘을 우회하여 성능을 향상할 수 있습니다. 이러한 최적화 기회는 CPython에서만 제한된 (대소 문자를 구분하는) 별칭 집합에 대해서 인식됩니다: utf-8, utf8, latin-1, latin1, iso-8859-1, iso8859-1, mbcs (윈도우 전용), ascii, us-ascii, utf-16, utf16, utf-32, utf32 및 대시 대신 밑줄을 사용한 것들. 이러한 인코딩에 대체 대안 별칭을 사용하면 실행 속도가 느려질 수 있습니다.

버전 3.6에서 변경: us-ascii에서 최적화 기회가 인식됩니다.

많은 문자 집합이 같은 언어를 지원합니다. 개별 문자(예를 들어 EURO SIGN 지원 여부)와 코드 위치에 문자를 대입하는 것에서 다릅니다. 특히 유럽 언어의 경우, 일반적으로 다음과 같은 변형이 있습니다:

- ISO 8859 코드 집합
- Microsoft 윈도우 코드 페이지, 일반적으로 8859 코드 집합에서 파생되지만, 제어 문자를 추가 그래픽 문자로 대체합니다

- IBM EBCDIC 코드 페이지
- IBM PC 코드 페이지, ASCII와 호환됩니다

코덱	별칭	언어
ascii	646, us-ascii	영어
big5	big5-tw, csbig5	중국어 번체
big5hkscs	big5-hkscs, hkscs	중국어 번체
cp037	IBM037, IBM039	영어
cp273	273, IBM273, csIBM273	독일어 Added in version 3.4.
cp424	EBCDIC-CP-HE, IBM424	히브리어
cp437	437, IBM437	영어
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	서유럽어
cp720		아랍어
cp737		그리스어
cp775	IBM775	발트어
cp850	850, IBM850	서유럽어
cp852	852, IBM852	중부와 동유럽어
cp855	855, IBM855	벨라루스어, 불가리아어, 마케도니아어, 러시아어, 세르비아어
cp856		히브리어
cp857	857, IBM857	터키어
cp858	858, IBM858	서유럽어
cp860	860, IBM860	포르투갈어
cp861	861, CP-IS, IBM861	아이슬란드어
cp862	862, IBM862	히브리어
cp863	863, IBM863	캐나다어
cp864	IBM864	아랍어
cp865	865, IBM865	덴마크어, 노르웨이어
cp866	866, IBM866	러시아어
cp869	869, CP-GR, IBM869	그리스어
cp874		태국어
cp875		그리스어
cp932	932, ms932, mskanji, ms-kanji, windows-31j	일본어
cp949	949, ms949, uhc	한국어
cp950	950, ms950	중국어 번체
cp1006		우르두어
cp1026	ibm1026	터키어
cp1125	1125, ibm1125, cp866u, ruscii	우크라이나어 Added in version 3.4.
cp1140	ibm1140	서유럽어
cp1250	windows-1250	중부와 동유럽어
cp1251	windows-1251	벨라루스어, 불가리아어, 마케도니아어, 러시아어, 세르비아어
cp1252	windows-1252	서유럽어
cp1253	windows-1253	그리스어
cp1254	windows-1254	터키어
cp1255	windows-1255	히브리어
cp1256	windows-1256	아랍어
cp1257	windows-1257	발트어
cp1258	windows-1258	베트남어
euc_jp	eucjp, ujis, u-jis	일본어

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

코덱	별칭	언어
euc_jis_2004	jisx0213, eucjis2004	일본어
euc_jisx0213	eucjisx0213	일본어
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	한국어
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	중국어 간체
gbk	936, cp936, ms936	통합 중국어
gb18030	gb18030-2000	통합 중국어
hz	hzgb, hz-gb, hz-gb-2312	중국어 간체
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	일본어
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	일본어
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	일본어, 한국어, 중국어 간체, 서유럽어, 그리스어
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	일본어
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	일본어
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	일본어
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	한국어
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	서유럽어
iso8859_2	iso-8859-2, latin2, L2	중부와 동유럽어
iso8859_3	iso-8859-3, latin3, L3	에스페란토어, 몰타어
iso8859_4	iso-8859-4, latin4, L4	발트어
iso8859_5	iso-8859-5, cyrillic	벨라루스어, 불가리아어, 마케도니아어, 러시아어, 세르비아어
iso8859_6	iso-8859-6, arabic	아랍어
iso8859_7	iso-8859-7, greek, greek8	그리스어
iso8859_8	iso-8859-8, hebrew	히브리어
iso8859_9	iso-8859-9, latin5, L5	터키어
iso8859_10	iso-8859-10, latin6, L6	북유럽어
iso8859_11	iso-8859-11, thai	태국어
iso8859_13	iso-8859-13, latin7, L7	발트어
iso8859_14	iso-8859-14, latin8, L8	켈틱어
iso8859_15	iso-8859-15, latin9, L9	서유럽어
iso8859_16	iso-8859-16, latin10, L10	남유럽어
johab	cp1361, ms1361	한국어
koi8_r		러시아어
koi8_t		타지크어 Added in version 3.5.
koi8_u		우크라이나어
kz1048	kz_1048, strk1048_2002, rk1048	카자흐어 Added in version 3.5.
mac_cyrillic	maccyrillic	벨라루스어, 불가리아어, 마케도니아어, 러시아어, 세르비아어
mac_greek	macgreek	그리스어
mac_iceland	maciceland	아이슬란드어
mac_latin2	maclatin2, maccentraleurope, mac_centeuro	중부와 동유럽어
mac_roman	macroman, macintosh	서유럽어

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

코덱	별칭	언어
mac_turkish	macturkish	터키어
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	카자흐어
shift_jis	csshiftjis, shiftjis, sjis, s_jis	일본어
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	일본어
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	일본어
utf_32	U32, utf32	모든 언어
utf_32_be	UTF-32BE	모든 언어
utf_32_le	UTF-32LE	모든 언어
utf_16	U16, utf16	모든 언어
utf_16_be	UTF-16BE	모든 언어
utf_16_le	UTF-16LE	모든 언어
utf_7	U7, unicode-1-1-utf-7	모든 언어
utf_8	U8, UTF, utf8, cp65001	모든 언어
utf_8_sig		모든 언어

버전 3.4에서 변경: utf-16* 과 utf-32* 인코더는 더는 서로게이트 코드 포인트(`U+D800-U+DFFF`)를 인코딩할 수 없습니다. utf-32* 디코더는 더는 서로게이트 코드 포인트에 해당하는 바이트 시퀀스를 디코딩하지 않습니다.

버전 3.8에서 변경: `cp65001`은 이제 `utf_8`의 별칭입니다.

버전 3.14에서 변경: On Windows, `cpXXX` codecs are now available for all code pages.

7.2.4 파이썬 특정 인코딩

사전 정의된 많은 코덱이 파이썬에만 해당하여, 코덱 이름은 파이썬 외부에서 의미가 없습니다. 예상되는 입력과 출력형에 따라 아래 표에 나열되어 있습니다 (텍스트 인코딩은 코덱의 가장 일반적인 사용 사례이지만, 하부 코덱 인프라는 단지 텍스트 인코딩이 아닌 임의의 데이터 변환을 지원합니다). 비대칭 코덱의 경우, 언급된 의미는 인코딩 방향을 설명합니다.

텍스트 인코딩

다음 코덱은 유니코드 텍스트 인코딩과 유사하게, *str*에서 *bytes*로의 인코딩과 바이트열류 객체에서 *str*로의 디코딩을 제공합니다.

코덱	별칭	의미
idna		RFC 3490을 구현합니다. *encodings.idna*도 참조하십시오. `errors='strict'`만 지원됩니다.
mbcs	ansi, dbcs	윈도우 전용: ANSI 코드 페이지 (CP_ACP)에 따라 피연산자를 인코딩합니다.
oem		윈도우 전용: OEM 코드 페이지 (CP_OEMCP)에 따라 피연산자를 인코딩합니다. Added in version 3.6.
palmos		PalmOS 3.5의 인코딩.
punycode		RFC 3492를 구현합니다. 상태 있는 코덱은 지원되지 않습니다.
raw_unicode_escape		다른 코드 포인트를 위해 `\uXXXX`와 `\UXXXXXXXX`를 사용하는 Latin-1 인코딩. 기존 역 슬래시는 어떤 방식으로도 이스케이프 되지 않습니다. 파이썬 피클 프로토콜에서 사용됩니다.
undefined		This Codec should only be used for testing purposes. 모든 변환에 대해 예외를 발생시킵니다, 빈 문자열조차. 에러 처리기는 무시됩니다.
unicode_escape		따옴표가 이스케이프 되지 않는 것을 제외하고, ASCII로 인코딩된 파이썬 소스 코드에서 유니코드 리터럴 내용으로 적합한 인코딩. Latin-1 소스 코드에서 디코딩합니다. 파이썬 소스 코드는 실제로는 기본적으로 UTF-8을 사용합니다.

버전 3.8에서 변경: "unicode_internal" 코덱이 제거되었습니다.

바이너리 변환

다음 코덱은 바이너리 변환을 제공합니다: 바이트열류 객체에서 *bytes*로의 매핑. (*str* 출력만 생성하는) *bytes.decode()*에서는 지원되지 않습니다.

코덱	별칭	의미	인코더 / 디코더
base64_codec[1]	base64, base_64	피연산자를 여러 줄 MIME base64로 변환합니다 (결과에는 항상 후행 `'\n'`이 포함됩니다). 버전 3.4에서 변경: 인코딩과 디코딩을 위해 모든 바이트열류 객체를 입력으로 받아들입니다.	`base64.encodebytes()` / `base64.decodebytes()`
bz2_codec	bz2	bz2를 사용하여 피연산자를 압축합니다.	`bz2.compress()` / `bz2.decompress()`
hex_codec	hex	바이트 당 두 자리 숫자를 사용하여, 피연산자를 16진 표현으로 변환합니다.	`binascii.b2a_hex()` / `binascii.a2b_hex()`
quopri_codec	quopri, quoted-printable, quoted_print	피연산자를 MIME quoted printable로 변환합니다.	`quotetabs=True`를 사용한 `quopri.encode()` / `quopri.decode()`
uu_codec	uu	uuencode를 사용하여 피연산자를 변환합니다.	
zlib_codec	zip, zlib	gzip을 사용하여 피연산자를 압축합니다.	`zlib.compress()` / `zlib.decompress()`

Added in version 3.2: 바이너리 변환의 복원.

버전 3.4에서 변경: 바이너리 변환에 대한 별칭의 복원.

텍스트 변환

다음 코덱은 텍스트 변환을 제공합니다: `str`에서 `str`로의 매핑. (`bytes` 출력만 생성하는) `str.encode()`에서는 지원되지 않습니다.

코덱	별칭	의미
rot_13	rot13	피연산자의 시저 암호(Caesar-cypher) 암호화를 반환합니다.

Added in version 3.2: `rot_13` 텍스트 변환 복원.

버전 3.4에서 변경: `rot13` 별칭 복원.

7.2.5 `encodings` — Encodings package

This module implements the following functions:

`encodings.`**`normalize_encoding`**(*encoding*)

Normalize encoding name *encoding*.

Normalization works as follows: all non-alphanumeric characters except the dot used for Python package names are collapsed and replaced with a single underscore, leading and trailing underscores are removed. For example, `' -;#'` becomes `'_'`.

Note that *encoding* should be ASCII only.

[1] `'base64_codec'`는 바이트열류 객체 외에도 디코딩을 위해 ASCII만 있는 `str` 인스턴스도 허용합니다.

> **참고**
>
> The following functions should not be used directly, except for testing purposes; `codecs.lookup()` should be used instead.

`encodings.search_function(encoding)`

Search for the codec module corresponding to the given encoding name *encoding*.

This function first normalizes the *encoding* using `normalize_encoding()`, then looks for a corresponding alias. It attempts to import a codec module from the encodings package using either the alias or the normalized name. If the module is found and defines a valid `getregentry()` function that returns a `codecs.CodecInfo` object, the codec is cached and returned.

If the codec module defines a `getaliases()` function any returned aliases are registered for future use.

`encodings.win32_code_page_search_function(encoding)`

Search for a Windows code page encoding *encoding* of the form `cpXXXX`.

If the code page is valid and supported, return a `codecs.CodecInfo` object for it.

가용성: Windows.

Added in version 3.14.

This module implements the following exception:

exception `encodings.CodecRegistryError`

Raised when a codec is invalid or incompatible.

7.2.6 `encodings.idna` — 응용 프로그램에서의 국제화된 도메인 이름

이 모듈은 **RFC 3490**(Internationalized Domain Names in Applications)과 **RFC 3492**(Nameprep: A Stringprep Profile for Internationalized Domain Names (IDN))를 구현합니다. `punycode` 인코딩과 `stringprep`을 기반으로 합니다.

If you need the IDNA 2008 standard from **RFC 5891** and **RFC 5895**, use the third-party idna module.

이 RFC는 함께 도메인 이름에서 비 ASCII 문자를 지원하는 프로토콜을 정의합니다. 비 ASCII 문자(가령 `www.Alliancefrançaise.nu`)를 포함하는 도메인 이름은 ASCII 호환 인코딩(ACE, 가령 `www.xn--alliancefranaise-npb.nu`)으로 변환됩니다. 그런 다음 도메인 이름의 ACE 형식은 DNS 조회, HTTP `Host` 필드 등과 같이 프로토콜에 의해 임의의 문자가 허용되지 않는 모든 위치에서 사용됩니다. 이 변환은 응용 프로그램에서 수행됩니다; 가능하다면 사용자에게 보이지 않습니다: 응용 프로그램은 전송 시에 유니코드 도메인 레이블을 투명하게 IDNA로 변환하고, 사용자에게 표시하기 전에 ACE 레이블을 다시 유니코드로 변환해야 합니다.

파이썬은 여러 가지 방식으로 이 변환을 지원합니다: `idna` 코덱은 유니코드와 ACE 간의 변환을 수행하여, **RFC 3490의 섹션 3.1**에 정의된 구분 문자를 기반으로 입력 문자열을 레이블로 분리하고 필요에 따라 각 레이블을 ACE로 변환하고, 반대로 입력 바이트 문자열을 `.` 구분 기호를 기반으로 레이블로 분리하고 모든 ACE 레이블을 유니코드로 변환합니다. 또한, `socket` 모듈은 유니코드 호스트 이름을 투명하게 ACE로 변환하므로, 응용 프로그램이 호스트 이름을 소켓 모듈로 전달할 때 호스트 이름 자체를 변환할 필요가 없습니다. 이에 더해, `http.client`와 `ftplib`와 같은, 함수 매개 변수로 호스트 이름이 있는 모듈은 유니코드 호스트 이름을 받아들입니다 (`http.client`는 해당 필드를 전송한다면 `Host` 필드에 IDNA 호스트 이름을 투명하게 전송합니다).

회선에서 호스트 이름을 수신할 때 (가령 역 이름 조회(reverse name lookup)에서), 유니코드로 자동 변환되지 않습니다: 이러한 호스트 이름을 사용자에게 제시하려는 응용 프로그램은 유니코드로 디코딩해야 합니다.

또한 모듈 `encodings.idna`는 nameprep 절차를 구현합니다. 이는 국제 도메인 이름의 대소 문자를 구분하지 않고 유사한 문자를 통합하기 위해 호스트 이름에 대해 특정 정규화를 수행합니다. 원한다면 nameprep 함수를 직접 사용할 수 있습니다.

encodings.idna.**nameprep**(*label*)

*label*의 nameprep 된 버전을 반환합니다. 구현은 현재 쿼리 문자열을 가정하므로, `AllowUnassigned`는 참입니다.

encodings.idna.**ToASCII**(*label*)

RFC 3490에 지정된 대로 레이블을 ASCII로 변환합니다. `UseSTD3ASCIIRules`는 거짓으로 가정합니다.

encodings.idna.**ToUnicode**(*label*)

RFC 3490에 지정된 대로 레이블을 유니코드로 변환합니다.

7.2.7 encodings.mbcs — 윈도우 ANSI 코드 페이지

이 모듈은 ANSI 코드 페이지(CP_ACP)를 구현합니다.

가용성: Windows.

버전 3.2에서 변경: 3.2 이전에는, *errors* 인자가 무시되었습니다; 인코딩에는 항상 `'replace'`가 사용되고, 디코딩에는 항상 `'ignore'`가 사용되었습니다.

버전 3.3에서 변경: 모든 에러 처리기를 지원합니다.

7.2.8 encodings.utf_8_sig — BOM 서명이 있는 UTF-8 코덱

이 모듈은 UTF-8 코덱의 변형을 구현합니다. 인코딩 시, UTF-8로 인코딩된 BOM을 UTF-8로 인코딩된 바이트열 앞에 붙입니다. 상태 있는 인코더의 경우 이 작업은 한 번만 수행됩니다 (바이트 스트림에 대한 첫 번째 쓰기 시). 디코딩 시, 데이터 시작에 있는 선택적 UTF-8 인코딩된 BOM을 건너뜁니다.

CHAPTER 8

데이터형

이 장에서 설명하는 모듈은 날짜와 시간, 고정형 배열, 힙 큐, 데크, 열거형과 같은 다양한 특수 데이터형을 제공합니다.

파이썬은 또한 일부 내장 데이터형, 특히 `dict`, `list`, `set`과 `frozenset`, `tuple`을 제공합니다. `str` 클래스는 유니코드 문자열을 저장하는 데 사용되고, `bytes`와 `bytearray` 클래스는 바이너리 데이터를 저장하는 데 사용됩니다.

이 장에서는 다음 모듈에 관해 설명합니다:

8.1 `datetime` — Basic date and time types

소스 코드: Lib/datetime.py

The `datetime` module supplies classes for manipulating dates and times.

날짜와 시간 산술이 지원되지만, 구현의 초점은 출력 포매팅과 조작을 위한 효율적인 어트리뷰트 추출입니다.

팁

Skip to *the format codes*.

더 보기

모듈 `calendar`
일반 달력 관련 함수들.

모듈 `time`
시간 액세스와 변환.

Module `zoneinfo`
Concrete time zones representing the IANA time zone database.

패키지 dateutil
시간대와 구문 분석 지원이 확장된 제삼자 라이브러리.

Package DateType
Third-party library that introduces distinct static types to e.g. allow *static type checkers* to differentiate between naive and aware datetimes.

8.1.1 어웨어와 나이브 객체

Date and time objects may be categorized as "aware" or "naive" depending on whether or not they include time zone information.

시간대와 일광 절약 시간 정보와 같은 적용 가능한 알고리즘과 정치적 시간 조정에 대한 충분한 지식을 통해, **어웨어** 객체는 다른 어웨어 객체와의 상대적인 위치를 파악할 수 있습니다. 어웨어 객체는 자의적으로 해석할 여지 없는 특정 시간을 나타냅니다.[1]

A **naive** object does not contain enough information to unambiguously locate itself relative to other date/time objects. Whether a naive object represents Coordinated Universal Time (UTC), local time, or time in some other time zone is purely up to the program, just like it is up to the program whether a particular number represents metres, miles, or mass. Naive objects are easy to understand and to work with, at the cost of ignoring some aspects of reality.

어웨어 객체가 필요한 응용 프로그램을 위해, `datetime` 과 `time` 객체에는 추상 `tzinfo` 클래스의 서브클래스 인스턴스로 설정할 수 있는 선택적 시간대 정보 어트리뷰트인 `tzinfo`가 있습니다. 이러한 `tzinfo` 객체는 UTC 시간으로부터의 오프셋, 시간대 이름 및 일광 절약 시간이 적용되는지에 대한 정보를 보관합니다.

Only one concrete `tzinfo` class, the `timezone` class, is supplied by the `datetime` module. The `timezone` class can represent simple time zones with fixed offsets from UTC, such as UTC itself or North American EST and EDT time zones. Supporting time zones at deeper levels of detail is up to the application. The rules for time adjustment across the world are more political than rational, change frequently, and there is no standard suitable for every application aside from UTC.

8.1.2 상수

The `datetime` module exports the following constants:

`datetime.`**`MINYEAR`**

The smallest year number allowed in a `date` or `datetime` object. `MINYEAR` is 1.

`datetime.`**`MAXYEAR`**

The largest year number allowed in a `date` or `datetime` object. `MAXYEAR` is 9999.

`datetime.`**`UTC`**

Alias for the UTC time zone singleton `datetime.timezone.utc`.

Added in version 3.11.

8.1.3 사용 가능한 형

class `datetime.`**`date`**

현재의 그레고리력이 언제나 적용되어왔고, 앞으로도 그럴 것이라는 가정하에 이상적인 나이브 날짜. 어트리뷰트: `year`, `month` 및 `day`.

class `datetime.`**`time`**

특정 날짜와 관계없이, 하루가 정확히 24*60*60초를 갖는다는 가정하에 이상적인 시간. (여기에는 "윤초"라는 개념이 없습니다.) 어트리뷰트: `hour`, `minute`, `second`, `microsecond` 및 `tzinfo`.

class `datetime.`**`datetime`**

날짜와 시간의 조합. 어트리뷰트: `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond` 및 `tzinfo`.

class `datetime.`**`timedelta`**

A duration expressing the difference between two `datetime` or `date` instances to microsecond resolution.

[1] 즉, 상대론적 효과를 무시한다면

class datetime.**tzinfo**

시간대 정보 객체의 추상 베이스 클래스. 이것들은 *datetime*과 *time* 클래스에서 사용자 정의할 수 있는 시간 조정 개념(예를 들어, 시간대와/나 일광 절약 시간을 다루는 것)을 제공하기 위해 사용됩니다.

class datetime.**timezone**

tzinfo 추상 베이스 클래스를 구현하는 클래스로, UTC로부터의 고정 오프셋을 나타냅니다.

Added in version 3.2.

이러한 형의 객체는 불변입니다.

서브 클래스 관계:

```
object
    timedelta
    tzinfo
        timezone
    time
    date
        datetime
```

공통 속성

date, *datetime*, *time* 및 *timezone* 형은 다음과 같은 공통 기능을 공유합니다:

- 이러한 형의 객체는 불변입니다.
- Objects of these types are *hashable*, meaning that they can be used as dictionary keys.
- 이러한 형의 객체는 *pickle* 모듈을 통한 효율적인 피클링을 지원합니다.

객체가 어웨어한지 나이브한지 판단하기

date 형의 객체는 항상 나이브합니다.

*time*이나 *datetime* 형의 객체는 어웨어할 수도 나이브할 수도 있습니다.

A *datetime* object `d` is aware if both of the following hold:

1. `d.tzinfo`가 `None`이 아닙니다
2. `d.tzinfo.utcoffset(d)`가 `None`을 반환하지 않습니다

Otherwise, `d` is naive.

A *time* object `t` is aware if both of the following hold:

1. `t.tzinfo`가 `None`이 아닙니다
2. `t.tzinfo.utcoffset(None)`이 `None`을 반환하지 않습니다

Otherwise, `t` is naive.

어웨어와 나이브 간의 차이점은 *timedelta* 객체에는 적용되지 않습니다.

8.1.4 timedelta 객체

A *timedelta* object represents a duration, the difference between two *datetime* or *date* instances.

class datetime.**timedelta**(*days=0*, *seconds=0*, *microseconds=0*, *milliseconds=0*, *minutes=0*, *hours=0*, *weeks=0*)

All arguments are optional and default to 0. Arguments may be integers or floats, and may be positive or negative.

days, *seconds* 및 *microseconds*만 내부적으로 저장됩니다. 인자는 이 단위로 변환됩니다:

- 밀리 초는 1000마이크로초로 변환됩니다.

- 분은 60초로 변환됩니다.
- 시간은 3600초로 변환됩니다.
- 주는 7일로 변환됩니다.

그런 다음 days, seconds 및 microseconds를 다음처럼 정규화하여 표현이 고유하도록 만듭니다

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (하루 내의 초 수)
- `-999999999 <= days <= 999999999`

다음 예는 *days, seconds* 및 *microseconds* 이외의 인자가 어떻게 "병합" 되어 세 개의 결과 어트리뷰트로 정규화되는지를 보여줍니다:

```
>>> from datetime import timedelta
>>> delta = timedelta(
...     days=50,
...     seconds=27,
...     microseconds=10,
...     milliseconds=29000,
...     minutes=5,
...     hours=8,
...     weeks=2
... )
>>> # Only days, seconds, and microseconds remain
>>> delta
datetime.timedelta(days=64, seconds=29156, microseconds=10)
```

인자가 float이고 부분 마이크로초가 있으면, 모든 인자의 남은 부분 마이크로초가 합쳐지고, 그 합은 동률일 때 짝수로 반올림하는 방식으로 가장 가까운 마이크로초로 반올림됩니다. float 인자가 없으면, 변환과 정규화 프로세스는 정확합니다 (정보가 손실되지 않습니다).

정규화된 days 값이 표시된 범위를 벗어나면, *OverflowError*가 발생합니다.

음수 값의 정규화는 처음 보면 놀라울 수 있습니다. 예를 들어:

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

Since the string representation of `timedelta` objects can be confusing, use the following recipe to produce a more readable format:

```
>>> def pretty_timedelta(td):
...     if td.days >= 0:
...         return str(td)
...     return f'-({-td!s})'
...
>>> d = timedelta(hours=-1)
>>> str(d)  # not human-friendly
'-1 day, 23:00:00'
>>> pretty_timedelta(d)
'-(1:00:00)'
```

클래스 어트리뷰트:

`timedelta.min`

가장 음수인 *timedelta* 객체, `timedelta(-999999999)`.

`timedelta.max`

가장 양수인 *timedelta* 객체, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

`timedelta.resolution`

같지 않은 *timedelta* 객체 간의 가능한 가장 작은 차이, `timedelta(microseconds=1)`.

Note that, because of normalization, `timedelta.max` is greater than `-timedelta.min`. `-timedelta.max` is not representable as a *timedelta* object.

인스턴스 어트리뷰트 (읽기 전용):

`timedelta.days`

Between -999,999,999 and 999,999,999 inclusive.

`timedelta.seconds`

Between 0 and 86,399 inclusive.

> **조심**
>
> It is a somewhat common bug for code to unintentionally use this attribute when it is actually intended to get a *total_seconds()* value instead:
>
> ```
> >>> from datetime import timedelta
> >>> duration = timedelta(seconds=11235813)
> >>> duration.days, duration.seconds
> (130, 3813)
> >>> duration.total_seconds()
> 11235813.0
> ```

`timedelta.microseconds`

Between 0 and 999,999 inclusive.

지원되는 연산:

연산	결과
`t1 = t2 + t3`	Sum of `t2` and `t3`. Afterwards `t1 - t2 == t3` and `t1 - t3 == t2` are true. (1)
`t1 = t2 - t3`	Difference of `t2` and `t3`. Afterwards `t1 == t2 - t3` and `t2 == t1 + t3` are true. (1)(6)
`t1 = t2 * i` 또는 `t1 = i * t2`	Delta multiplied by an integer. Afterwards `t1 // i == t2` is true, provided `i != 0`.
	In general, `t1 * i == t1 * (i-1) + t1` is true. (1)
`t1 = t2 * f` 또는 `t1 = f * t2`	델타에 float를 곱합니다. 결과는 동률일 때 짝수로 반올림하는 방식으로 timedelta.resolution의 가장 가까운 배수로 자리 올림 됩니다.
`f = t2 / t3`	Division (3) of overall duration `t2` by interval unit `t3`. Returns a *float* object.
`t1 = t2 / f` 또는 `t1 = t2 / i`	델타를 float나 int로 나눈 값. 결과는 동률일 때 짝수로 반올림하는 방식으로 timedelta.resolution의 가장 가까운 배수로 자리 올림 됩니다.
`t1 = t2 // i` 또는 `t1 = t2 // t3`	floor가 계산되고 나머지(있다면)를 버립니다. 두 번째 경우에는, 정수가 반환됩니다. (3)
`t1 = t2 % t3`	나머지가 *timedelta* 객체로 계산됩니다. (3)
`q, r = divmod(t1, t2)`	Computes the quotient and the remainder: `q = t1 // t2` (3) and `r = t1 % t2`. `q` is an integer and `r` is a *timedelta* object.
`+t1`	같은 값을 갖는 *timedelta* 객체를 반환합니다. (2)
`-t1`	Equivalent to `timedelta(-t1.days, -t1.seconds, -t1.microseconds)`, and to `t1 * -1`. (1)(4)
`abs(t)`	Equivalent to `+t` when `t.days >= 0`, and to `-t` when `t.days < 0`. (2)
`str(t)`	`[D day[s], ][H]H:MM:SS[.UUUUUU]` 형식의 문자열을 반환합니다. 여기서 D는 음의 `t`일 때 음수입니다. (5)
`repr(t)`	규범적 어트리뷰트 값을 가진 생성자 호출로 표현한 *timedelta* 객체의 문자열 표현을 반환합니다.

노트:

(1) 이것은 정확하지만, 오버플로 할 수 있습니다.

(2) 이것은 정확하고, 오버플로 할 수 없습니다.

(3) Division by zero raises *ZeroDivisionError*.

(4) `-timedelta.max` is not representable as a *timedelta* object.

(5) *timedelta* 객체의 문자열 표현은 내부 표현과 유사하게 정규화됩니다. 이것은 음의 timedelta가 다소 이상하게 표현되는 결과로 이어집니다. 예를 들어:

```
>>> timedelta(hours=-5)
datetime.timedelta(days=-1, seconds=68400)
>>> print(_)
-1 day, 19:00:00
```

(6) `t2 - t3` 표현식은 항상 `t2 + (-t3)` 표현식과 같아지는데, t3이 `timedelta.max`일 때만 예외입니다; 이때는 앞에 있는 것은 결과를 만들지만, 뒤에 있는 것은 오버플로를 일으킵니다.

위에 나열된 연산 외에도, *timedelta* 객체는 *date*와 *datetime* 객체와의 어떤 합과 차를 지원합니다 (아래를 참조하세요).

버전 3.2에서 변경: 나머지 연산과 *divmod()* 함수와 마찬가지로, *timedelta* 객체를 다른 *timedelta* 객체로 정수 나누기(floor division)와 실수 나누기(true division)가 이제 지원됩니다. *timedelta* 객체를 *float* 객체로 실수 나누기와 곱셈도 이제 이제 지원됩니다.

timedelta objects support equality and order comparisons.

불리언 문맥에서 *timedelta* 객체는 `timedelta(0)`와 같지 않을 때만 참으로 간주합니다.

인스턴스 메서드:

timedelta.**total_seconds**()

기간에 포함된 총 시간을 초(seconds)로 반환합니다. `td / timedelta(seconds=1)`와 동등합니다. 초 이외의 구간 단위에는, 나누기 형식을 직접 사용하십시오 (예를 들어, `td / timedelta(microseconds=1)`).

매우 큰 시간 구간에서는 (대부분 플랫폼에서 270년 이상), 이 메서드는 마이크로초의 정확도를 잃게 됩니다.

Added in version 3.2.

사용 예: timedelta

정규화의 추가 예:

```
>>> # Components of another_year add up to exactly 365 days
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)
>>> year == another_year
True
>>> year.total_seconds()
31536000.0
```

timedelta 산술의 예:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> ten_years = 10 * year
>>> ten_years
datetime.timedelta(days=3650)
>>> ten_years.days // 365
10
>>> nine_years = ten_years - year
>>> nine_years
datetime.timedelta(days=3285)
>>> three_years = nine_years // 3
>>> three_years, three_years.days // 365
(datetime.timedelta(days=1095), 3)
```

8.1.5 date 객체

date 객체는 현재의 그레고리력을 무한히 양방향으로 확장한, 이상적인 달력에서의 날짜(년, 월, 일)를 나타냅니다.

1년 1월 1일을 날 번호 1, 1년 1월 2일을 날 번호 2라고 부르고, 이런 식으로 계속됩니다.[2]

class datetime.**date**(*year*, *month*, *day*)

모든 인자가 필수입니다. 인자는 다음 범위에 있는 정수이어야 합니다:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= day <=` 주어진 `month`와 `year`에서의 날 수

이 범위를 벗어나는 인자가 주어지면, *ValueError*가 발생합니다.

다른 생성자, 모든 클래스 메서드:

[2] 이것은 Dershowitz와 Reingold의 책 *Calendrical Calculations*에 나오는 "역산 그레고리(proleptic Gregorian)" 달력의 정의와 일치합니다. 이 달력은 모든 계산의 기본 달력입니다. 역산 그레고리력 서수(ordinal)와 다른 많은 달력 시스템 사이의 변환을 위한 알고리즘에 관해서는 이 책을 참조하십시오.

classmethod date.**today**()

현재 지역 날짜를 반환합니다.

이것은 `date.fromtimestamp(time.time())`과 동등합니다.

classmethod date.**fromtimestamp**(*timestamp*)

`time.time()`에 의해 반환된 것과 같은 POSIX 타임스탬프에 해당하는 지역 날짜를 반환합니다.

타임스탬프가 플랫폼 C `localtime()` 함수에서 지원하는 값 범위를 벗어나면 `OverflowError`가 발생하고, `localtime()` 실패 시 `OSError`가 발생합니다. 이것이 1970년에서 2038년으로 제한되는 것이 일반적입니다. 타임스탬프라는 개념에 윤초를 포함하는 POSIX가 아닌 시스템에서는, 윤초가 `fromtimestamp()`에서 무시됨에 유의하십시오.

버전 3.3에서 변경: timestamp가 플랫폼 C `localtime()` 함수에서 지원하는 값 범위를 벗어나면 `ValueError` 대신 `OverflowError`를 발생시킵니다. `localtime()` 실패 시 `ValueError` 대신 `OSError`를 발생시킵니다.

classmethod date.**fromordinal**(*ordinal*)

역산 그레고리력 서수에 해당하는 date를 반환합니다. 1년 1월 1일이 서수 1입니다.

`ValueError` is raised unless `1 <= ordinal <= date.max.toordinal()`. For any date `d`, `date.fromordinal(d.toordinal()) == d`.

classmethod date.**fromisoformat**(*date_string*)

Return a `date` corresponding to a *date_string* given in any valid ISO 8601 format, with the following exceptions:

1. Reduced precision dates are not currently supported (`YYYY-MM`, `YYYY`).
2. Extended date representations are not currently supported (`±YYYYYY-MM-DD`).
3. Ordinal dates are not currently supported (`YYYY-OOO`).

예제:

```
>>> from datetime import date
>>> date.fromisoformat('2019-12-04')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('20191204')
datetime.date(2019, 12, 4)
>>> date.fromisoformat('2021-W01-1')
datetime.date(2021, 1, 4)
```

Added in version 3.7.

버전 3.11에서 변경: Previously, this method only supported the format `YYYY-MM-DD`.

classmethod date.**fromisocalendar**(*year*, *week*, *day*)

년, 주 및 일로 지정된 ISO 달력 날짜에 해당하는 `date`를 반환합니다. 이것은 함수 `date.isocalendar()`의 역입니다.

Added in version 3.8.

classmethod date.**strptime**(*date_string*, *format*)

Return a `date` corresponding to *date_string*, parsed according to *format*. This is equivalent to:

```
date(*(time.strptime(date_string, format)[0:3]))
```

`ValueError` is raised if the date_string and format can' t be parsed by `time.strptime()` or if it returns a value which isn' t a time tuple. See also *strftime() and strptime() Behavior* and `date.fromisoformat()`.

> **참고**
>
> If *format* specifies a day of month without a year a `DeprecationWarning` is emitted. This is to avoid a quadrennial leap year bug in code seeking to parse only a month and day as the default year used in

absence of one in the format is not a leap year. Such *format* values may raise an error as of Python 3.15. The workaround is to always include a year in your *format*. If parsing *date_string* values that do not have a year, explicitly add a year that is a leap year before parsing:

```
>>> from datetime import date
>>> date_string = "02/29"
>>> when = date.strptime(f"{date_string};1984", "%m/%d;%Y")  # Avoids leap␣
↪year bug.
>>> when.strftime("%B %d")
'February 29'
```

Added in version 3.14.

클래스 어트리뷰트:

`date.`**`min`**

표현 가능한 가장 이른 date, `date(MINYEAR, 1, 1)`.

`date.`**`max`**

표현 가능한 가장 늦은 date, `date(MAXYEAR, 12, 31)`.

`date.`**`resolution`**

같지 않은 date 객체 간의 가능한 가장 작은 차이, `timedelta(days=1)`.

인스턴스 어트리뷰트 (읽기 전용):

`date.`**`year`**

*`MINYEAR`*와 *`MAXYEAR`* 사이, 경계 포함.

`date.`**`month`**

1과 12 사이, 경계 포함.

`date.`**`day`**

1과 주어진 year의 주어진 month의 날 수 사이.

지원되는 연산:

연산	결과
`date2 = date1 + timedelta`	`date2` will be `timedelta.days` days after `date1`. (1)
`date2 = date1 - timedelta`	Computes `date2` such that `date2 + timedelta == date1`. (2)
`timedelta = date1 - date2`	(3)
`date1 == date2` `date1 != date2`	Equality comparison. (4)

`date1 < date2`
`date1 > date2`
`date1 <= date2`
`date1 >= date2`
	Order comparison. (5)

노트:

(1) *date2*는 `timedelta.days > 0`이면 미래로, `timedelta.days < 0`이면 과거로 이동합니다. 결국 `date2 - date1 == timedelta.days`이 됩니다. `timedelta.seconds`와 `timedelta.`

microseconds는 무시됩니다. date2.year가 MINYEAR보다 작거나 MAXYEAR보다 크게 되려고 하면 OverflowError가 발생합니다.

(2) timedelta.seconds와 timedelta.microseconds는 무시됩니다.

(3) This is exact, and cannot overflow. timedelta.seconds and timedelta.microseconds are 0, and date2 + timedelta == date1 after.

(4) date objects are equal if they represent the same date.

date objects that are not also datetime instances are never equal to datetime objects, even if they represent the same date.

(5) *date1* is considered less than *date2* when *date1* precedes *date2* in time. In other words, date1 < date2 if and only if date1.toordinal() < date2.toordinal().

Order comparison between a date object that is not also a datetime instance and a datetime object raises TypeError.

버전 3.13에서 변경: Comparison between datetime object and an instance of the date subclass that is not a datetime subclass no longer converts the latter to date, ignoring the time part and the time zone. The default behavior can be changed by overriding the special comparison methods in subclasses.

불리언 문맥에서, 모든 date 객체는 참으로 간주합니다.

인스턴스 메서드:

date.**replace**(*year=self.year*, *month=self.month*, *day=self.day*)

Return a new date object with the same values, but with specified parameters updated.

예제:

```
>>> from datetime import date
>>> d = date(2002, 12, 31)
>>> d.replace(day=26)
datetime.date(2002, 12, 26)
```

The generic function copy.replace() also supports date objects.

date.**timetuple**()

time.localtime()이 반환하는 것과 같은 time.struct_time을 반환합니다.

시, 분 및 초는 0이고, DST 플래그는 -1입니다.

d.timetuple()은 다음과 동등합니다:

```
time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))
```

where yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 is the day number within the current year starting with 1 for January 1st.

date.**toordinal**()

Return the proleptic Gregorian ordinal of the date, where January 1 of year 1 has ordinal 1. For any date object d, date.fromordinal(d.toordinal()) == d.

date.**weekday**()

정수로 요일을 반환합니다. 월요일은 0이고 일요일은 6입니다. 예를 들어, date(2002, 12, 4).weekday() == 2, 수요일. isoweekday()도 참조하십시오.

date.**isoweekday**()

정수로 요일을 반환합니다. 월요일은 1이고 일요일은 7입니다. 예를 들어, date(2002, 12, 4).isoweekday() == 3, 수요일. weekday(), isocalendar()도 참조하십시오.

date.**isocalendar**()

세 개의 구성 요소가 있는 네임드 튜플 객체를 반환합니다: year, week 및 weekday.

ISO 달력은 널리 사용되는 그레고리력의 변형입니다.[3]

ISO 연도는 52나 53개의 완전한 주로 구성되고, 주는 월요일에 시작하여 일요일에 끝납니다. ISO 연도의 첫 번째 주는 그 해의 (그레고리) 달력에서 목요일이 들어있는 첫 번째 주입니다. 이것을 주 번호 1이라고 하며, 그 목요일의 ISO 연도는 그레고리 연도와 같습니다.

예를 들어, 2004년은 목요일에 시작되므로, ISO 연도 2004의 첫 주는 월요일, 2003년 12월 29일에 시작하고, 일요일, 2004년 1월 4일에 끝납니다:

```
>>> from datetime import date
>>> date(2003, 12, 29).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=1)
>>> date(2004, 1, 4).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=7)
```

버전 3.9에서 변경: 결과가 튜플에서 네임드 튜플로 변경되었습니다.

date.**isoformat**()

ISO 8601 형식으로 날짜를 나타내는 문자열을 반환합니다, YYYY-MM-DD:

```
>>> from datetime import date
>>> date(2002, 12, 4).isoformat()
'2002-12-04'
```

date.**__str__**()

For a date d, str(d) is equivalent to d.isoformat().

date.**ctime**()

날짜를 나타내는 문자열을 반환합니다:

```
>>> from datetime import date
>>> date(2002, 12, 4).ctime()
'Wed Dec  4 00:00:00 2002'
```

d.ctime() 은 다음과:

```
time.ctime(time.mktime(d.timetuple()))
```

네이티브 C ctime() 함수(*time.ctime()* 은 호출하지만 *date.ctime()* 은 호출하지 않습니다)가 C 표준을 준수하는 플랫폼에서 동등합니다.

date.**strftime**(*format*)

Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values. See also *strftime() and strptime() Behavior* and *date.isoformat()*.

date.**__format__**(*format*)

Same as *date.strftime()*. This makes it possible to specify a format string for a *date* object in formatted string literals and when using *str.format()*. See also *strftime() and strptime() Behavior* and *date.isoformat()*.

사용 예: date

이벤트까지 남은 날 수 계산 예:

[3] See R. H. van Gent' s guide to the mathematics of the ISO 8601 calendar for a good explanation.

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
...
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

*date*로 작업하는 추가 예:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)

>>> # Methods related to formatting string output
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> d.ctime()
'Mon Mar 11 00:00:00 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'

>>> # Methods for to extracting 'components' under different calendars
>>> t = d.timetuple()
>>> for i in t:
...     print(i)
2002                # year
3                   # month
11                  # day
0
0
0
0                   # weekday (0 = Monday)
70                  # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print(i)
2002                # ISO year
11                  # ISO week number
1                   # ISO day number ( 1 = Monday )

>>> # A date object is immutable; all operations produce a new object
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> d.replace(year=2005)
datetime.date(2005, 3, 11)
```

8.1.6 datetime 객체

datetime 객체는 *date* 객체와 *time* 객체의 모든 정보를 포함하는 단일 객체입니다.

date 객체와 마찬가지로, *datetime*은 현재의 그레고리력을 양방향으로 확장한다고 가정합니다; *time* 객체와 마찬가지로, *datetime*은 하루가 정확히 3600*24초인 것으로 가정합니다.

생성자:

class datetime.**datetime**(*year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None, *, fold=0*)

year, month, day 인자는 필수입니다. *tzinfo*는 None이거나 *tzinfo* 서브 클래스의 인스턴스일 수 있습니다. 나머지 인자는 다음 범위의 정수이어야 합니다:

- MINYEAR <= year <= MAXYEAR,
- 1 <= month <= 12,
- 1 <= day <= 주어진 month와 year에서의 날 수,
- 0 <= hour < 24,
- 0 <= minute < 60,
- 0 <= second < 60,
- 0 <= microsecond < 1000000,
- fold in [0, 1].

이 범위를 벗어나는 인자가 주어지면, *ValueError*가 발생합니다.

버전 3.6에서 변경: Added the *fold* parameter.

다른 생성자, 모든 클래스 메서드:

classmethod datetime.**today**()

Return the current local date and time, with *tzinfo* None.

다음과 동등합니다:

```
datetime.fromtimestamp(time.time())
```

now(), *fromtimestamp()*를 참조하십시오.

이 메서드는 기능적으로 *now()*와 동등하지만, tz 매개 변수는 없습니다.

classmethod datetime.**now**(*tz=None*)

현재의 지역 날짜와 시간을 반환합니다.

선택적 인자 *tz*가 None이거나 지정되지 않으면, *today()*와 유사합니다. 하지만, 가능하면 *time.time()* 타임스탬프를 통해 얻을 수 있는 것보다 더 높은 정밀도를 제공합니다 (예를 들어, C gettimeofday() 함수를 제공하는 플랫폼에서 가능합니다).

*tz*가 None이 아니면, *tzinfo* 서브 클래스의 인스턴스여야 하며, 현재 날짜와 시간이 *tz*의 시간대로 변환됩니다.

이 함수는 *today()*와 *utcnow()*보다 선호됩니다.

> **참고**
>
> Subsequent calls to `datetime.now()` may return the same instant depending on the precision of the underlying clock.

classmethod datetime.**utcnow**()

*tzinfo*가 None인 현재 UTC 날짜와 시간을 반환합니다.

이것은 *now()*와 비슷하지만, 현재의 UTC 날짜와 시간을 나이브 *datetime* 객체로 반환합니다. 현재 어웨어 UTC datetime은 `datetime.now(timezone.utc)`를 호출하여 얻을 수 있습니다. *now()*도 참조하십시오.

> **경고**
>
> 나이브 `datetime` 객체는 많은 `datetime` 메서드에 의해 지역 시간으로 취급되므로, UTC로 시간을 나타내는 어웨어 datetime을 사용하는 것이 좋습니다. 따라서 UTC로 현재 시간을 나타내는 객체를 만드는 권장 방법은 `datetime.now(timezone.utc)`를 호출하는 것입니다.

버전 3.12부터 폐지됨: Use *datetime.now()* with *UTC* instead.

classmethod datetime.**fromtimestamp**(*timestamp*, *tz=None*)

*time.time()*가 반환하는 것과 같은, POSIX timestamp에 해당하는 지역 날짜와 시간을 반환합니다. 선택적 인자 *tz*가 None이거나 지정되지 않으면 timestamp는 플랫폼의 지역 날짜와 시간으로 변환되며, 반환된 *datetime* 객체는 나이브합니다.

*tz*가 None이 아니면, *tzinfo* 서브 클래스의 인스턴스여야 하며, timestamp는 *tz*의 시간대로 변환됩니다.

timestamp가 플랫폼 C `localtime()` 이나 `gmtime()` 함수에서 지원하는 값 범위를 벗어나면 *fromtimestamp()*가 *OverflowError*를 발생시킬 수 있고, `localtime()` 이나 `gmtime()`이 실패하면 *OSError*를 발생시킬 수 있습니다. 1970년에서 2038년까지로 제한되는 것이 일반적입니다. 타임스탬프에 윤초 개념을 포함하는 비 POSIX 시스템에서, *fromtimestamp()*는 윤초를 무시하므로, 1초 차이가 나는 두 개의 타임스탬프가 같은 *datetime* 객체를 산출할 수 있습니다. 이 방법은 *utcfromtimestamp()*보다 선호됩니다.

버전 3.3에서 변경: timestamp가 플랫폼 C `localtime()` 이나 `gmtime()` 함수에서 지원하는 값 범위를 벗어나면 *ValueError* 대신 *OverflowError*를 발생시킵니다. `localtime()` 이나 `gmtime()`이 실패하면 *ValueError* 대신 *OSError*를 발생시킵니다.

버전 3.6에서 변경: *fromtimestamp()*는 *fold*가 1로 설정된 인스턴스를 반환할 수 있습니다.

classmethod datetime.**utcfromtimestamp**(*timestamp*)

*tzinfo*가 None인 POSIX timestamp에 해당하는 UTC *datetime*을 반환합니다. (결과 객체는 나이브합니다.)

timestamp가 플랫폼 C `gmtime()` 함수에서 지원하는 값 범위를 벗어나면 *OverflowError*가 발생하고, `gmtime()`이 실패하면 *OSError*가 발생합니다. 1970년에서 2038년까지로 제한되는 것이 일반적입니다.

어웨어 *datetime* 객체를 얻으려며, *fromtimestamp()*를 호출하십시오:

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

POSIX 호환 플랫폼에서, 다음 표현식과 동등합니다:

```
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
```

단, 후자의 식은 항상 전체 연도 범위를 지원합니다: *MINYEAR*와 *MAXYEAR* 사이, 경계 포함.

> **경고**
>
> 나이브 `datetime` 객체는 많은 `datetime` 메서드에 의해 지역 시간으로 취급되므로. UTC로 시간을 나타내는 어웨어 datetime을 사용하는 것이 좋습니다. 따라서 UTC로 특정 timestamp를 나타내는 객체를 만드는 권장 방법은 `datetime.fromtimestamp(timestamp, tz=timezone.utc)`를 호출하는 것입니다.

버전 3.3에서 변경: timestamp가 플랫폼 C `gmtime()` 함수에서 지원하는 값 범위를 벗어나면 `ValueError` 대신 `OverflowError`를 발생시킵니다. `gmtime()`이 실패하면 `ValueError` 대신 `OSError`를 발생시킵니다.

버전 3.12부터 폐지됨: Use `datetime.fromtimestamp()` with `UTC` instead.

classmethod `datetime.`**`fromordinal`**(*ordinal*)

역산 그레고리력 서수(ordinal)에 해당하는 `datetime`을 반환합니다. 1년 1월 1일이 서수 1입니다. `1 <= ordinal <= datetime.max.toordinal()`이 아니면 `ValueError`가 발생합니다. 결과의 hour, minute, second 및 microsecond는 모두 0이고, `tzinfo`는 `None`입니다.

classmethod `datetime.`**`combine`**(*date*, *time*, *tzinfo=time.tzinfo*)

Return a new `datetime` object whose date components are equal to the given `date` object' s, and whose time components are equal to the given `time` object' s. If the *tzinfo* argument is provided, its value is used to set the `tzinfo` attribute of the result, otherwise the `tzinfo` attribute of the *time* argument is used. If the *date* argument is a `datetime` object, its time components and `tzinfo` attributes are ignored.

For any `datetime` object `d`, `d == datetime.combine(d.date(), d.time(), d.tzinfo)`.

버전 3.6에서 변경: *tzinfo* 인자가 추가되었습니다.

classmethod `datetime.`**`fromisoformat`**(*date_string*)

Return a `datetime` corresponding to a *date_string* in any valid ISO 8601 format, with the following exceptions:

1. Time zone offsets may have fractional seconds.
2. The `T` separator may be replaced by any single unicode character.
3. Fractional hours and minutes are not supported.
4. Reduced precision dates are not currently supported (`YYYY-MM`, `YYYY`).
5. Extended date representations are not currently supported (`±YYYYYY-MM-DD`).
6. Ordinal dates are not currently supported (`YYYY-OOO`).

예제:

```
>>> from datetime import datetime
>>> datetime.fromisoformat('2011-11-04')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('20111104')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('2011-11-04T00:05:23')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-11-04T00:05:23Z')
datetime.datetime(2011, 11, 4, 0, 5, 23, tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('20111104T000523')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-W01-2T00:05:23.283')
datetime.datetime(2011, 1, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283+00:00')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000, tzinfo=datetime.timezone.utc)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> datetime.fromisoformat('2011-11-04T00:05:23+04:00')
datetime.datetime(2011, 11, 4, 0, 5, 23,
    tzinfo=datetime.timezone(datetime.timedelta(seconds=14400)))
```

Added in version 3.7.

버전 3.11에서 변경: Previously, this method only supported formats that could be emitted by `date.isoformat()` or `datetime.isoformat()`.

classmethod datetime.**fromisocalendar**(*year*, *week*, *day*)

년, 주 및 일로 지정된 ISO 달력 날짜에 해당하는 `datetime`을 반환합니다. datetime의 날짜가 아닌 구성 요소는 일반적인 기본값으로 채워집니다. 이것은 함수 `datetime.isocalendar()`의 역입니다.

Added in version 3.8.

classmethod datetime.**strptime**(*date_string*, *format*)

*format*에 따라 구문 분석된, *date_string*에 해당하는 `datetime`를 반환합니다.

If *format* does not contain microseconds or time zone information, this is equivalent to:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

`ValueError` is raised if the date_string and format can't be parsed by `time.strptime()` or if it returns a value which isn't a time tuple. See also *strftime() and strptime() Behavior* and `datetime.fromisoformat()`.

버전 3.13에서 변경: If *format* specifies a day of month without a year a `DeprecationWarning` is now emitted. This is to avoid a quadrennial leap year bug in code seeking to parse only a month and day as the default year used in absence of one in the format is not a leap year. Such *format* values may raise an error as of Python 3.15. The workaround is to always include a year in your *format*. If parsing *date_string* values that do not have a year, explicitly add a year that is a leap year before parsing:

```
>>> from datetime import datetime
>>> date_string = "02/29"
>>> when = datetime.strptime(f"{date_string};1984", "%m/%d;%Y")  # Avoids leap␣
↪year bug.
>>> when.strftime("%B %d")
'February 29'
```

클래스 어트리뷰트:

datetime.**min**

표현 가능한 가장 이른 `datetime`, `datetime(MINYEAR, 1, 1, tzinfo=None)`.

datetime.**max**

표현 가능한 가장 늦은 `datetime`, `datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)`.

datetime.**resolution**

같지 않은 `datetime` 객체 간의 가능한 가장 작은 차이, `timedelta(microseconds=1)`.

인스턴스 어트리뷰트 (읽기 전용):

datetime.**year**

`MINYEAR`와 `MAXYEAR` 사이, 경계 포함.

datetime.**month**

1과 12 사이, 경계 포함.

datetime.**day**

1과 주어진 year의 주어진 month의 날 수 사이.

datetime.**hour**

범위 `range(24)`.

datetime.**minute**

범위 `range(60)`.

datetime.**second**

범위 `range(60)`.

datetime.**microsecond**

범위 `range(1000000)`.

datetime.**tzinfo**

datetime 생성자에 *tzinfo* 인자로 전달된 객체이거나, 전달되지 않았으면 `None` 입니다.

datetime.**fold**

In `[0, 1]`. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The values 0 and 1 represent, respectively, the earlier and later of the two moments with the same wall time representation.

Added in version 3.6.

지원되는 연산:

연산	결과
`datetime2 = datetime1 + timedelta`	(1)
`datetime2 = datetime1 - timedelta`	(2)
`timedelta = datetime1 - datetime2`	(3)
`datetime1 == datetime2` `datetime1 != datetime2`	Equality comparison. (4)

`datetime1 < datetime2`
`datetime1 > datetime2`
`datetime1 <= datetime2`
`datetime1 >= datetime2`
	Order comparison. (5)

(1) `datetime2` is a duration of `timedelta` removed from `datetime1`, moving forward in time if `timedelta.days > 0`, or backward if `timedelta.days < 0`. The result has the same *tzinfo* attribute as the input datetime, and `datetime2 - datetime1 == timedelta` after. *OverflowError* is raised if `datetime2.year` would be smaller than *MINYEAR* or larger than *MAXYEAR*. Note that no time zone adjustments are done even if the input is an aware object.

(2) Computes the `datetime2` such that `datetime2 + timedelta == datetime1`. As for addition, the result has the same *tzinfo* attribute as the input datetime, and no time zone adjustments are done even if the input is aware.

(3) *datetime*에서 *datetime*을 빼는 것은 두 피연산자 모두 나이브하거나, 모두 어웨어할 때만 정의됩니다. 하나가 어웨어이고 다른 하나가 나이브면, *TypeError*가 발생합니다.

If both are naive, or both are aware and have the same *tzinfo* attribute, the *tzinfo* attributes are ignored, and the result is a *timedelta* object `t` such that `datetime2 + t == datetime1`. No time zone adjustments are done in this case.

If both are aware and have different *tzinfo* attributes, `a-b` acts as if `a` and `b` were first converted to naive UTC datetimes. The result is `(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` except that the implementation never overflows.

(4) *datetime* objects are equal if they represent the same date and time, taking into account the time zone.

Naive and aware `datetime` objects are never equal.

If both comparands are aware, and have the same `tzinfo` attribute, the `tzinfo` and *fold* attributes are ignored and the base datetimes are compared. If both comparands are aware and have different *tzinfo* attributes, the comparison acts as comparands were first converted to UTC datetimes except that the implementation never overflows. `datetime` instances in a repeated interval are never equal to `datetime` instances in other time zone.

(5) *datetime1* is considered less than *datetime2* when *datetime1* precedes *datetime2* in time, taking into account the time zone.

Order comparison between naive and aware *datetime* objects raises *TypeError*.

If both comparands are aware, and have the same `tzinfo` attribute, the `tzinfo` and *fold* attributes are ignored and the base datetimes are compared. If both comparands are aware and have different *tzinfo* attributes, the comparison acts as comparands were first converted to UTC datetimes except that the implementation never overflows.

버전 3.3에서 변경: 어웨어와 나이브 *datetime* 인스턴스 간의 동등 비교는 *TypeError*를 발생시키지 않습니다.

버전 3.13에서 변경: Comparison between *datetime* object and an instance of the *date* subclass that is not a `datetime` subclass no longer converts the latter to `date`, ignoring the time part and the time zone. The default behavior can be changed by overriding the special comparison methods in subclasses.

인스턴스 메서드:

`datetime.`**`date`**`()`

같은 year, month, day의 *date* 객체를 반환합니다.

`datetime.`**`time`**`()`

같은 hour, minute, second, microsecond 및 fold의 *time* 객체를 반환합니다. *tzinfo*는 `None`입니다. 메서드 *timetz()*도 참조하십시오.

버전 3.6에서 변경: fold 값은 반환된 *time* 객체에 복사됩니다.

`datetime.`**`timetz`**`()`

같은 hour, minute, second, microsecond, fold 및 tzinfo 어트리뷰트의 *time* 객체를 반환합니다. 메서드 *time()*도 참조하십시오.

버전 3.6에서 변경: fold 값은 반환된 *time* 객체에 복사됩니다.

`datetime.`**`replace`**`(`*year=self.year, month=self.month, day=self.day, hour=self.hour, minute=self.minute, second=self.second, microsecond=self.microsecond, tzinfo=self.tzinfo, *, fold=0*`)`

Return a new *datetime* object with the same attributes, but with specified parameters updated. Note that `tzinfo=None` can be specified to create a naive datetime from an aware datetime with no conversion of date and time data.

datetime objects are also supported by generic function *copy.replace()*.

버전 3.6에서 변경: Added the *fold* parameter.

`datetime.`**`astimezone`**`(`*tz=None*`)`

새로운 *tzinfo* 어트리뷰트 *tz*를 갖는 *datetime* 객체를 반환하는데, 결과가 *self*와 같은 UTC 시간이지만 *tz*의 지역 시간이 되도록 날짜와 시간 데이터를 조정합니다.

If provided, *tz* must be an instance of a *tzinfo* subclass, and its *utcoffset()* and *dst()* methods must not return `None`. If *self* is naive, it is presumed to represent time in the system time zone.

If called without arguments (or with `tz=None`) the system local time zone is assumed for the target time zone. The `.tzinfo` attribute of the converted datetime instance will be set to an instance of *timezone* with the zone name and offset obtained from the OS.

If `self.tzinfo` is *tz*, `self.astimezone(tz)` is equal to *self*: no adjustment of date or time data is performed. Else the result is local time in the time zone *tz*, representing the same UTC time as *self*: after `astz`

= dt.astimezone(tz), astz - astz.utcoffset() will have the same date and time data as dt - dt.utcoffset().

If you merely want to attach a *timezone* object *tz* to a datetime *dt* without adjustment of date and time data, use dt.replace(tzinfo=tz). If you merely want to remove the timezone object from an aware datetime *dt* without conversion of date and time data, use dt.replace(tzinfo=None).

기본 *tzinfo.fromutc()* 메서드는 *astimezone()*에 의해 반환된 결과에 영향을 주도록 *tzinfo* 서브 클래스에서 재정의할 수 있습니다. 에러가 발생하는 경우를 무시하고, *astimezone()*는 다음과 같이 작동합니다:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new timezone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

버전 3.3에서 변경: 이제 *tz*를 생략할 수 있습니다.

버전 3.6에서 변경: 이제 *astimezone()* 메서드는 이제 나이브 인스턴스에서 호출될 수 있는데, 시스템 지역 시간을 나타내는 것으로 간주합니다.

datetime.**utcoffset**()

*tzinfo*가 None이면, None을 반환하고, 그렇지 않으면 self.tzinfo.utcoffset(self)를 반환하고, 후자가 None이나 하루 미만의 크기를 가진 *timedelta* 객체를 반환하지 않으면 예외를 발생시킵니다.

버전 3.7에서 변경: UTC 오프셋은 분 단위로 제한되지 않습니다.

datetime.**dst**()

*tzinfo*가 None이면, None을 반환하고, 그렇지 않으면 self.tzinfo.dst(self)를 반환하고, 후자가 None이나 하루 미만의 크기를 가진 *timedelta* 객체를 반환하지 않으면 예외를 발생시킵니다.

버전 3.7에서 변경: DST 오프셋은 분 단위로 제한되지 않습니다.

datetime.**tzname**()

*tzinfo*가 None이면, None을 반환하고, 그렇지 않으면 self.tzinfo.tzname(self)를 반환하고, 후자가 None이나 문자열 객체를 반환하지 않으면 예외를 발생시킵니다.

datetime.**timetuple**()

*time.localtime()*이 반환하는 것과 같은 *time.struct_time*을 반환합니다.

d.timetuple()은 다음과 동등합니다:

```
time.struct_time((d.year, d.month, d.day,
                  d.hour, d.minute, d.second,
                  d.weekday(), yday, dst))
```

where yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 is the day number within the current year starting with 1 for January 1st. The *tm_isdst* flag of the result is set according to the *dst()* method: *tzinfo* is None or *dst()* returns None, tm_isdst is set to -1; else if *dst()* returns a non-zero value, tm_isdst is set to 1; else tm_isdst is set to 0.

datetime.**utctimetuple**()

If *datetime* instance d is naive, this is the same as d.timetuple() except that *tm_isdst* is forced to 0 regardless of what d.dst() returns. DST is never in effect for a UTC time.

If d is aware, d is normalized to UTC time, by subtracting d.utcoffset(), and a *time.struct_time* for the normalized time is returned. tm_isdst is forced to 0. Note that an *OverflowError* may be raised if d.year was MINYEAR or MAXYEAR and UTC adjustment spills over a year boundary.

> **경고**
>
> Because naive `datetime` objects are treated by many `datetime` methods as local times, it is preferred to use aware datetimes to represent times in UTC; as a result, using *datetime.utctimetuple()* may give misleading results. If you have a naive `datetime` representing UTC, use `datetime.replace(tzinfo=timezone.utc)` to make it aware, at which point you can use *datetime.timetuple()*.

`datetime.`**`toordinal`**`()`

날짜의 역산 그레고리력 서수를 반환합니다. `self.date().toordinal()`과 같습니다.

`datetime.`**`timestamp`**`()`

datetime 인스턴스에 해당하는 POSIX 타임스탬프를 반환합니다. 반환 값은 *time.time()*이 반환하는 것과 비슷한 *float*입니다.

Naive *datetime* instances are assumed to represent local time and this method relies on the platform C `mktime()` function to perform the conversion. Since *datetime* supports wider range of values than `mktime()` on many platforms, this method may raise *OverflowError* or *OSError* for times far in the past or far in the future.

어웨어 *datetime* 인스턴스의 경우, 반환 값은 다음과 같이 계산됩니다:

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

Added in version 3.3.

버전 3.6에서 변경: *timestamp()* 메서드는 *fold* 어트리뷰트를 사용하여 반복되는 구간의 시간을 구분합니다.

> **참고**
>
> There is no method to obtain the POSIX timestamp directly from a naive *datetime* instance representing UTC time. If your application uses this convention and your system time zone is not set to UTC, you can obtain the POSIX timestamp by supplying `tzinfo=timezone.utc`:
>
> ```
> timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
> ```
>
> 또는 직접 타임스탬프를 계산할 수 있습니다:
>
> ```
> timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
> ```

`datetime.`**`weekday`**`()`

정수로 요일을 반환합니다. 월요일은 0이고 일요일은 6입니다. `self.date().weekday()`와 같습니다. *isoweekday()*도 참조하십시오.

`datetime.`**`isoweekday`**`()`

정수로 요일을 반환합니다. 월요일은 1이고 일요일은 7입니다. `self.date().isoweekday()` 와 같습니다. *weekday()*, *isocalendar()*도 참조하십시오.

`datetime.`**`isocalendar`**`()`

세 개의 컴포넌트를 가진 네임드 튜플을 반환합니다: `year`, `week` 및 `weekday`. `self.date().isocalendar()`와 같습니다.

`datetime.`**`isoformat`**`(`*sep='T', timespec='auto'*`)`

ISO 8601 형식으로 날짜와 시간을 나타내는 문자열을 반환합니다:

- `YYYY-MM-DDTHH:MM:SS.ffffff`, *microsecond*가 0이 아니면
- `YYYY-MM-DDTHH:MM:SS`, *microsecond*가 0이면

*utcoffset()*이 `None`을 반환하지 않으면, UTC 오프셋을 제공하는 문자열을 덧붙입니다:

- YYYY-MM-DDTHH:MM:SS.ffffff+HH:MM[:SS[.ffffff]], *microsecond*가 0이 아니면
- YYYY-MM-DDTHH:MM:SS+HH:MM[:SS[.ffffff]], *microsecond*가 0이면

예제:

```
>>> from datetime import datetime, timezone
>>> datetime(2019, 5, 18, 15, 17, 8, 132263).isoformat()
'2019-05-18T15:17:08.132263'
>>> datetime(2019, 5, 18, 15, 17, tzinfo=timezone.utc).isoformat()
'2019-05-18T15:17:00+00:00'
```

선택적 인자 *sep*(기본값 'T')은 한 문자 구분자로, 결과의 날짜와 시간 부분 사이에 배치됩니다. 예를 들어:

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     """A time zone with an arbitrary, constant -06:39 offset."""
...     def utcoffset(self, dt):
...         return timedelta(hours=-6, minutes=-39)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
>>> datetime(2009, 11, 27, microsecond=100, tzinfo=TZ()).isoformat()
'2009-11-27T00:00:00.000100-06:39'
```

선택적 인자 *timespec*은 포함할 시간의 추가 구성 요소 수를 지정합니다 (기본값은 'auto'입니다). 다음 중 하나일 수 있습니다:

- 'auto': *microsecond*가 0이면 'seconds'와 같고, 그렇지 않으면 'microseconds'와 같습니다.
- 'hours': *hour*를 두 자리 숫자 HH 형식으로 포함합니다.
- 'minutes': *hour*와 *minute*를 HH:MM 형식으로 포함합니다.
- 'seconds': *hour*, *minute* 및 *second*를 HH:MM:SS 형식으로 포함합니다.
- 'milliseconds': 전체 시간을 포함하지만, 초 미만은 밀리초 단위로 자릅니다. HH:MM:SS.sss 형식입니다.
- 'microseconds': 전체 시간을 HH:MM:SS.ffffff 형식으로 포함합니다.

> **참고**
>
> 제외된 시간 구성 요소는 반올림되지 않고 잘립니다.

잘못된 *timespec* 인자는 *ValueError*를 발생시킵니다:

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

버전 3.6에서 변경: Added the *timespec* parameter.

datetime.**__str__**()

For a *datetime* instance d, str(d) is equivalent to d.isoformat(' ').

datetime.**ctime**()

날짜와 시간을 나타내는 문자열을 반환합니다:

```
>>> from datetime import datetime
>>> datetime(2002, 12, 4, 20, 30, 40).ctime()
'Wed Dec  4 20:30:40 2002'
```

출력 문자열은 입력이 어웨어인지 나이브인지와 관계없이 시간대 정보를 포함하지 않습니다.

`d.ctime()` 은 다음과:

```
time.ctime(time.mktime(d.timetuple()))
```

네이티브 C `ctime()` 함수(*time.ctime()* 이 호출하지만, *datetime.ctime()* 은 호출하지 않습니다)가 C 표준을 준수하는 플랫폼에서 동등합니다.

datetime.**strftime**(*format*)

Return a string representing the date and time, controlled by an explicit format string. See also *strftime() and strptime() Behavior* and *datetime.isoformat()*.

datetime.**__format__**(*format*)

Same as *datetime.strftime()*. This makes it possible to specify a format string for a *datetime* object in formatted string literals and when using *str.format()*. See also *strftime() and strptime() Behavior* and *datetime.isoformat()*.

사용 예: datetime

Examples of working with *datetime* objects:

```
>>> from datetime import datetime, date, time, timezone

>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)

>>> # Using datetime.now()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
>>> datetime.now(timezone.utc)
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060, tzinfo=datetime.timezone.utc)

>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)

>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None

>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
...
2006    # ISO year
47      # ISO week
2       # ISO weekday

>>> # Formatting a datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day
↪", "month", "time")
'The day is 21, the month is November, the time is 04:30PM.'
```

아래 예제는 1945년까지 +4 UTC를 사용한 후 +4:30 UTC로 변경한 아프가니스탄 카불의 시간대 정보를 캡처하는 *tzinfo* 서브 클래스를 정의합니다:

```
from datetime import timedelta, datetime, tzinfo, timezone

class KabulTz(tzinfo):
    # Kabul used +4 until 1945, when they moved to +4:30
    UTC_MOVE_DATE = datetime(1944, 12, 31, 20, tzinfo=timezone.utc)

    def utcoffset(self, dt):
        if dt.year < 1945:
            return timedelta(hours=4)
        elif (1945, 1, 1, 0, 0) <= dt.timetuple()[:5] < (1945, 1, 1, 0, 30):
            # An ambiguous ("imaginary") half-hour range representing
            # a 'fold' in time due to the shift from +4 to +4:30.
            # If dt falls in the imaginary range, use fold to decide how
            # to resolve. See PEP495.
            return timedelta(hours=4, minutes=(30 if dt.fold else 0))
        else:
            return timedelta(hours=4, minutes=30)

    def fromutc(self, dt):
        # Follow same validations as in datetime.tzinfo
        if not isinstance(dt, datetime):
            raise TypeError("fromutc() requires a datetime argument")
        if dt.tzinfo is not self:
            raise ValueError("dt.tzinfo is not self")

        # A custom implementation is required for fromutc as
        # the input to this function is a datetime with utc values
        # but with a tzinfo set to self.
        # See datetime.astimezone or fromtimestamp.
        if dt.replace(tzinfo=timezone.utc) >= self.UTC_MOVE_DATE:
            return dt + timedelta(hours=4, minutes=30)
        else:
            return dt + timedelta(hours=4)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    def dst(self, dt):
        # Kabul does not observe daylight saving time.
        return timedelta(0)

    def tzname(self, dt):
        if dt >= self.UTC_MOVE_DATE:
            return "+04:30"
        return "+04"
```

위의 `KabulTz` 사용법:

```
>>> tz1 = KabulTz()

>>> # Datetime before the change
>>> dt1 = datetime(1900, 11, 21, 16, 30, tzinfo=tz1)
>>> print(dt1.utcoffset())
4:00:00

>>> # Datetime after the change
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=tz1)
>>> print(dt2.utcoffset())
4:30:00

>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(timezone.utc)
>>> dt3
datetime.datetime(2006, 6, 14, 8, 30, tzinfo=datetime.timezone.utc)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=KabulTz())
>>> dt2 == dt3
True
```

8.1.7 `time` 객체

A *time* object represents a (local) time of day, independent of any particular day, and subject to adjustment via a *tzinfo* object.

class datetime.**time**(*hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

모든 인자는 선택적입니다. *tzinfo*는 `None`, 또는 *tzinfo* 서브 클래스의 인스턴스일 수 있습니다. 나머지 인자는 다음 범위의 정수이어야 합니다:

- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`.

If an argument outside those ranges is given, *ValueError* is raised. All default to 0 except *tzinfo*, which defaults to `None`.

클래스 어트리뷰트:

time.**min**

표현 가능한 가장 이른 *time*, `time(0, 0, 0, 0)`.

time.**max**

표현 가능한 가장 늦은 *time*, time(23, 59, 59, 999999).

time.**resolution**

같지 않은 *time* 객체 간의 가능한 가장 작은 차이, timedelta(microseconds=1), 하지만 *time* 객체에 대한 산술은 지원되지 않습니다.

인스턴스 어트리뷰트 (읽기 전용):

time.**hour**

범위 range(24).

time.**minute**

범위 range(60).

time.**second**

범위 range(60).

time.**microsecond**

범위 range(1000000).

time.**tzinfo**

time 생성자에 *tzinfo* 인자로 전달된 객체이거나, 전달되지 않았으면 None 입니다.

time.**fold**

In [0, 1]. Used to disambiguate wall times during a repeated interval. (A repeated interval occurs when clocks are rolled back at the end of daylight saving time or when the UTC offset for the current zone is decreased for political reasons.) The values 0 and 1 represent, respectively, the earlier and later of the two moments with the same wall time representation.

Added in version 3.6.

time objects support equality and order comparisons, where a is considered less than b when a precedes b in time.

Naive and aware time objects are never equal. Order comparison between naive and aware time objects raises *TypeError*.

If both comparands are aware, and have the same *tzinfo* attribute, the tzinfo and fold attributes are ignored and the base times are compared. If both comparands are aware and have different tzinfo attributes, the comparands are first adjusted by subtracting their UTC offsets (obtained from self.utcoffset()).

버전 3.3에서 변경: Equality comparisons between aware and naive *time* instances don' t raise *TypeError*.

불리언 문맥에서, *time* 객체는 항상 참으로 간주합니다.

버전 3.5에서 변경: 파이썬 3.5 이전에, *time* 객체는 UTC 자정을 나타낼 때 거짓으로 간주했습니다. 이 동작은 애매하고 에러가 발생하기 쉬운 것으로 간주하여 파이썬 3.5에서 제거되었습니다. 자세한 내용은 bpo-13936을 참조하십시오.

Other constructors:

classmethod time.**fromisoformat**(*time_string*)

Return a *time* corresponding to a *time_string* in any valid ISO 8601 format, with the following exceptions:

1. Time zone offsets may have fractional seconds.
2. The leading T, normally required in cases where there may be ambiguity between a date and a time, is not required.
3. Fractional seconds may have any number of digits (anything beyond 6 will be truncated).
4. Fractional hours and minutes are not supported.

Examples:

```
>>> from datetime import time
>>> time.fromisoformat('04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('T04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('T042301')
datetime.time(4, 23, 1)
>>> time.fromisoformat('04:23:01.000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01,000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01+04:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone(datetime.
↪timedelta(seconds=14400)))
>>> time.fromisoformat('04:23:01Z')
datetime.time(4, 23, 1, tzinfo=datetime.timezone.utc)
>>> time.fromisoformat('04:23:01+00:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone.utc)
```

Added in version 3.7.

버전 3.11에서 변경: Previously, this method only supported formats that could be emitted by `time.isoformat()`.

classmethod time.**strptime**(*date_string*, *format*)

Return a `time` corresponding to *date_string*, parsed according to *format*.

If *format* does not contain microseconds or timezone information, this is equivalent to:

```
time(*(time.strptime(date_string, format)[3:6]))
```

`ValueError` is raised if the *date_string* and *format* cannot be parsed by `time.strptime()` or if it returns a value which is not a time tuple. See also *strftime() and strptime() Behavior* and `time.fromisoformat()`.

Added in version 3.14.

인스턴스 메서드:

time.**replace**(*hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, **, fold=0*)

Return a new `time` with the same values, but with specified parameters updated. Note that `tzinfo=None` can be specified to create a naive `time` from an aware `time`, without conversion of the time data.

`time` objects are also supported by generic function `copy.replace()`.

버전 3.6에서 변경: Added the *fold* parameter.

time.**isoformat**(*timespec='auto'*)

ISO 8601 형식으로 시간을 나타내는 문자열을 반환합니다, 다음 중 한가지입니다:

- `HH:MM:SS.ffffff`, `microsecond`가 0이 아니면
- `HH:MM:SS`, `microsecond`가 0이면
- `HH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]`, `utcoffset()`이 `None`을 반환하지 않으면
- `HH:MM:SS+HH:MM[:SS[.ffffff]]`, `microsecond`가 0이고 `utcoffset()`이 `None`을 반환하지 않으면

선택적 인자 *timespec*은 포함할 시간의 추가 구성 요소 수를 지정합니다 (기본값은 `'auto'`입니다). 다음 중 하나일 수 있습니다:

- `'auto'`: `microsecond`가 0이면 `'seconds'`와 같고, 그렇지 않으면 `'microseconds'`와 같습니다.

- 'hours': *hour*를 두 자리 숫자 HH 형식으로 포함합니다.
- 'minutes': *hour*와 *minute*를 HH:MM 형식으로 포함합니다.
- 'seconds': *hour*, *minute* 및 *second*를 HH:MM:SS 형식으로 포함합니다.
- 'milliseconds': 전체 시간을 포함하지만, 초 미만은 밀리초 단위로 자릅니다. HH:MM:SS.sss 형식입니다.
- 'microseconds': 전체 시간을 HH:MM:SS.ffffff 형식으로 포함합니다.

> **참고**
>
> 제외된 시간 구성 요소는 반올림되지 않고 잘립니다.

잘못된 *timespec* 인자는 *ValueError*를 발생시킵니다.

예제:

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56, microsecond=123456).isoformat(timespec=
↪'minutes')
'12:34'
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

버전 3.6에서 변경: Added the *timespec* parameter.

time.**__str__**()

For a time t, str(t) is equivalent to t.isoformat().

time.**strftime**(*format*)

Return a string representing the time, controlled by an explicit format string. See also *strftime() and strptime() Behavior* and *time.isoformat()*.

time.**__format__**(*format*)

Same as *time.strftime()*. This makes it possible to specify a format string for a *time* object in formatted string literals and when using *str.format()*. See also *strftime() and strptime() Behavior* and *time.isoformat()*.

time.**utcoffset**()

*tzinfo*가 None이면, None을 반환하고, 그렇지 않으면 self.tzinfo.utcoffset(None)를 반환하고, 후자가 None이나 하루 미만의 크기를 가진 *timedelta* 객체를 반환하지 않으면 예외를 발생시킵니다.

버전 3.7에서 변경: UTC 오프셋은 분 단위로 제한되지 않습니다.

time.**dst**()

*tzinfo*가 None이면, None을 반환하고, 그렇지 않으면 self.tzinfo.dst(None)를 반환하고, 후자가 None이나 하루 미만의 크기를 가진 *timedelta* 객체를 반환하지 않으면 예외를 발생시킵니다.

버전 3.7에서 변경: DST 오프셋은 분 단위로 제한되지 않습니다.

time.**tzname**()

*tzinfo*가 None이면, None을 반환하고, 그렇지 않으면 self.tzinfo.tzname(None)를 반환하고, 후자가 None이나 문자열 객체를 반환하지 않으면 예외를 발생시킵니다.

사용 예: time

time 객체로 작업하는 예제:

```
>>> from datetime import time, tzinfo, timedelta
>>> class TZ1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "+01:00"
...     def  __repr__(self):
...         return f"{self.__class__.__name__}()"
...
>>> t = time(12, 10, 30, tzinfo=TZ1())
>>> t
datetime.time(12, 10, 30, tzinfo=TZ1())
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'+01:00'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 +01:00'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.8 tzinfo 객체

class datetime.**tzinfo**

이것은 추상 베이스 클래스입니다. 즉, 이 클래스를 직접 인스턴스로 만들면 안 됩니다. 특정 시간대에 대한 정보를 캡처하려면 *tzinfo*의 서브 클래스를 정의하십시오.

*tzinfo*의 (구상 서브 클래스의) 인스턴스는 *datetime*과 *time* 객체의 생성자에 전달될 수 있습니다. 이 객체들은 자신의 어트리뷰트를 지역 시간으로 간주하며, *tzinfo* 객체는 지역 시간의 UTC 로부터의 오프셋, 시간대 이름 및 DST 오프셋을 모두 전달된 날짜나 시간 객체에 상대적으로 얻는 메서드들을 지원합니다.

You need to derive a concrete subclass, and (at least) supply implementations of the standard *tzinfo* methods needed by the *datetime* methods you use. The `datetime` module provides *timezone*, a simple concrete subclass of *tzinfo* which can represent time zones with fixed offset from UTC such as UTC itself or North American EST and EDT.

Special requirement for pickling: A *tzinfo* subclass must have an `__init__()` method that can be called with no arguments, otherwise it can be pickled but possibly not unpickled again. This is a technical requirement that may be relaxed in the future.

A concrete subclass of *tzinfo* may need to implement the following methods. Exactly which methods are needed depends on the uses made of aware `datetime` objects. If in doubt, simply implement all of them.

tzinfo.**utcoffset**(*dt*)

지역 시간의 UTC로부터의 오프셋을 UTC의 동쪽에 있을 때 양의 값을 갖는 *timedelta* 객체로 반환합니다. 지역 시간이 UTC의 서쪽이면 이 값은 음수여야 합니다.

이것은 UTC로부터의 총 오프셋을 나타냅니다; 예를 들어, *tzinfo* 객체가 시간대와 DST 조정을 모두 나타내면, *utcoffset()*은 그들의 합계를 반환해야 합니다. UTC 오프셋을 알 수 없으면, `None`을 반환합니다. 그렇지 않으면 반환되는 값은 반드시 `-timedelta(hours=24)`와 `timedelta(hours=24)` 사이의 *timedelta* 객체여야 합니다 (오프셋의 크기는 하루 미만이어야 합니다). *utcoffset()*의 대부분 구현은 아마도 이 두 가지 중 하나일 것입니다:

```
return CONSTANT                 # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

*utcoffset()*이 None을 반환하지 않으면, *dst()*도 None을 반환하지 않아야 합니다.

*utcoffset()*의 기본 구현은 *NotImplementedError*를 발생시킵니다.

버전 3.7에서 변경: UTC 오프셋은 분 단위로 제한되지 않습니다.

tzinfo.**dst**(*dt*)

일광 절약 시간 (DST) 조정을 *timedelta* 객체로, 또는 DST 정보를 모르면 None을 반환합니다.

Return `timedelta(0)` if DST is not in effect. If DST is in effect, return the offset as a *timedelta* object (see *utcoffset()* for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by *utcoffset()*, so there' s no need to consult *dst()* unless you' re interested in obtaining DST info separately. For example, *datetime.timetuple()* calls its *tzinfo* attribute' s *dst()* method to determine how the *tm_isdst* flag should be set, and *tzinfo.fromutc()* calls *dst()* to account for DST changes when crossing time zones.

표준과 일광 절약 시간을 모두 모형화하는 *tzinfo* 서브 클래스의 인스턴스 *tz*는 다음과 같은 의미에서 일관되어야 합니다:

`tz.utcoffset(dt) - tz.dst(dt)`

must return the same result for every *datetime* *dt* with `dt.tzinfo == tz`. For sane *tzinfo* subclasses, this expression yields the time zone' s "standard offset", which should not depend on the date or the time, but only on geographic location. The implementation of *datetime.astimezone()* relies on this, but cannot detect violations; it' s the programmer' s responsibility to ensure it. If a *tzinfo* subclass cannot guarantee this, it may be able to override the default implementation of *tzinfo.fromutc()* to work correctly with *astimezone()* regardless.

*dst()*의 대부분 구현은 아마도 이 두 가지 중 하나일 것입니다:

```
def dst(self, dt):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

또는:

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time.

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

*dst()*의 기본 구현은 *NotImplementedError*를 발생시킵니다.

버전 3.7에서 변경: DST 오프셋은 분 단위로 제한되지 않습니다.

tzinfo.**tzname**(*dt*)

Return the time zone name corresponding to the *datetime* object *dt*, as a string. Nothing about string names is defined by the `datetime` module, and there' s no requirement that it mean anything in particular. For example, `"GMT"`, `"UTC"`, `"-500"`, `"-5:00"`, `"EDT"`, `"US/Eastern"`, `"America/New York"` are all valid replies. Return `None` if a string name isn' t known. Note that this is a method rather than a fixed string primarily because some *tzinfo* subclasses will wish to return different names depending on the specific value of *dt* passed, especially if the *tzinfo* class is accounting for daylight time.

*tzname()*의 기본 구현은 *NotImplementedError*를 발생시킵니다.

이 메서드들은 *datetime*나 *time* 객체에서 같은 이름의 메서드에 대한 응답으로 호출됩니다. *datetime* 객체는 자신을 인자로 전달하고, *time* 객체는 인자로 None을 전달합니다. 따라서 *tzinfo* 서브 클래스의 메서드는 None이나 *datetime* 클래스의 *dt* 인자를 받아들일 준비가 되어 있어야 합니다.

None이 전달되면, 최선의 응답을 결정하는 것은 클래스 설계자에게 달려있습니다. 예를 들어, 클래스가 *tzinfo* 프로토콜에 time 객체가 참여하지 않는다고 말하고 싶다면 None을 반환하는 것이 적절합니다. 표준 오프셋을 발견하는 다른 규칙이 없으므로, utcoffset(None) 이 표준 UTC 오프셋을 반환하는 것이 더 유용할 수 있습니다.

When a *datetime* object is passed in response to a *datetime* method, dt.tzinfo is the same object as *self*. *tzinfo* methods can rely on this, unless user code calls *tzinfo* methods directly. The intent is that the *tzinfo* methods interpret *dt* as being in local time, and not need worry about objects in other time zones.

서브 클래스가 재정의할 수 있는 *tzinfo* 메서드가 하나 더 있습니다:

tzinfo.**fromutc**(*dt*)

This is called from the default *datetime.astimezone()* implementation. When called from that, dt.tzinfo is *self*, and *dt*' s date and time data are to be viewed as expressing a UTC time. The purpose of *fromutc()* is to adjust the date and time data, returning an equivalent datetime in *self*' s local time.

Most *tzinfo* subclasses should be able to inherit the default *fromutc()* implementation without problems. It' s strong enough to handle fixed-offset time zones, and time zones accounting for both standard and daylight time, and the latter even if the DST transition times differ in different years. An example of a time zone the default *fromutc()* implementation may not handle correctly in all cases is one where the standard offset (from UTC) depends on the specific date and time passed, which can happen for political reasons. The default implementations of *astimezone()* and *fromutc()* may not produce the result you want if the result is one of the hours straddling the moment the standard offset changes.

에러가 발생하는 경우를 위한 코드를 생략하면, 기본 *fromutc()* 구현은 다음과 같이 동작합니다:

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta   # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

다음 tzinfo_examples.py 파일에는 *tzinfo* 클래스의 몇 가지 예가 나와 있습니다:

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)

# A class capturing the platform's idea of local time.
# (May result in wrong values on historical times in
#  timezones where UTC offset and/or the DST rules had
#  changed in the past.)
import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) // SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # Detect fold
        fold = (args == _time.localtime(stamp - dst_diff))
        return datetime(*args, microsecond=dt.microsecond,
                        tzinfo=self, fold=fold)

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, 0)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()


# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt


# US DST Rules
#
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# https://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 2)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 2)
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
# Sunday in April (the one on or after April 24) and to end at 2am (DST time)
# on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006

def us_dst_range(year):
    # Find start and end times for US DST. For years before 1967, return
    # start = end for no DST.
    if 2006 < year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return (datetime(year, 1, 1), ) * 2

    start = first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end


class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them.  The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        dt = dt.replace(tzinfo=None)
        if start + HOUR <= dt < end - HOUR:
            # DST is in effect.
            return HOUR
        if end - HOUR <= dt < end:
            # Fold (an ambiguous hour): use dt.fold to disambiguate.
            return ZERO if dt.fold else HOUR
        if start <= dt < start + HOUR:
            # Gap (a non-existent hour): reverse the fold rule.
            return HOUR if dt.fold else ZERO
        # DST is off.
        return ZERO

    def fromutc(self, dt):
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        start = start.replace(tzinfo=self)
        end = end.replace(tzinfo=self)
        std_time = dt + self.stdoffset
        dst_time = std_time + HOUR
        if end <= dst_time < end + HOUR:
            # Repeated hour
            return std_time.replace(fold=1)
        if std_time < start or dst_time >= end:
            # Standard time
            return std_time
        if start <= std_time < end - HOUR:
            # Daylight saving time
            return dst_time


Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

DST 전환점에서 표준 시간과 일광 절약 시간을 모두 고려하는 *tzinfo* 서브 클래스에는 일 년에 두 번 불가피한 미묘함이 있음에 유의하십시오. 구체적으로, 3월 두 번째 일요일의 1:59 (EST) 다음 분에 시작하고, 11월 첫 번째 일요일 1:59 (EDT) 다음 분에 끝나는 미국 Eastern(UTC -0500)을 고려하십시오:

```
  UTC    3:MM   4:MM   5:MM   6:MM   7:MM   8:MM
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  EST  22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
  EDT  23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start  22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

  end  23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

DST가 시작할 때 ("start" 줄), 지역 벽시계는 1:59에서 3:00로 도약합니다. 그날에는 2:MM 형식의 벽 시간은 실질적인 의미가 없으므로, `astimezone(Eastern)` 은 DST가 시작하는 날에 `hour == 2` 인 결과를 전달하지 않습니다. 예를 들어, 2016년 봄의 전진 전환(forward transition)에서, 다음과 같은 결과를 얻습니다:

```
>>> from datetime import datetime, timezone
>>> from tzinfo_examples import HOUR, Eastern
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

When DST ends (the "end" line), there' s a potentially worse problem: there' s an hour that can' t be spelled unambiguously in local wall time: the last hour of daylight time. In Eastern, that' s times of the form 5:MM UTC on the day daylight time ends. The local wall clock leaps from 1:59 (daylight time) back to 1:00 (standard time) again. Local times of the form 1:MM are ambiguous. *astimezone()* mimics the local clock' s behavior by mapping two adjacent UTC hours into the same local hour then. In the Eastern example, UTC times of the form 5:MM and 6:MM both map to 1:MM when converted to Eastern, but earlier times have the *fold* attribute set to 0 and the later times have it set to 1. For example, at the Fall back transition of 2016, we get:

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

Note that the *datetime* instances that differ only by the value of the *fold* attribute are considered equal in comparisons.

Applications that can' t bear wall-time ambiguities should explicitly check the value of the *fold* attribute or avoid using hybrid *tzinfo* subclasses; there are no ambiguities when using *timezone*, or any other fixed-offset *tzinfo* subclass (such as a class representing only EST (fixed offset -5 hours), or only EDT (fixed offset -4 hours)).

더 보기

zoneinfo

The `datetime` module has a basic *timezone* class (for handling arbitrary fixed offsets from UTC) and its *timezone.utc* attribute (a UTC `timezone` instance).

`zoneinfo` brings the *IANA time zone database* (also known as the Olson database) to Python,

and its usage is recommended.

IANA time zone database
시간대 데이터베이스 (종종 tz, tzdata 또는 zoneinfo라고 합니다)에는 전 세계 여러 지역에서 지역 시간의 히스토리를 표현하는 코드와 데이터가 포함되어 있습니다. 정치 단체가 변경 한 시간대 경계, UTC 오프셋 및 일광 절약 시간 규칙을 반영하기 위해 주기적으로 갱신됩니다.

8.1.9 `timezone` 객체

The `timezone` class is a subclass of `tzinfo`, each instance of which represents a time zone defined by a fixed offset from UTC.

Objects of this class cannot be used to represent time zone information in the locations where different offsets are used in different days of the year or where historical changes have been made to civil time.

class datetime.**timezone**(*offset*, *name=None*)

offset 인자는 지역 시간과 UTC 간의 차이를 나타내는 `timedelta` 객체로 지정해야 합니다. 엄격히 (경계를 포함하지 않는) `-timedelta(hours=24)` 와 `timedelta(hours=24)` 사이여야 합니다. 그렇지 않으면 `ValueError`가 발생합니다.

name 인자는 선택적입니다. 지정되면 `datetime.tzname()` 메서드가 반환하는 값으로 사용될 문자열이어야 합니다.

Added in version 3.2.

버전 3.7에서 변경: UTC 오프셋은 분 단위로 제한되지 않습니다.

timezone.**utcoffset**(*dt*)

`timezone` 인스턴스가 구축될 때 지정된 고정값을 반환합니다.

dt 인자는 무시됩니다. 반환 값은 지역 시간과 UTC 간의 차이와 같은 `timedelta` 인스턴스입니다.

버전 3.7에서 변경: UTC 오프셋은 분 단위로 제한되지 않습니다.

timezone.**tzname**(*dt*)

`timezone` 인스턴스가 구축될 때 지정된 고정값을 반환합니다.

*name*을 생성자에 제공하지 않았으면, `tzname(dt)` 에 의해 반환되는 이름은 다음과 같이 `offset` 값으로부터 생성됩니다. *offset*이 `timedelta(0)` 이면, 이름은 "UTC" 이고, 그렇지 않으면 문자열 `UTC±HH:MM`입니다. 여기서 ±는 `offset`의 부호이고, HH와 MM은 각각 `offset.hours`와 `offset.minutes`의 두 자리 숫자입니다.

버전 3.6에서 변경: Name generated from `offset=timedelta(0)` is now plain `'UTC'`, not `'UTC+00:00'`.

timezone.**dst**(*dt*)

항상 `None`을 반환합니다.

timezone.**fromutc**(*dt*)

`dt + offset`을 반환합니다. *dt* 인자는 `tzinfo`가 `self`로 설정된 어웨어 `datetime` 인스턴스여야 합니다.

클래스 어트리뷰트:

timezone.**utc**

The UTC time zone, `timezone(timedelta(0))`.

8.1.10 `strftime()` and `strptime()` Behavior

`date`, `datetime` 및 `time` 객체는 모두 `strftime(format)` 메서드를 지원하여, 명시적 포맷 문자열로 제어된 시간을 나타내는 문자열을 만듭니다.

Conversely, the `date.strptime()`, `datetime.strptime()` and `time.strptime()` class methods create an object from a string representing the time and a corresponding format string.

The table below provides a high-level comparison of `strftime()` versus `strptime()`:

	strftime	strptime
용도	주어진 포맷에 따라 객체를 문자열로 변환합니다	Parse a string into an object given a corresponding format
메서드의 형	인스턴스 메서드	클래스 메서드
서명	strftime(format)	strptime(date_string, format)

strftime() and strptime() Format Codes

These methods accept format codes that can be used to parse and format dates:

```
>>> datetime.strptime('31/01/22 23:59:59.999999',
...                   '%d/%m/%y %H:%M:%S.%f')
datetime.datetime(2022, 1, 31, 23, 59, 59, 999999)
>>> _.strftime('%a %d %b %Y, %I:%M%p')
'Mon 31 Jan 2022, 11:59PM'
```

다음은 1989 C 표준이 요구하는 모든 포맷 코드 목록이며, 표준 C 구현이 있는 모든 플랫폼에서 작동합니다.

지시자	의미	예	노트
%a	요일을 로케일의 축약된 이름으로.	Sun, Mon, …, Sat (en_US); So, Mo, …, Sa (de_DE)	(1)
%A	요일을 로케일의 전체 이름으로.	Sunday, Monday, …, Saturday (en_US); Sonntag, Montag, …, Samstag (de_DE)	(1)
%w	요일을 10진수로, 0은 일요일이고 6은 토요일입니다.	0, 1, …, 6	
%d	월중 일(day of the month)을 0으로 채워진 10진수로.	01, 02, …, 31	(9)
%b	월을 로케일의 축약된 이름으로.	Jan, Feb, …, Dec (en_US); Jan, Feb, …, Dez (de_DE)	(1)
%B	월을 로케일의 전체 이름으로.	January, February, …, December (en_US); Januar, Februar, …, Dezember (de_DE)	(1)
%m	월을 0으로 채워진 10진수로.	01, 02, …, 12	(9)
%y	세기가 없는 해(year)를 0으로 채워진 10진수로.	00, 01, …, 99	(9)
%Y	세기가 있는 해(year)를 10진수로.	0001, 0002, …, 2013, 2014, …, 9998, 9999	(2)
%H	시(24시간제)를 0으로 채워진 십진수로.	00, 01, …, 23	(9)
%I	시(12시간제)를 0으로 채워진 십진수로.	01, 02, …, 12	(9)
%p	로케일의 오전이나 오후에 해당하는 것.	AM, PM (en_US); am, pm (de_DE)	(1), (3)
%M	분을 0으로 채워진 십진수로.	00, 01, …, 59	(9)
%S	초를 0으로 채워진 10진수로.	00, 01, …, 59	(4), (9)
%f	Microsecond as a decimal number, zero-padded to 6 digits.	000000, 000001, …, 999999	(5)
%z	±HHMM[SS[.ffffff]] 형태의 UTC 오프셋(객체가 나이브하면 빈 문자열).	(비어 있음), +0000, -0400, +1030, +063415, -030712.345216	(6)
%Z	타임존 이름(객체가 나이브하면 빈 문자열).	(비어 있음), UTC, GMT	(6)
%j	연중 일(day of the year)을 0으로 채워진 십진수로.	001, 002, …, 366	(9)

C89 표준에서 요구하지 않는 몇 가지 추가 지시자가 편의상 포함되어 있습니다. 이 파라미터들은 모두 ISO 8601 날짜 값에 해당합니다.

지시자	의미	예	노트
`%G`	ISO 주(`%V`)의 더 큰 부분을 포함하는 연도를 나타내는 세기가 있는 ISO 8601 연도.	0001, 0002, …, 2013, 2014, …, 9998, 9999	(8)
`%u`	ISO 8601 요일을 10진수로, 1은 월요일입니다.	1, 2, …, 7	
`%V`	ISO 8601 주를 월요일을 주의 시작으로 하는 십진수로. 주 01은 1월 4일을 포함하는 주입니다.	01, 02, …, 53	(8), (9)
`%:z`	UTC offset in the form `±HH:MM[:SS[.ffffff]]` (empty string if the object is naive).	(empty), +00:00, -04:00, +10:30, +06:34:15, -03:07:12.345216	(6)

These may not be available on all platforms when used with the *strftime()* method. The ISO 8601 year and ISO 8601 week directives are not interchangeable with the year and week number directives above. Calling *strptime()* with incomplete or ambiguous ISO 8601 directives will raise a *ValueError*.

The full set of format codes supported varies across platforms, because Python calls the platform C library' s `strftime()` function, and platform variations are common. To see the full set of format codes supported on your platform, consult the *strftime(3)* documentation. There are also differences between platforms in handling of unsupported format specifiers.

Added in version 3.6: `%G`, `%u` 및 `%V`가 추가되었습니다.

Added in version 3.12: `%:z` was added.

기술적 세부 사항

Broadly speaking, `d.strftime(fmt)` acts like the *time* module' s `time.strftime(fmt, d.timetuple())` although not all objects support a *timetuple()* method.

For the *datetime.strptime()* class method, the default value is `1900-01-01T00:00:00.000`: any components not specified in the format string will be pulled from the default value.[4]

`datetime.strptime(date_string, format)`은 다음과 동등합니다:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

except when the format includes sub-second components or time zone offset information, which are supported in `datetime.strptime` but are discarded by `time.strptime`.

For *time* objects, the format codes for year, month, and day should not be used, as `time` objects have no such values. If they' re used anyway, 1900 is substituted for the year, and 1 for the month and day.

For *date* objects, the format codes for hours, minutes, seconds, and microseconds should not be used, as *date* objects have no such values. If they' re used anyway, 0 is substituted for them.

같은 이유로, 현재 로케일의 문자 집합으로는 표현할 수 없는 유니코드 코드 포인트를 포함하는 포맷 문자열의 처리도 플랫폼에 따라 다릅니다. 일부 플랫폼에서는 이러한 코드 포인트가 그대로 출력에 보존되지만, 다른 곳에서는 `strftime`이 *UnicodeError*를 발생시키거나 대신 빈 문자열을 반환할 수 있습니다.

노트:

(1) Because the format depends on the current locale, care should be taken when making assumptions about the output value. Field orderings will vary (for example, "month/day/year" versus "day/month/year"), and the output may contain non-ASCII characters.

(2) The *strptime()* method can parse years in the full [1, 9999] range, but years < 1000 must be zero-filled to 4-digit width.

버전 3.2에서 변경: In previous versions, *strftime()* method was restricted to years >= 1900.

[4] Passing `datetime.strptime('Feb 29', '%b %d')` will fail since 1900 is not a leap year.

버전 3.3에서 변경: In version 3.2, *strftime()* method was restricted to years >= 1000.

(3) When used with the *strptime()* method, the `%p` directive only affects the output hour field if the `%I` directive is used to parse the hour.

(4) Unlike the *time* module, the `datetime` module does not support leap seconds.

(5) When used with the *strptime()* method, the `%f` directive accepts from one to six digits and zero pads on the right. `%f` is an extension to the set of format characters in the C standard (but implemented separately in datetime objects, and therefore always available).

(6) For a naive object, the `%z`, `%:z` and `%Z` format codes are replaced by empty strings.

어웨어 객체의 경우:

%z

utcoffset() is transformed into a string of the form `±HHMM[SS[.ffffff]]`, where `HH` is a 2-digit string giving the number of UTC offset hours, `MM` is a 2-digit string giving the number of UTC offset minutes, `SS` is a 2-digit string giving the number of UTC offset seconds and `ffffff` is a 6-digit string giving the number of UTC offset microseconds. The `ffffff` part is omitted when the offset is a whole number of seconds and both the `ffffff` and the `SS` part is omitted when the offset is a whole number of minutes. For example, if *utcoffset()* returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string `'-0330'`.

버전 3.7에서 변경: UTC 오프셋은 분 단위로 제한되지 않습니다.

버전 3.7에서 변경: When the `%z` directive is provided to the *strptime()* method, the UTC offsets can have a colon as a separator between hours, minutes and seconds. For example, `'+01:00:00'` will be parsed as an offset of one hour. In addition, providing `'Z'` is identical to `'+00:00'`.

%:z

Behaves exactly as `%z`, but has a colon separator added between hours, minutes and seconds.

%Z

In *strftime()*, `%Z` is replaced by an empty string if *tzname()* returns `None`; otherwise `%Z` is replaced by the returned value, which must be a string.

strptime() only accepts certain values for `%Z`:

1. 컴퓨터의 로케일에 대한 `time.tzname`의 모든 값
2. 하드 코딩된 값 `UTC`와 `GMT`

따라서 일본에 거주하는 사람은 `JST`, `UTC` 및 `GMT`를 유효한 값으로 가질 수 있지만, 아마도 `EST`는 아닙니다. 유효하지 않은 값에 대해서는 `ValueError`가 발생합니다.

버전 3.2에서 변경: When the `%z` directive is provided to the *strptime()* method, an aware *datetime* object will be produced. The `tzinfo` of the result will be set to a *timezone* instance.

(7) When used with the *strptime()* method, `%U` and `%W` are only used in calculations when the day of the week and the calendar year (`%Y`) are specified.

(8) Similar to `%U` and `%W`, `%V` is only used in calculations when the day of the week and the ISO year (`%G`) are specified in a *strptime()* format string. Also note that `%G` and `%Y` are not interchangeable.

(9) When used with the *strptime()* method, the leading zero is optional for formats `%d`, `%m`, `%H`, `%I`, `%M`, `%S`, `%j`, `%U`, `%W`, and `%V`. Format `%y` does require a leading zero.

(10) When parsing a month and day using *strptime()*, always include a year in the format. If the value you need to parse lacks a year, append an explicit dummy leap year. Otherwise your code will raise an exception when it encounters leap day because the default year used by the parser is not a leap year. Users run into this bug every four years…

```
>>> month_day = "02/29"
>>> datetime.strptime(f"{month_day};1984", "%m/%d;%Y")  # No leap year bug.
datetime.datetime(1984, 2, 29, 0, 0)
```

Deprecated since version 3.13, will be removed in version 3.15: `strptime()` calls using a format string containing a day of month without a year now emit a `DeprecationWarning`. In 3.15 or later we may change this into an error or change the default year to a leap year. See gh-70647.

8.2 zoneinfo — IANA 시간대 지원

Added in version 3.9.

소스 코드: Lib/zoneinfo

`zoneinfo` 모듈은 원래 **PEP 615**에서 지정된 대로 IANA 시간대 데이터베이스를 지원하기 위한 구상 시간대 구현을 제공합니다. 기본적으로, `zoneinfo`는 가능하면 시스템의 시간대 데이터를 사용합니다; 사용 가능한 시스템 시간대 데이터가 없으면, 라이브러리는 PyPI에 있는 자사(first-party) tzdata 패키지를 사용하는 것으로 대체합니다.

더 보기

모듈: *datetime*
`ZoneInfo` 클래스가 사용되도록 설계된 `time`과 `datetime` 형을 제공합니다.

패키지 tzdata
PyPI를 통해 시간대 데이터를 제공하기 위해 CPython 핵심 개발자가 유지 보수하는 자사(first-party) 패키지.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

8.2.1 ZoneInfo 사용하기

`ZoneInfo`는 `datetime.tzinfo` 추상 베이스 클래스의 구상 구현이며, 생성자, `datetime.replace` 메서드 또는 `datetime.astimezone`을 통해 `tzinfo`에 연결하려는 것입니다:

```
>>> from zoneinfo import ZoneInfo
>>> from datetime import datetime, timedelta

>>> dt = datetime(2020, 10, 31, 12, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-10-31 12:00:00-07:00

>>> dt.tzname()
'PDT'
```

이 방식으로 구성된 datetime들은 datetime 산술과 호환되며 추가 개입 없이 일광 절약 시간제 전환을 처리합니다:

```
>>> dt_add = dt + timedelta(days=1)

>>> print(dt_add)
2020-11-01 12:00:00-08:00

>>> dt_add.tzname()
'PST'
```

이 시간대는 **PEP 495**에 도입된 `fold` 어트리뷰트도 지원합니다. 모호한 시간을 유발하는 오프셋 전환(가령 일광 절약 시간에서 표준 시간으로의 전환) 중, 전환 전의 오프셋이 `fold=0`일 때 사용되며, 전환 후의 오프셋은 `fold=1`일 때 사용됩니다, 예를 들어:

```
>>> dt = datetime(2020, 11, 1, 1, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-11-01 01:00:00-07:00

>>> print(dt.replace(fold=1))
2020-11-01 01:00:00-08:00
```

다른 시간대에서 변환할 때, fold는 올바른 값으로 설정됩니다:

```
>>> from datetime import timezone
>>> LOS_ANGELES = ZoneInfo("America/Los_Angeles")
>>> dt_utc = datetime(2020, 11, 1, 8, tzinfo=timezone.utc)

>>> # PDT -> PST 전환 전
>>> print(dt_utc.astimezone(LOS_ANGELES))
2020-11-01 01:00:00-07:00

>>> # PDT -> PST 전환 후
>>> print((dt_utc + timedelta(hours=1)).astimezone(LOS_ANGELES))
2020-11-01 01:00:00-08:00
```

8.2.2 데이터 소스

zoneinfo 모듈은 시간대 데이터를 직접 제공하지 않고, 시스템 시간대 데이터베이스나 자사(first-party) PyPI 패키지 tzdata(있다면)에서 시간대 정보를 가져옵니다. 특히 윈도우 시스템을 포함한 일부 시스템에는 사용 가능한 IANA 데이터베이스가 없어서, 시간대 데이터가 필요한 플랫폼 간 호환성을 목표로 하는 프로젝트는 tzdata에 대한 종속성을 선언하는 것이 좋습니다. 시스템 데이터나 tzdata를 모두 사용할 수 없으면, *ZoneInfo*에 대한 모든 호출은 *ZoneInfoNotFoundError*를 발생시킵니다.

데이터 소스 구성

ZoneInfo(key) 가 호출될 때, 생성자는 먼저 *TZPATH*에 지정된 디렉터리에서 key와 일치하는 파일을 검색하고, 실패하면 tzdata 패키지에서 일치하는 것을 찾습니다. 이 동작은 세 가지 방법으로 구성할 수 있습니다:

1. 달리 지정되지 않을 때의 기본 *TZPATH*는 컴파일 시간에 구성할 수 있습니다.
2. *TZPATH*는 환경 변수를 사용하여 구성할 수 있습니다.
3. 실행 시간에는, *reset_tzpath()* 함수를 사용하여 검색 경로를 조작할 수 있습니다.

컴파일 시간 구성

기본 *TZPATH*에는 시간대 데이터베이스에 대한 몇 가지 일반적인 배포 위치가 포함되어 있습니다 (윈도우는 예외인데, 시간대 데이터에 대한 "잘 알려진" 위치가 없습니다). POSIX 시스템에서, 시스템 시간대 데이터가 배치된 위치를 알고 있는 다운 스트림 배포자와 소스에서 파이썬을 빌드하는 사람은 컴파일 시간 옵션 TZPATH(또는, 더 가능성 있는, configure flag --with-tzpath)를 지정하여 기본 시간대 경로를 변경할 수 있습니다. *os.pathsep*으로 구분된 문자열이어야 합니다.

모든 플랫폼에서, 구성된 값은 *sysconfig.get_config_var()*에서 TZPATH 키로 사용 가능합니다.

환경 구성

*TZPATH*를 초기화할 때 (임포트 시점이나 인자 없이 *reset_tzpath()*가 호출될 때마다) zoneinfo 모듈은 환경 변수 PYTHONTZPATH(있다면)를 사용하여 검색 경로를 설정합니다.

PYTHONTZPATH

사용할 시간대 검색 경로를 포함하는 *os.pathsep*으로 구분된 문자열입니다. 상대 경로가 아닌 절대 경로로만 구성되어야 합니다. PYTHONTZPATH에 지정된 상대 컴포넌트는 사용되지 않지만, 그 밖의 상대 경로가 지정된 경우의 동작은 구현이 정의합니다; CPython은 *InvalidTZPathWarning*을 발생시키지만, 다른 구현에서는 잘못된 구성 요소를 조용히 무시하거나 예외를 발생시킬 수 있습니다.

시스템이 시스템 데이터를 무시하고 tzdata 패키지를 대신 사용하도록 설정하려면, `PYTHONTZPATH=""`를 설정하십시오.

실행 시간 구성

reset_tzpath() 함수를 사용하여 실행 시간에 TZ 검색 경로를 구성할 수도 있습니다. 일반적으로 권장되는 작업은 아닙니다만, 특정 시간대 경로를 사용해야 하는 (또는 시스템 시간대에 대한 액세스를 비활성화해야 하는) 테스트 함수에 사용하는 것은 합리적입니다.

8.2.3 ZoneInfo 클래스

class zoneinfo.**ZoneInfo**(*key*)

문자열 `key`로 지정된 IANA 시간대를 나타내는 구상 *datetime.tzinfo* 서브 클래스. 기본 생성자에 대한 호출은 항상 동일하다고 비교되는 객체를 반환합니다; 다르게 말하면, *ZoneInfo.clear_cache()*를 통한 캐시 무효화를 방지한다면, 다음 어서션이 항상 참입니다:

```
a = ZoneInfo(key)
b = ZoneInfo(key)
assert a is b
```

`key`는 상위 수준 참조가 없는 상대적인 정규화된 POSIX 경로의 형식이어야 합니다. 적합하지 않은 key가 전달되면 생성자가 *ValueError*를 발생시킵니다.

`key`와 일치하는 파일이 없으면, 생성자는 *ZoneInfoNotFoundError*를 발생시킵니다.

`ZoneInfo` 클래스에는 두 개의 대체 생성자가 있습니다:

classmethod ZoneInfo.**from_file**(*file_obj*, /, *key=None*)

바이트열을 반환하는 파일류 객체(예를 들어 바이너리 모드로 열린 파일이나 *io.BytesIO* 객체)에서 `ZoneInfo` 객체를 생성합니다. 기본 생성자와 달리, 항상 새로운 객체를 생성합니다.

`key` 매개 변수는 `__str__()`과 `__repr__()`를 위해 시간대 이름을 설정합니다.

이 생성자를 통해 만들어진 객체는 피클 할 수 없습니다 (피클 직렬화를 참조하십시오).

classmethod ZoneInfo.**no_cache**(*key*)

생성자의 캐시를 우회하는 대체 생성자. 기본 생성자와 동일하지만, 호출마다 새 객체를 반환합니다. 이것은 테스트나 데모 목적으로 유용 할 수 있지만, 다른 캐시 무효화 전략을 갖는 시스템을 만드는데 사용될 수도 있습니다.

이 생성자를 통해 만들어진 객체는 역 피클 될 때 역 직렬화 프로세스의 캐시도 우회합니다.

> **조심**
>
> 이 생성자를 사용하면 예기치 못한 방식으로 날짜 시간의 의미가 변경될 수 있기 때문에, 필요한 경우에만 사용하십시오.

다음과 같은 클래스 메서드도 사용할 수 있습니다:

classmethod ZoneInfo.**clear_cache**(*, *only_keys=None*)

`ZoneInfo` 클래스에서 캐시를 무효로 하는 메서드. 인자가 전달되지 않으면, 모든 캐시가 무효가 되고 각 키의 기본 생성자에 대한 다음 호출은 새 인스턴스를 반환합니다.

키 이름의 이터러블이 `only_keys` 매개 변수에 전달되면, 지정된 키만 캐시에서 제거됩니다. `only_keys`에 전달되었지만, 캐시에서 찾을 수 없는 키는 무시됩니다.

> **경고**

> 이 함수를 호출하면 예기치 못한 방식으로 ZoneInfo를 사용하는 날짜 시간의 의미를 변경할 수 있습니다; 이는 모듈 상태를 수정하므로 광범위한 영향을 미칠 수 있습니다. 필요한 경우에만 사용하십시오.

이 클래스에는 하나의 어트리뷰트가 있습니다:

ZoneInfo.**key**

이것은 읽기 전용 어트리뷰트이며 생성자에 전달된 key의 값을 반환하는데, IANA 시간대 데이터베이스의 조회 키이어야 합니다 (예를 들어 America/New_York, Europe/Paris 또는 Asia/Tokyo).

key 매개 변수를 지정하지 않고 파일에서 생성된 시간대의 경우, None으로 설정됩니다.

> **참고**
>
> 이들을 최종 사용자에게 노출하는 것이 다소 일반적인 관행이지만, 이 값들은 관련 시간대를 나타내는 기본 키로 설계되었을 뿐, 사용자 대면 요소일 필요는 없습니다. CLDR (the Unicode Common Locale Data Repository)과 같은 프로젝트를 사용하면 이러한 키에서 더 사용자 친화적인 문자열을 얻을 수 있습니다.

문자열 표현

ZoneInfo 객체에 대해 *str*을 호출할 때 반환되는 문자열 표현은 기본적으로 *ZoneInfo.key* 어트리뷰트를 사용합니다 (어트리뷰트 설명서의 사용법에 관한 참고를 보십시오):

```
>>> zone = ZoneInfo("Pacific/Kwajalein")
>>> str(zone)
'Pacific/Kwajalein'

>>> dt = datetime(2020, 4, 1, 3, 15, tzinfo=zone)
>>> f"{dt.isoformat()} [{dt.tzinfo}]"
'2020-04-01T03:15:00+12:00 [Pacific/Kwajalein]'
```

key 매개 변수를 지정하지 않고 파일에서 생성된 객체의 경우, str은 *repr()* 호출로 대체됩니다. ZoneInfo의 repr은 구현 정의이며 버전 간에 반드시 안정적일 필요는 없지만, 유효한 ZoneInfo 키가 될 수는 없음이 보장됩니다.

피클 직렬화

모든 전환 데이터를 직렬화하는 대신, ZoneInfo 객체는 키로 직렬화되며 파일에서 생성된 ZoneInfo 객체는 (지정된 key 값을 가진 객체조차) 피클 할 수 없습니다.

ZoneInfo 파일의 동작은 파일 생성 방식에 따라 다릅니다:

1. ZoneInfo(key): 기본 생성자로 생성될 때, ZoneInfo 객체는 키로 직렬화되며, 역 직렬화할 때, 역 직렬화 프로세스는 기본 생성자를 사용해서 같은 시간대에 대한 다른 참조와 같은 객체가 될 것으로 예상됩니다. 예를 들어, europe_berlin_pkl이 ZoneInfo("Europe/Berlin")에서 생성된 피클을 포함하는 문자열이면, 다음과 같은 동작이 예상됩니다:

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl)
   >>> a is b
   True
   ```

2. ZoneInfo.no_cache(key): 캐시 우회 생성자에서 생성될 때, 이 ZoneInfo 객체도 키로 직렬화되지만, 역 직렬화할 때, 역 직렬화 프로세스는 캐시 우회 생성자를 사용합니다. europe_berlin_pkl_nc가 ZoneInfo.no_cache("Europe/Berlin")에서 생성된 피클을 포함하는 문자열이면, 다음과 같은 동작이 예상됩니다:

```
>>> a = ZoneInfo("Europe/Berlin")
>>> b = pickle.loads(europe_berlin_pkl_nc)
>>> a is b
False
```

3. `ZoneInfo.from_file(file_obj, /, key=None)`: When constructed from a file, the `ZoneInfo` object raises an exception on pickling. If an end user wants to pickle a `ZoneInfo` constructed from a file, it is recommended that they use a wrapper type or a custom serialization function: either serializing by key or storing the contents of the file object and serializing that.

이 직렬화 방법에서는 직렬화와 역 직렬화 환경 모두에서 클래스와 함수에 대한 참조가 존재해야 하는 방식과 유사하게, 직렬화와 역 직렬화 측 모두에서 필요한 키의 시간대 데이터를 사용할 수 있어야 합니다. 또한 다른 버전의 시간대 데이터가 있는 환경에서 피클링 된 `ZoneInfo`를 역 피클링 할 때 결과의 일관성에 대해 보장되지 않습니다.

8.2.4 함수

`zoneinfo.`**`available_timezones`**`()`

시간대 경로의 어느 곳에서건 사용 가능한 IANA 시간대에 유효한 모든 키가 포함된 집합을 가져옵니다. 이것은 함수를 호출할 때마다 다시 계산됩니다.

이 함수는 규범적(canonical) 시간대 이름 만 포함하고 `posix/`와 `right/` 디렉터리에 있는 것이나 `posixrules` 시간대와 같은 "특수" 시간대는 포함하지 않습니다.

> **조심**
>
> 시간대 경로에 있는 파일이 유효한 시간대인지 확인하는 가장 좋은 방법은 시작 부분에 나오는 "매직 문자열"을 읽는 것이라서, 이 함수는 많은 파일을 열 수 있습니다.

> **참고**
>
> 이 값은 최종 사용자에게 노출되도록 설계되지 않았습니다; 사용자 대면 요소의 경우, 응용 프로그램은 CLDR (the Unicode Common Locale Data Repository)과 같은 것을 사용하여 더 사용자 친화적인 문자열을 가져와야 합니다. *`ZoneInfo.key`*에 있는 주의 사항도 참조하십시오.

`zoneinfo.`**`reset_tzpath`**`(`*to=None*`)`

모듈의 시간대 검색 경로(*`TZPATH`*)를 설정하거나 재설정합니다. 인자 없이 호출하면, *`TZPATH`*가 기본 값으로 설정됩니다.

`reset_tzpath`를 호출해도 *`ZoneInfo`* 캐시가 무효가 되지 않아서, 기본 `ZoneInfo` 생성자에 대한 호출은 캐시 누락의 경우에만 새 `TZPATH`를 사용합니다.

`to` 매개 변수는 문자열이나 *`os.PathLike`*의 시퀀스이어야 하며 문자열이 아닙니다. 모두 절대 경로여야 합니다. 절대 경로 이외의 것이 전달되면 *`ValueError`*가 발생합니다.

8.2.5 전역

`zoneinfo.`**`TZPATH`**

시간대 검색 경로를 나타내는 읽기 전용 시퀀스 – 키에서 `ZoneInfo`를 생성할 때, 키는 `TZPATH`의 각 항목에 결합하며, 발견된 첫 번째 파일이 사용됩니다.

`TZPATH`는 구성 방법과 관계없이 절대 경로만 포함할 수 있고, 상대 경로는 절대 포함하지 않습니다.

`zoneinfo.TZPATH`가 가리키는 객체는 *`reset_tzpath()`*에 대한 호출에 따라 변경될 수 있어서, `zoneinfo`에서 `TZPATH`를 임포트 하거나 수명이 긴 변수에 `zoneinfo.TZPATH`를 대입하는 대신 `zoneinfo.TZPATH`를 사용하는 것이 좋습니다.

시간대 검색 경로 구성에 대한 자세한 정보는 데이터 소스 구성을 참조하십시오.

8.2.6 예외와 경고

exception zoneinfo.**ZoneInfoNotFoundError**

지정된 키를 시스템에서 찾을 수 없어서 *ZoneInfo* 객체 생성에 실패할 때 발생합니다. 이것은 *KeyError*의 서브 클래스입니다.

exception zoneinfo.**InvalidTZPathWarning**

*PYTHONTZPATH*에 상대 경로와 같이 필터링 될 유효하지 않은 구성 요소가 포함되어있을 때 발생합니다.

8.3 calendar — 일반 달력 관련 함수

소스 코드: Lib/calendar.py

이 모듈을 사용하면 유닉스 **cal** 프로그램과 같은 달력을 출력할 수 있으며, 달력과 관련된 유용한 추가 함수를 제공합니다. 기본적으로, 이 달력은 월요일을 주의 첫째 날로 하고, 일요일을 마지막 날로 합니다 (유럽 관례). 주의 첫째 날을 일요일(6)이나 다른 요일로 설정하려면 *setfirstweekday()*를 사용하십시오. 날짜를 지정하는 매개 변수는 정수로 제공됩니다. 관련 기능에 대해서는, *datetime*과 *time* 모듈도 참조하십시오.

이 모듈에 정의된 함수와 클래스는 이상적인 달력을 사용합니다, 양방향으로 무한정 확장된 현재 그레고리력. 이것은 Dershowitz와 Reingold의 저서 "Calendrical Calculations" 에 나오는 "역산 그레고리(proleptic Gregorian)" 달력의 정의와 일치하며, 모든 계산의 기본 달력입니다. 0과 음의 연도는 ISO 8601 표준에 규정된 대로 해석됩니다. 0년은 BC 1년, -1년은 BC 2년 등입니다.

class calendar.**Calendar**(*firstweekday=0*)

Calendar 객체를 만듭니다. *firstweekday*는 주의 첫 번째 날을 지정하는 정수입니다. *MONDAY*는 0(기본값)이고, *SUNDAY*는 6입니다.

Calendar 객체는 포매팅을 위해 달력 데이터를 준비하는 데 사용할 수 있는 몇 가지 메서드를 제공합니다. 이 클래스는 스스로 포매팅을 수행하지 않습니다. 이는 서브 클래스의 역할입니다.

Calendar 인스턴스에는 다음과 같은 메서드와 어트리뷰트가 있습니다:

firstweekday

The first weekday as an integer (0–6).

This property can also be set and read using *setfirstweekday()* and *getfirstweekday()* respectively.

getfirstweekday()

Return an *int* for the current first weekday (0–6).

Identical to reading the *firstweekday* property.

setfirstweekday(*firstweekday*)

Set the first weekday to *firstweekday*, passed as an *int* (0–6)

Identical to setting the *firstweekday* property.

iterweekdays()

한 주 동안 사용될 요일 번호의 이터레이터를 반환합니다. 이터레이터의 첫 번째 값은 *firstweekday* 프로퍼티 값과 같습니다.

itermonthdates(*year*, *month*)

year 연도의 *month* 월 (1–12) 동안의 이터레이터를 반환합니다. 이 이터레이터는 해당 월의 모든 날(*datetime.date* 객체로)과 완전한 주를 얻기 위해 필요한 해당 월의 시작일 전이나 해당 월의 종료일 이후의 모든 날을 반환합니다.

itermonthdays(*year*, *month*)

`itermonthdates()`와 유사하게 *year* 연도의 *month* 월 동안의 이터레이터를 반환하지만, `datetime.date` 범위로 제한되지 않습니다. 반환된 날은 단순히 월 중 날 번호입니다. 지정된 월 바깥에 있는 날의 경우, 날 번호는 `0`입니다.

itermonthdays2(*year*, *month*)

`itermonthdates()`와 유사하게 *year* 연도의 *month* 월 동안의 이터레이터를 반환하지만, `datetime.date` 범위로 제한되지 않습니다. 반환된 날은 월 중 날 번호와 요일 번호로 구성된 튜플입니다.

itermonthdays3(*year*, *month*)

`itermonthdates()`와 유사하게 *year* 연도의 *month* 월 동안의 이터레이터를 반환하지만, `datetime.date` 범위로 제한되지 않습니다. 반환된 날은 연도, 월 및 월 중 날 번호로 구성된 튜플입니다.

Added in version 3.7.

itermonthdays4(*year*, *month*)

`itermonthdates()`와 유사하게 *year* 연도의 *month* 월 동안의 이터레이터를 반환하지만, `datetime.date` 범위로 제한되지 않습니다. 반환된 날은 연도, 월, 월 중 날 및 요일 번호로 구성된 튜플입니다.

Added in version 3.7.

monthdatescalendar(*year*, *month*)

*year*의 *month* 월에 있는 주의 리스트를 전체 주로 반환합니다. 주는 7개의 `datetime.date` 객체 리스트입니다.

monthdays2calendar(*year*, *month*)

*year*의 *month* 월에 있는 주의 리스트를 전체 주로 반환합니다. 주는 날 번호와 요일 번호 튜플 7개의 리스트입니다.

monthdayscalendar(*year*, *month*)

*year*의 *month* 월에 있는 주의 리스트를 전체 주로 반환합니다. 주는 날 번호 7개의 리스트입니다.

yeardatescalendar(*year*, *width=3*)

포매팅 준비된 지정된 연도의 데이터를 반환합니다. 반환 값은 월 행의 리스트입니다. 각 월 행에는 최대 *width* 월(기본값은 3)이 포함됩니다. 각 월은 4–6주를 포함하고, 각 주는 1–7일을 포함합니다. 날은 `datetime.date` 객체입니다.

yeardays2calendar(*year*, *width=3*)

포매팅 준비된 지정된 연도의 데이터를 반환합니다 (`yeardatescalendar()`와 유사합니다). 주 리스트의 항목은 날 번호와 요일 번호의 튜플입니다. 이달 밖의 날 번호는 0입니다.

yeardayscalendar(*year*, *width=3*)

포매팅 준비된 지정된 연도의 데이터를 반환합니다 (`yeardatescalendar()`와 유사합니다). 주 리스트의 항목은 날 번호입니다. 이달 밖의 날 번호는 0입니다.

class calendar.**TextCalendar**(*firstweekday=0*)

이 클래스는 평문 텍스트 달력을 생성하는 데 사용할 수 있습니다.

`TextCalendar` 인스턴스에는 다음과 같은 메서드가 있습니다:

formatday(*theday*, *weekday*, *width*)

Return a string representing a single day formatted with the given *width*. If *theday* is `0`, return a string of spaces of the specified width, representing an empty day. The *weekday* parameter is unused.

formatweek(*theweek*, *w=0*)

한 주를 줄 넘김이 없는 문자열로 반환합니다. *w*가 제공되면, 가운데 정렬되는 날짜 열의 너비를 지정합니다. 생성자에 지정되거나 `setfirstweekday()` 메서드로 설정된 첫 번째 요일에 따라 다릅니다.

formatweekday(*weekday*, *width*)

Return a string representing the name of a single weekday formatted to the specified *width*. The *weekday* parameter is an integer representing the day of the week, where `0` is Monday and `6` is Sunday.

formatweekheader(*width*)

Return a string containing the header row of weekday names, formatted with the given *width* for each column. The names depend on the locale settings and are padded to the specified width.

formatmonth(*theyear*, *themonth*, *w=0*, *l=0*)

월의 달력을 여러 줄 문자열로 반환합니다. *w*가 제공되면, 가운데 정렬되는 날짜 열의 너비를 지정합니다. *l*이 제공되면, 각 주가 사용할 줄 수를 지정합니다. 생성자에 지정되거나 `setfirstweekday()` 메서드로 설정된 첫 번째 요일에 따라 다릅니다.

formatmonthname(*theyear*, *themonth*, *width=0*, *withyear=True*)

Return a string representing the month' s name centered within the specified *width*. If *withyear* is `True`, include the year in the output. The *theyear* and *themonth* parameters specify the year and month for the name to be formatted respectively.

prmonth(*theyear*, *themonth*, *w=0*, *l=0*)

`formatmonth()`에서 반환한 월의 달력을 인쇄합니다.

formatyear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)

전체 연도의 *m*-열 달력을 여러 줄 문자열로 반환합니다. 선택적 매개 변수 *w*, *l* 및 *c*는 각각 날짜 열 너비, 주당 줄 수 및 월 열 사이의 스페이스 수입니다. 생성자에 지정되거나 `setfirstweekday()` 메서드로 설정된 첫 번째 요일에 따라 다릅니다. 달력을 생성할 수 있는 가장 빠른 연도는 플랫폼에 따라 다릅니다.

pryear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)

`formatyear()`에서 반환 연도의 달력을 인쇄합니다.

class calendar.**HTMLCalendar**(*firstweekday=0*)

이 클래스는 HTML 달력을 생성하는 데 사용할 수 있습니다.

`HTMLCalendar` 인스턴스에는 다음과 같은 메서드가 있습니다:

formatmonth(*theyear*, *themonth*, *withyear=True*)

월의 달력을 HTML 테이블로 반환합니다. *withyear*가 참이면 연도가 헤더에 포함되고, 그렇지 않으면 월 이름 만 사용됩니다.

formatyear(*theyear*, *width=3*)

연도의 달력을 HTML 테이블로 반환합니다. *width*(기본값은 3)는 행 당 개월 수를 지정합니다.

formatyearpage(*theyear*, *width=3*, *css='calendar.css'*, *encoding=None*)

연도의 달력을 완전한 HTML 페이지로 반환합니다. *width*(기본값은 3)는 행 당 개월 수를 지정합니다. *css*는 사용할 캐스케이딩 스타일 시트의 이름입니다. 스타일 시트를 사용하지 않으면 `None`을 전달할 수 있습니다. *encoding*은 출력에 사용될 인코딩을 지정합니다 (기본값은 시스템 기본 인코딩입니다).

formatmonthname(*theyear*, *themonth*, *withyear=True*)

월 이름을 HTML 테이블 행으로 반환합니다. *withyear*가 참이면 연도가 행에 포함되고, 그렇지 않으면 월 이름 만 사용됩니다.

`HTMLCalendar`에는 달력에서 사용하는 CSS 클래스를 사용자 정의하기 위해 재정의할 수 있는 다음과 같은 어트리뷰트가 있습니다:

cssclasses

각 요일에 사용되는 CSS 클래스 리스트. 기본 클래스 리스트는 다음과 같습니다:

```
cssclasses = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"]
```

각 날에 더 많은 스타일을 추가할 수 있습니다:

```
cssclasses = ["mon text-bold", "tue", "wed", "thu", "fri", "sat", "sun red
↪"]
```

이 리스트의 길이는 7개의 항목임에 유의하십시오.

cssclass_noday

지난달이나 다음 달에 등장하는 요일의 CSS 클래스.

Added in version 3.7.

cssclasses_weekday_head

헤더 행에 있는 요일 이름에 사용되는 CSS 클래스 리스트. 기본값은 *cssclasses*와 같습니다.

Added in version 3.7.

cssclass_month_head

월 헤드 CSS 클래스 (*formatmonthname()* 에서 사용됩니다). 기본값은 "month" 입니다.

Added in version 3.7.

cssclass_month

월 전체 테이블의 CSS 클래스 (*formatmonth()* 에서 사용됩니다). 기본값은 "month" 입니다.

Added in version 3.7.

cssclass_year

연도 전체 표의 CSS 클래스 (*formatyear()* 에서 사용됩니다). 기본값은 "year" 입니다.

Added in version 3.7.

cssclass_year_head

연도 전체의 테이블 헤드의 CSS 클래스 (*formatyear()* 에서 사용됩니다). 기본값은 "year" 입니다.

Added in version 3.7.

위에서 설명한 클래스 어트리뷰트의 이름은 단수이지만 (예를 들어 cssclass_month cssclass_noday), 단일 CSS 클래스를 스페이스로 구분된 CSS 클래스 목록으로 바꿀 수 있습니다. 예를 들면 다음과 같습니다:

```
"text-bold text-red"
```

다음은 HTMLCalendar를 사용자 정의하는 방법에 대한 예입니다:

```
class CustomHTMLCal(calendar.HTMLCalendar):
    cssclasses = [style + " text-nowrap" for style in
                  calendar.HTMLCalendar.cssclasses]
    cssclass_month_head = "text-center month-head"
    cssclass_month = "text-center month"
    cssclass_year = "text-italic lead"
```

class calendar.**LocaleTextCalendar**(*firstweekday=0, locale=None*)

*TextCalendar*의 이 서브 클래스는 생성자에 로케일 이름을 전달할 수 있으며 지정된 로케일에서 월과 요일 이름을 반환합니다.

class calendar.**LocaleHTMLCalendar**(*firstweekday=0, locale=None*)

*HTMLCalendar*의 이 서브 클래스는 생성자에 로케일 이름을 전달할 수 있으며 지정된 로케일에서 월과 요일 이름을 반환합니다.

> **참고**
>
> 이 두 클래스의 생성자, `formatweekday()` 와 `formatmonthname()` 메서드는 `LC_TIME` 로케일을 주어진 *locale*로 임시 변경합니다. 현재 로케일은 프로세스 전체 설정이므로, 스레드 안전하지 않습니다.

간단한 텍스트 달력을 위해 이 모듈은 다음 함수를 제공합니다.

`calendar.`**`setfirstweekday`**(*weekday*)

매주 시작일을 weekday(`0`은 월요일, `6`은 일요일)로 설정합니다. 편의를 위해 `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` 및 `SUNDAY` 값이 제공됩니다. 예를 들어, 주의 첫 번째 날을 일요일로 설정하려면:

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

`calendar.`**`firstweekday`**()

각 주를 시작하는 요일의 현재 설정을 반환합니다.

`calendar.`**`isleap`**(*year*)

*year*가 윤년이면 `True`를, 그렇지 않으면 `False`를 반환합니다.

`calendar.`**`leapdays`**(*y1*, *y2*)

*y1*에서 *y2*(우측 경계 제외) 범위에서 윤년의 수를 반환합니다, 여기서 *y1*과 *y2*는 연도입니다.

이 함수는 세기(century)의 변경을 포함하는 범위에서 작동합니다.

`calendar.`**`weekday`**(*year*, *month*, *day*)

year (`1970-…`), *month* (`1-12`), *day* (`1-31`)의 요일(`0`은 월요일)을 반환합니다.

`calendar.`**`weekheader`**(*n*)

약식 요일 이름이 포함된 헤더를 반환합니다. *n*은 한 주의 너비를 문자 수로 지정합니다.

`calendar.`**`monthrange`**(*year*, *month*)

지정된 *year*와 *month*에 대해 월의 첫 번째 날의 요일과 월의 날 수를 반환합니다.

`calendar.`**`monthcalendar`**(*year*, *month*)

한 달의 달력을 나타내는 행렬을 반환합니다. 각 행은 한 주를 나타냅니다; 월 바깥의 날은 0으로 표시됩니다. `setfirstweekday()`로 설정하지 않는 한 각 주는 월요일에 시작합니다.

`calendar.`**`prmonth`**(*theyear*, *themonth*, *w=0*, *l=0*)

`month()`가 반환하는 월의 달력을 인쇄합니다.

`calendar.`**`month`**(*theyear*, *themonth*, *w=0*, *l=0*)

`TextCalendar` 클래스의 `formatmonth()`를 사용하여, 한 달의 달력을 여러 줄 문자열로 반환합니다.

`calendar.`**`prcal`**(*year*, *w=0*, *l=0*, *c=6*, *m=3*)

`calendar()`에서 반환 한 연도 전체 달력을 인쇄합니다.

`calendar.`**`calendar`**(*year*, *w=2*, *l=1*, *c=6*, *m=3*)

`TextCalendar` 클래스의 `formatyear()`를 사용하여 전체 연도의 3-열 달력을 여러 줄 문자열로 반환합니다.

`calendar.`**`timegm`**(*tuple*)

`time` 모듈의 `gmtime()` 함수가 반환하는 것과 같은 시간 튜플을 취하고, 1970년의 시작과 POSIX 인코딩을 가정하는 해당 유닉스 타임스탬프 값을 반환하는 관련이 없지만 편리한 함수. 실제로, `time.gmtime()`과 `timegm()`은 서로의 역 함수입니다.

`calendar` 모듈은 다음 데이터 어트리뷰트를 내보냅니다:

calendar.**day_name**

현재 로케일의 요일을 나타내는 시퀀스, 여기서 Monday 는 요일 번호 0 입니다.

```
>>> import calendar
>>> list(calendar.day_name)
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

calendar.**day_abbr**

현재 로케일의 약식 요일을 나타내는 시퀀스, 여기서 Mon 은 요일 번호 0 입니다.

```
>>> import calendar
>>> list(calendar.day_abbr)
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
```

calendar.**MONDAY**

calendar.**TUESDAY**

calendar.**WEDNESDAY**

calendar.**THURSDAY**

calendar.**FRIDAY**

calendar.**SATURDAY**

calendar.**SUNDAY**

Aliases for the days of the week, where `MONDAY` is `0` and `SUNDAY` is `6`.

Added in version 3.12.

class calendar.**Day**

Enumeration defining days of the week as integer constants. The members of this enumeration are exported to the module scope as *MONDAY* through *SUNDAY*.

Added in version 3.12.

calendar.**month_name**

현재 로케일에서 연중 월을 나타내는 시퀀스. 이는 1 월이 월 번호 1 인 일반적인 규칙을 따르므로, 길이는 13 이고 `month_name[0]` 은 빈 문자열입니다.

```
>>> import calendar
>>> list(calendar.month_name)
['', 'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
↪'September', 'October', 'November', 'December']
```

calendar.**month_abbr**

현재 로케일에서 연중 약식 월을 나타내는 시퀀스. 이는 1 월이 월 번호 1 인 일반적인 규칙을 따르므로, 길이는 13 이고 `month_abbr[0]` 은 빈 문자열입니다.

```
>>> import calendar
>>> list(calendar.month_abbr)
['', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov
↪', 'Dec']
```

calendar.**JANUARY**

calendar.**FEBRUARY**

calendar.**MARCH**

calendar.**APRIL**

calendar.**MAY**

calendar.**JUNE**

calendar.**JULY**

calendar.**AUGUST**

calendar.**SEPTEMBER**

calendar.**OCTOBER**

calendar.**NOVEMBER**

calendar.**DECEMBER**

Aliases for the months of the year, where `JANUARY` is `1` and `DECEMBER` is `12`.

Added in version 3.12.

class calendar.**Month**

Enumeration defining months of the year as integer constants. The members of this enumeration are exported to the module scope as *JANUARY* through *DECEMBER*.

Added in version 3.12.

calendar 모듈은 다음 예외를 정의합니다:

exception calendar.**IllegalMonthError**(*month*)

A subclass of *ValueError*, raised when the given month number is outside of the range 1-12 (inclusive).

month

The invalid month number.

exception calendar.**IllegalWeekdayError**(*weekday*)

A subclass of *ValueError*, raised when the given weekday number is outside of the range 0-6 (inclusive).

weekday

The invalid weekday number.

더 보기

모듈 ***datetime***
time 모듈과 유사한 기능을 가진 날짜와 시간에 대한 객체 지향 인터페이스.

모듈 ***time***
저수준 시간 관련 함수.

8.3.1 Command-line usage

Added in version 2.5.

The *calendar* module can be executed as a script from the command line to interactively print a calendar.

```
python -m calendar [-h] [-L LOCALE] [-e ENCODING] [-t {text,html}]
                   [-w WIDTH] [-l LINES] [-s SPACING] [-m MONTHS] [-c CSS]
                   [-f FIRST_WEEKDAY] [year] [month]
```

For example, to print a calendar for the year 2000:

```
$ python -m calendar 2000
                                  2000

      January                   February                   March
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                 1  2          1  2  3  4  5  6              1  2  3  4  5
 3  4  5  6  7  8  9       7  8  9 10 11 12 13       6  7  8  9 10 11 12
10 11 12 13 14 15 16      14 15 16 17 18 19 20      13 14 15 16 17 18 19
17 18 19 20 21 22 23      21 22 23 24 25 26 27      20 21 22 23 24 25 26
24 25 26 27 28 29 30      28 29                     27 28 29 30 31
31
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
       April                     May                       June
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2       1  2  3  4  5  6  7                1  2  3  4
 3  4  5  6  7  8  9       8  9 10 11 12 13 14       5  6  7  8  9 10 11
10 11 12 13 14 15 16      15 16 17 18 19 20 21      12 13 14 15 16 17 18
17 18 19 20 21 22 23      22 23 24 25 26 27 28      19 20 21 22 23 24 25
24 25 26 27 28 29 30      29 30 31                  26 27 28 29 30

        July                     August                   September
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                1  2          1  2  3  4  5  6                   1  2  3
 3  4  5  6  7  8  9       7  8  9 10 11 12 13       4  5  6  7  8  9 10
10 11 12 13 14 15 16      14 15 16 17 18 19 20      11 12 13 14 15 16 17
17 18 19 20 21 22 23      21 22 23 24 25 26 27      18 19 20 21 22 23 24
24 25 26 27 28 29 30      28 29 30 31               25 26 27 28 29 30
31

      October                  November                  December
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
                   1             1  2  3  4  5                   1  2  3
 2  3  4  5  6  7  8       6  7  8  9 10 11 12       4  5  6  7  8  9 10
 9 10 11 12 13 14 15      13 14 15 16 17 18 19      11 12 13 14 15 16 17
16 17 18 19 20 21 22      20 21 22 23 24 25 26      18 19 20 21 22 23 24
23 24 25 26 27 28 29      27 28 29 30               25 26 27 28 29 30 31
30 31
```

The following options are accepted:

--help, -h

Show the help message and exit.

--locale LOCALE, **-L** LOCALE

The locale to use for month and weekday names. Defaults to English.

--encoding ENCODING, **-e** ENCODING

The encoding to use for output. *--encoding* is required if *--locale* is set.

--type {text,html}, **-t** {text,html}

Print the calendar to the terminal as text, or as an HTML document.

--first-weekday FIRST_WEEKDAY, **-f** FIRST_WEEKDAY

The weekday to start each week. Must be a number between 0 (Monday) and 6 (Sunday). Defaults to 0.

Added in version 3.13.

year

The year to print the calendar for. Defaults to the current year.

month

The month of the specified *year* to print the calendar for. Must be a number between 1 and 12, and may only be used in text mode. Defaults to printing a calendar for the full year.

Text-mode options:

--width WIDTH, **-w** WIDTH

The width of the date column in terminal columns. The date is printed centred in the column. Any value lower than 2 is ignored. Defaults to 2.

--lines LINES, **-l** LINES

The number of lines for each week in terminal rows. The date is printed top-aligned. Any value lower than 1 is ignored. Defaults to 1.

--spacing SPACING, **-s** SPACING

The space between months in columns. Any value lower than 2 is ignored. Defaults to 6.

--months MONTHS, **-m** MONTHS

The number of months printed per row. Defaults to 3.

버전 3.14에서 변경: By default, today' s date is highlighted in color and can be controlled using environment variables.

HTML-mode options:

--css CSS, **-c** CSS

The path of a CSS stylesheet to use for the calendar. This must either be relative to the generated HTML, or an absolute HTTP or `file:///` URL.

8.4 collections — 컨테이너 데이터형

소스 코드: Lib/collections/__init__.py

이 모듈은 파이썬의 범용 내장 컨테이너 *dict*, *list*, *set* 및 *tuple*에 대한 대안을 제공하는 특수 컨테이너 데이터형을 구현합니다.

namedtuple()	이름 붙은 필드를 갖는 튜플 서브 클래스를 만들기 위한 팩토리 함수
deque	양쪽 끝에서 빠르게 추가와 삭제를 할 수 있는 리스트류 컨테이너
ChainMap	여러 매핑의 단일 뷰를 만드는 딕셔너리류 클래스
Counter	해시 가능 객체를 세는 데 사용하는 딕셔너리 서브 클래스
OrderedDict	항목이 추가된 순서를 기억하는 딕셔너리 서브 클래스
defaultdict	누락된 값을 제공하기 위해 팩토리 함수를 호출하는 딕셔너리 서브 클래스
UserDict	더 쉬운 딕셔너리 서브 클래싱을 위해 딕셔너리 객체를 감싸는 래퍼
UserList	더 쉬운 리스트 서브 클래싱을 위해 리스트 객체를 감싸는 래퍼
UserString	더 쉬운 문자열 서브 클래싱을 위해 문자열 객체를 감싸는 래퍼

8.4.1 ChainMap 객체

Added in version 3.3.

ChainMap 클래스는 여러 매핑을 빠르게 연결하여 단일 단위로 취급 할 수 있도록 합니다. 종종 새로운 딕셔너리를 만들고 여러 *update()* 호출을 실행하는 것보다 훨씬 빠릅니다.

이 클래스는 중첩된 스코프를 시뮬레이션하는 데 사용할 수 있으며 템플릿에 유용합니다.

class collections.**ChainMap**(**maps*)

*ChainMap*은 여러 딕셔너리나 다른 매핑을 함께 묶어 갱신 가능한 단일 뷰를 만듭니다. *maps*가 지정되지 않으면, 새 체인에 항상 하나 이상의 매핑이 있도록, 빈 딕셔너리 하나가 제공됩니다.

하부 매핑은 리스트에 저장됩니다. 이 리스트는 공개이며 *maps* 어트리뷰트를 사용하여 액세스하거나 갱신할 수 있습니다. 다른 상태는 없습니다.

조회는 키를 찾을 때까지 하부 매핑을 검색합니다. 반면에, 쓰기, 갱신 및 삭제는 첫 번째 매핑에만 작동합니다.

*ChainMap*은 하부 매핑을 참조로 통합합니다. 따라서 하부 매핑 중 하나가 갱신되면 해당 변경 사항이 *ChainMap*에 반영됩니다.

일반적인 딕셔너리 메서드가 모두 지원됩니다. 또한, *maps* 어트리뷰트, 새 서브 컨텍스트를 만드는 메서드 및 첫 번째 매핑을 제외한 모든 것에 액세스하는 프로퍼티가 있습니다:

maps

사용자 갱신 가능한 매핑 리스트. 리스트는 먼저 검색되는 것에서 나중에 검색되는 순서를 따릅니다. 저장된 유일한 상태이며 검색할 매핑을 변경하도록 수정할 수 있습니다. 리스트는 항상 하나 이상의 매핑이 포함되어야 합니다.

new_child(*m=None, **kwargs*)

새 맵과 그 뒤로 현재 인스턴스의 모든 맵을 포함하는 새 *ChainMap*을 반환합니다. `m`이 지정되면, 매핑 리스트의 맨 앞에 놓이는 새 맵이 됩니다. 지정하지 않으면, 빈 딕셔너리가 사용되므로 `d.new_child()` 호출은 `ChainMap({}, *d.maps)`과 동등합니다. 키워드 인자가 지정되면, 전달된 맵이나 새로운 빈 딕셔너리를 갱신합니다. 이 메서드는 어떤 부모 매핑에 있는 값도 변경하지 않으면서 갱신할 수 있는 서브 컨텍스트를 만드는 데 사용됩니다.

버전 3.4에서 변경: 선택적 `m` 매개 변수가 추가되었습니다.

버전 3.10에서 변경: Keyword arguments support was added.

parents

첫 번째 맵을 제외하고 현재 인스턴스의 모든 맵을 포함하는 새 *ChainMap*을 반환하는 프로퍼티. 검색에서 첫 번째 맵을 건너뛰려고 할 때 유용합니다. 사용 사례는 중첩된 스코프에서 사용되는 `nonlocal` 키워드와 유사합니다. 사용 사례는 내장 *super()* 함수와도 유사합니다. `d.parents`에 대한 참조는 `ChainMap(*d.maps[1:])`과 동등합니다.

*ChainMap*의 이터레이션 순서는 매핑을 마지막에서 첫 번째 방향으로 스캔하여 결정됩니다:

```
>>> baseline = {'music': 'bach', 'art': 'rembrandt'}
>>> adjustments = {'art': 'van gogh', 'opera': 'carmen'}
>>> list(ChainMap(adjustments, baseline))
['music', 'art', 'opera']
```

이것은 마지막 매핑에서 시작하는 일련의 *dict.update()* 호출과 같은 순서를 제공합니다:

```
>>> combined = baseline.copy()
>>> combined.update(adjustments)
>>> list(combined)
['music', 'art', 'opera']
```

버전 3.9에서 변경: **PEP 584**에 지정된, `|`와 `|=` 연산자에 대한 지원이 추가되었습니다.

더 보기

- Enthought CodeTools 패키지의 MultiContext 클래스에는 체인의 모든 매핑으로의 쓰기를 지원하는 옵션이 있습니다.
- 템플릿을 위한 Django의 Context 클래스는 읽기 전용 매핑 체인입니다. 또한 *new_child()* 메서드와 *parents* 프로퍼티와 유사하게 컨텍스트를 푸시(push)하고 팝(pop) 하는 기능이 있습니다.
- 중첩된 컨텍스트 조리법에는 쓰기와 기타 변경이 첫 번째 매핑에만 적용되는지 아니면 체인의 모든 매핑에 적용되는지를 제어하는 옵션이 있습니다.
- 매우 단순화된 체인 맵의 읽기 전용 버전

ChainMap 예제와 조리법

이 절에서는 체인 맵으로 작업하는 다양한 접근 방식을 보여줍니다.

파이썬의 내부 조회 체인을 시뮬레이션하는 예:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

사용자 지정 명령 줄 인자가 환경 변수보다 우선하고, 환경 변수는 기본값보다 우선하도록 하는 예:

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k: v for k, v in vars(namespace).items() if v is not None}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

중첩된 컨텍스트를 시뮬레이션하기 위해 *ChainMap* 클래스를 사용하는 예제 패턴:

```
c = ChainMap()          # 루트 컨텍스트를 만듭니다
d = c.new_child()       # 중첩된 자식 컨텍스트를 만듭니다
e = c.new_child()       # c의 자식, d와 독립적입니다
e.maps[0]               # 현재 컨텍스트 딕셔너리 -- 파이썬의 locals()와 유사합니다
e.maps[-1]              # 루트 컨텍스트 -- 파이썬의 globals()와 유사합니다
e.parents               # 둘러싼 컨텍스트 체인 -- 파이썬의 nonlocal과 유사합니다

d['x'] = 1              # 현재 컨텍스트에 값을 설정합니다
d['x']                  # 컨텍스트 체인에서 첫 번째 키를 가져옵니다
del d['x']              # 현재 컨텍스트에서 지웁니다
list(d)                 # 모든 중첩된 값
k in d                  # 모든 중첩된 값을 검사합니다
len(d)                  # 중첩된 값의 개수
d.items()               # 모든 중첩된 항목
dict(d)                 # 일반 딕셔너리로 펼칩니다
```

ChainMap 클래스는 체인의 첫 번째 매핑만 갱신(쓰기와 삭제)하지만, 조회는 전체 체인을 검색합니다. 그러나, 깊은 쓰기와 삭제가 필요하면, 체인의 더 깊은 곳에서 발견된 키를 갱신하는 서브 클래스를 쉽게 만들 수 있습니다:

```
class DeepChainMap(ChainMap):
    '내부 스코프를 직접 갱신할 수 있는 ChainMap의 변형'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'          # 두 수준 아래에 있는 기존 키를 갱신합니다
>>> d['snake'] = 'red'            # 새 키를 최상단 딕셔너리에 추가합니다
>>> del d['elephant']             # 한 수준 아래에 있는 기존 키를 삭제합니다
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> d                                # 결과를 표시합니다
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})
```

8.4.2 Counter 객체

편리하고 빠르게 개수를 세도록 지원하는 계수기 도구가 제공됩니다. 예를 들면:

```
>>> # 목록에 있는 단어의 빈도를 집계합니다
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
...
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # 햄릿에서 가장 흔한 단어를 찾습니다
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class collections.**Counter**([*iterable-or-mapping*])

*Counter*는 해시 가능 객체를 세기 위한 *dict* 서브 클래스입니다. 요소가 딕셔너리 키로 저장되고 개수가 딕셔너리값으로 저장되는 컬렉션입니다. 개수는 0이나 음수를 포함하는 임의의 정숫값이 될 수 있습니다. *Counter* 클래스는 다른 언어의 백(bag)이나 멀티 셋(multiset)과 유사합니다.

요소는 이터러블로부터 계산되거나 다른 매핑(또는 계수기)에서 초기화됩니다:

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

계수기 객체는 누락된 항목에 대해 *KeyError*를 발생시키는 대신 0을 반환한다는 점을 제외하고 딕셔너리 인터페이스를 갖습니다:

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is␣
↪zero
0
```

개수를 0으로 설정해도 계수기에서 요소가 제거되지 않습니다. 완전히 제거하려면 `del`을 사용하십시오:

```
>>> c['sausage'] = 0                        # counter entry with a zero count
>>> del c['sausage']                        # del actually removes the entry
```

Added in version 3.1.

버전 3.7에서 변경: *dict* 서브 클래스로서, *Counter*는 삽입 순서를 기억하는 기능을 상속했습니다. *Counter* 객체에 대한 수학 연산도 순서를 유지합니다. 결과는 요소가 왼쪽 피연산자에서 처음 발견된 순서로 먼저 나열된 후 오른쪽 피연산자에서 새로 발견되는 순서로 나열되는 순서를 따릅니다.

계수기 객체는 모든 딕셔너리에서 사용할 수 있는 메서드 이외의 추가 메서드를 지원합니다:

elements()

개수만큼 반복되는 요소에 대한 이터레이터를 반환합니다. 요소는 처음 발견되는 순서대로 반환됩니다. 요소의 개수가 1보다 작으면 *elements()*는 이를 무시합니다.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([*n*])

n 개의 가장 흔한 요소와 그 개수를 가장 흔한 것부터 가장 적은 것 순으로 나열한 리스트를 반환합니다. *n*이 생략되거나 None이면, *most_common()*은 계수기의 모든 요소를 반환합니다. 개수가 같은 요소는 처음 발견된 순서를 유지합니다:

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

subtract([*iterable-or-mapping*])

이터러블이나 다른 매핑 (또는 계수기)으로부터 온 요소들을 뺍니다. *dict.update()*와 비슷하지만 교체하는 대신 개수를 뺍니다. 입력과 출력 모두 0이나 음수일 수 있습니다.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

Added in version 3.2.

total()

Compute the sum of the counts.

```
>>> c = Counter(a=10, b=5, c=0)
>>> c.total()
15
```

Added in version 3.10.

일반적인 딕셔너리 메서드를 *Counter* 객체에 사용할 수 있습니다만, 두 메서드는 계수기에서 다르게 동작합니다.

fromkeys(*iterable*)

이 클래스 메서드는 *Counter* 객체에 구현되지 않았습니다.

update([*iterable-or-mapping*])

요소는 이터러블에서 세거나 다른 매핑(또는 계수기)에서 더해집니다. *dict.update()*와 비슷하지만, 교체하는 대신 더합니다. 또한, 이터러블은 (key, value) 쌍의 시퀀스가 아닌, 요소의 시퀀스일 것으로 기대합니다.

Counters support rich comparison operators for equality, subset, and superset relationships: ==, !=, <, <=, >, >=. All of those tests treat missing elements as having zero counts so that Counter(a=1) == Counter(a=1, b=0) returns true.

버전 3.10에서 변경: Rich comparison operations were added.

버전 3.10에서 변경: In equality tests, missing elements are treated as having zero counts. Formerly, Counter(a=3) and Counter(a=3, b=0) were considered distinct.

Counter 객체로 작업하는 일반적인 패턴:

```
c.total()                       # 모든 개수의 총합
c.clear()                       # 모든 개수를 재설정합니다
list(c)                         # 고유한 요소들의 리스트
set(c)                          # 집합으로 변환합니다
dict(c)                         # 일반 딕셔너리로 변환합니다
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
c.items()                       # (elem, cnt) 쌍을 액세스합니다
Counter(dict(list_of_pairs))    # (elem, cnt) 쌍의 리스트를 변환합니다
c.most_common()[:-n-1:-1]       # 가장 드문 n 개의 요소
+c                              # 0과 음의 개수를 제거합니다
```

Counter 객체를 결합하여 멀티 셋(multiset, 개수가 0보다 큰 계수기)을 생성하는 여러 수학 연산이 제공됩니다. 더하기와 빼기는 해당 요소의 개수를 더 하거나 빼서 계수기를 결합합니다. 교집합(intersection)과 합집합(union)은 해당 개수의 최솟값과 최댓값을 반환합니다. 동등성과 포함은 해당 개수를 비교합니다. 각 연산은 부호 있는 개수를 입력으로 받을 수 있지만, 출력은 개수가 0 이하면 결과에서 제외합니다.

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                       # 두 계수기를 합합니다:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                       # 뺍니다 (양의 개수만 유지합니다)
Counter({'a': 2})
>>> c & d                       # 교집합:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                       # 합집합:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
>>> c == d                      # 동등성:  c[x] == d[x]
False
>>> c <= d                      # 포함:  c[x] <= d[x]
False
```

단항 덧셈과 뺄셈은 빈 계수기를 더하거나 빈 계수기를 빼는 것의 줄임 표현입니다.

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

Added in version 3.3: 단항 플러스, 단항 마이너스 및 제자리 멀티 셋 연산에 대한 지원이 추가되었습니다.

참고

계수기는 주로 양의 정수로 작동하여 횟수를 나타내도록 설계되었습니다; 그러나, 다른 형이나 음수 값이 필요한 사용 사례를 불필요하게 배제하지 않도록 주의를 기울였습니다. 이러한 사용 사례에 도움이 되도록, 이 절은 최소 범위와 형 제약 사항을 설명합니다.

- *Counter* 클래스 자체는 키와 값에 제한이 없는 딕셔너리 서브 클래스입니다. 값은 개수를 나타내는 숫자로 의도되었지만, 값 필드에 어떤 것이든 저장할 수 있습니다.
- *most_common()* 메서드는 값에 대해 순서만을 요구합니다.
- `c[key] += 1`과 같은 제자리 연산의 경우, 값 형은 덧셈과 뺄셈만 지원하면 됩니다. 따라서 분수(fractions), 부동 소수점(floats) 및 십진수(decimals)가 작동하고 음수 값이 지원됩니다. *update()*와 *subtract()*에 대해서도 마찬가지인데, 입력과 출력 모두 음수와 0을 허용합니다.
- 멀티 셋(multiset) 메서드는 양의 값에 대한 사용 사례를 위해서만 설계되었습니다. 입력은 음수이거나 0일 수 있지만, 양수 값을 갖는 출력만 만들어집니다. 형 제한은 없지만, 값 형은 더하기, 빼기 및 비교를 지원해야 합니다.
- *elements()* 메서드는 정수 개수를 요구합니다. 0과 음수 개수는 무시합니다.

더 보기

- 스몰토크(Smalltalk)의 Bag 클래스.
- Multisets에 대한 위키피디아 항목.
- 예제가 포함된 C++ multisets 자습서.
- 멀티 셋에 대한 수학 연산과 그 사용 사례에 대해서는, *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19*를 참조하십시오.
- 주어진 요소 집합에 대해 주어진 크기의 모든 서로 다른 멀티 셋을 열거하려면, `itertools.combinations_with_replacement()`를 참조하십시오:

```
map(Counter, combinations_with_replacement('ABC', 2)) # --> AA AB AC BB BC␣
↪CC
```

8.4.3 deque 객체

class collections.**deque**([*iterable*[, *maxlen*]])

*iterable*의 데이터로 왼쪽에서 오른쪽으로 (`append()`를 사용해서) 초기화된 새 데크(deque) 객체를 반환합니다. *iterable*을 지정하지 않으면, 새 데크는 비어 있습니다.

데크는 스택과 큐를 일반화 한 것입니다 (이름은 "deck" 이라고 발음하며 "double-ended queue" 의 약자입니다). 데크는 스레드 안전하고 메모리 효율적인 데크의 양쪽 끝에서의 추가(append)와 팝(pop)을 양쪽에서 거의 같은 $O(1)$ 성능으로 지원합니다.

`list` 객체는 유사한 연산을 지원하지만, 빠른 고정 길이 연산에 최적화되어 있으며, 하부 데이터 표현의 크기와 위치를 모두 변경하는 `pop(0)`과 `insert(0, v)` 연산에 대해 $O(n)$ 메모리 이동 비용이 발생합니다.

*maxlen*이 지정되지 않거나 `None`이면, 데크는 임의의 길이로 커질 수 있습니다. 그렇지 않으면, 데크는 지정된 최대 길이로 제한됩니다. 일단 제한된 길이의 데크가 가득 차면, 새 항목이 추가될 때, 해당하는 수의 항목이 반대쪽 끝에서 삭제됩니다. 제한된 길이의 데크는 유닉스의 `tail` 필터와 유사한 기능을 제공합니다. 또한 가장 최근 활동만 관심이 있는 트랜잭션과 기타 데이터 풀을 추적하는 데 유용합니다.

deque 객체는 다음 메서드를 지원합니다:

append(*x*)

데크의 오른쪽에 *x*를 추가합니다.

appendleft(*x*)

데크의 왼쪽에 *x*를 추가합니다.

clear()

데크에서 모든 요소를 제거하고 길이가 0인 상태로 만듭니다.

copy()

데크의 얕은 복사본을 만듭니다.

Added in version 3.5.

count(*x*)

x 와 같은 데크 요소의 수를 셉니다.

Added in version 3.2.

extend(*iterable*)

iterable 인자에서 온 요소를 추가하여 데크의 오른쪽을 확장합니다.

extendleft(*iterable*)

*iterable*에서 온 요소를 추가하여 데크의 왼쪽을 확장합니다. 일련의 왼쪽 추가는 iterable 인자에 있는 요소의 순서를 뒤집는 결과를 줍니다.

index(*x*[, *start*[, *stop*]])

데크에 있는 *x*의 위치를 반환합니다 (인덱스 *start* 또는 그 이후, 그리고 인덱스 *stop* 이전). 첫 번째 일치를 반환하거나 찾을 수 없으면 `ValueError`를 발생시킵니다.

Added in version 3.5.

insert(*i*, *x*)

*x*를 데크의 *i* 위치에 삽입합니다.

삽입으로 인해 제한된 길이의 데크가 *maxlen* 이상으로 커지면, `IndexError`가 발생합니다.

Added in version 3.5.

pop()

데크의 오른쪽에서 요소를 제거하고 반환합니다. 요소가 없으면, `IndexError`를 발생시킵니다.

popleft()

데크의 왼쪽에서 요소를 제거하고 반환합니다. 요소가 없으면, `IndexError`를 발생시킵니다.

remove(*value*)

*value*의 첫 번째 항목을 제거합니다. 찾을 수 없으면, `ValueError`를 발생시킵니다.

reverse()

데크의 요소들을 제자리에서 순서를 뒤집고 `None`을 반환합니다.

Added in version 3.2.

rotate(*n=1*)

데크를 *n* 단계 오른쪽으로 회전합니다. *n*이 음수이면, 왼쪽으로 회전합니다.

데크가 비어 있지 않으면, 오른쪽으로 한 단계 회전하는 것은 `d.appendleft(d.pop())`과 동등하고, 왼쪽으로 한 단계 회전하는 것은 `d.append(d.popleft())`와 동등합니다.

데크 객체는 하나의 읽기 전용 어트리뷰트도 제공합니다:

maxlen

데크의 최대 크기 또는 제한이 없으면 `None`.

Added in version 3.1.

상기한 것들 외에도, 데크는 이터레이션, 피클링, `len(d)`, `reversed(d)`, `copy.copy(d)`, `copy.deepcopy(d)`, `in` 연산자를 사용한 멤버십 검사 및 첫 번째 요소를 액세스하는 `d[0]`과 같은 서브 스크립트 참조를 지원합니다. 인덱스를 사용하는 액세스는 양쪽 끝에서는 $O(1)$ 이지만 중간에서는 $O(n)$ 으로 느려집니다. 빠른 무작위 액세스를 위해서는 대신 리스트를 사용하십시오.

버전 3.5부터, 데크는 `__add__()`, `__mul__()` 및 `__imul__()`을 지원합니다.

예:

```
>>> from collections import deque
>>> d = deque('ghi')                 # 세 항목의 새 데크를 만듭니다
>>> for elem in d:                   # 데크의 요소들을 이터레이트 합니다
...     print(elem.upper())
G
H
I

>>> d.append('j')                    # 오른쪽에 새 항목을 추가합니다
>>> d.appendleft('f')                # 왼쪽에 새 항목을 추가합니다
>>> d                                # 데크의 표현을 보여줍니다
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # 가장 오른쪽의 항목을 제거하고 반환합니다
'j'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> d.popleft()                         # 가장 왼쪽의 항목을 제거하고 반환합니다
'f'
>>> list(d)                             # 데크의 내용을 나열합니다
['g', 'h', 'i']
>>> d[0]                                # 가장 왼쪽의 항목을 들여다봅니다
'g'
>>> d[-1]                               # 가장 오른쪽의 항목을 들여다봅니다
'i'

>>> list(reversed(d))                   # 데크의 내용을 역순으로 나열합니다
['i', 'h', 'g']
>>> 'h' in d                            # 데크를 검색합니다
True
>>> d.extend('jkl')                     # 여러 요소를 한 번에 추가합니다
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                         # 오른쪽으로 회전합니다
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                        # 왼쪽으로 회전합니다
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))                  # 역순으로 새 데크를 만듭니다
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                           # 데크를 비웁니다
>>> d.pop()                             # 빈 데크에서는 pop 할 수 없습니다
Traceback (most recent call last):
    File "<pyshell#6>", line 1, in -toplevel-
        d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')                 # extendleft() 는 입력의 순서를 뒤집습니다
>>> d
deque(['c', 'b', 'a'])
```

deque 조리법

이 절은 데크로 작업하는 다양한 접근 방식을 보여줍니다.

제한된 길이의 데크는 유닉스의 `tail` 필터와 유사한 기능을 제공합니다:

```
def tail(filename, n=10):
    '파일의 마지막 n 줄을 반환합니다'
    with open(filename) as f:
        return deque(f, n)
```

데크를 사용하는 또 다른 접근법은 오른쪽에 추가하고 왼쪽에서 팝 하여 최근에 추가된 요소의 시퀀스를 유지하는 것입니다:

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # https://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

라운드 로빈 스케줄러(round-robin scheduler)는 *deque*에 저장된 입력 이터레이터로 구현할 수 있습니다. 위치 0에 있는 활성 이터레이터에서 값이 산출됩니다. 그 이터레이터가 소진되면, *popleft()*로 제거할 수 있습니다; 그렇지 않으면, *rotate()* 메서드로 끝으로 보내 순환할 수 있습니다:

```
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    iterators = deque(map(iter, iterables))
    while iterators:
        try:
            while True:
                yield next(iterators[0])
                iterators.rotate(-1)
        except StopIteration:
            # 소진된 이터레이터를 제거합니다.
            iterators.popleft()
```

rotate() 메서드는 *deque* 슬라이싱과 삭제를 구현하는 방법을 제공합니다. 예를 들어, `del d[n]`의 순수 파이썬 구현은 팝 될 요소의 위치를 잡기 위해 `rotate()` 메서드에 의존합니다:

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

deque 슬라이싱을 구현하려면, 대상 요소를 데크의 왼쪽으로 가져오기 위해 *rotate()*를 적용하는 유사한 접근법을 사용하십시오. *popleft()*로 이전 항목을 제거하고, *extend()*로 새 항목을 추가한 다음, 회전을 되돌립니다. 이 접근 방식에 약간의 변형을 가하면, `dup`, `drop`, `swap`, `over`, `pick`, `rot` 및 `roll`과 같은 Forth 스타일 스택 조작을 쉽게 구현할 수 있습니다.

8.4.4 defaultdict 객체

class collections.**defaultdict**(*default_factory=None, /[, ...]*)

새 딕셔너리류 객체를 반환합니다. *defaultdict*는 내장 *dict* 클래스의 서브 클래스입니다. 하나의 메서드를 재정의하고 하나의 쓰기 가능한 인스턴스 변수를 추가합니다. 나머지 기능은 *dict* 클래스와 같고 여기에 설명하지 않습니다.

첫 번째 인자는 *default_factory* 어트리뷰트의 초깃값을 제공합니다; 기본값은 `None`입니다. 나머지 모든 인자는 키워드 인자를 포함하여 *dict* 생성자에 전달될 때와 마찬가지로 취급됩니다.

defaultdict 객체는 표준 *dict* 연산 외에도 다음 메서드를 지원합니다:

__missing__(*key*)

default_factory 어트리뷰트가 `None`이면, *key*를 인자로 사용하는 *KeyError* 예외가 발생합니다.

*default_factory*가 `None`이 아니면, 주어진 *key*에 대한 기본값을 제공하기 위해 인자 없이 호출되며, 반환 값은 *key*로 딕셔너리에 삽입되고 반환됩니다.

*default_factory*를 호출할 때 예외가 발생하면 이 예외는 변경되지 않고 전파됩니다.

이 메서드는 요청된 키를 찾을 수 없을 때 *dict* 클래스의 `__getitem__()` 메서드에 의해 호출됩니다; 이것이 반환하거나 발생시키는 모든 것은, `__getitem__()`이 반환하거나 발생시킵니다.

*__missing__()*은 `__getitem__()` 이외의 어떤 연산에서도 호출되지 않음에 유의하십시오. 이것은 *get()*이 일반 딕셔너리와 마찬가지로 *default_factory*를 사용하지 않고 `None`을 기

본값으로 반환한다는 것을 의미합니다.

defaultdict 객체는 다음 인스턴스 변수를 지원합니다:

default_factory

This attribute is used by the *__missing__()* method; it is initialized from the first argument to the constructor, if present, or to `None`, if absent.

버전 3.9에서 변경: **PEP 584**에 지정된, 병합(`|`)과 업데이트(`|=`) 연산자가 추가되었습니다.

defaultdict 예

*list*를 *default_factory*로 사용하면, 키-값 쌍의 시퀀스를 리스트의 딕셔너리로 쉽게 그룹화 할 수 있습니다:

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the *default_factory* function which returns an empty *list*. The *list.append()* operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the *list.append()* operation adds another value to the list. This technique is simpler and faster than an equivalent technique using *dict.setdefault()*:

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

*default_factory*를 *int*로 설정하면 *defaultdict*를 세는(counting) 데 유용하게 사용할 수 있습니다 (다른 언어의 백(bag)이나 멀티 셋(multiset)처럼):

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

글자가 처음 발견될 때, 매핑에서 누락되었으므로, *default_factory* 함수는 *int()*를 호출하여 기본 계수 0을 제공합니다. 증분 연산은 각 문자의 개수를 쌓아나갑니다.

항상 0을 반환하는 함수 *int()*는 상수 함수의 특별한 경우일 뿐입니다. 상수 함수를 만드는 더 빠르고 유연한 방법은 (단지 0이 아니라) 임의의 상숫값을 제공 할 수 있는 람다 함수를 사용하는 것입니다:

```
>>> def constant_factory(value):
...     return lambda: value
...
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

*default_factory*를 *set*으로 설정하면, *defaultdict*를 집합의 딕셔너리를 만드는 데 유용하게 만듭니다:

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

8.4.5 이름있는 필드를 가진 튜플을 위한 namedtuple() 팩토리 함수

네임드 튜플은 튜플의 각 위치에 의미를 부여하고 더 읽기 쉽고 스스로 설명하는 코드를 만들도록 합니다. 일반 튜플이 사용되는 곳이라면 어디에서나 사용할 수 있으며, 위치 인덱스 대신 이름으로 필드에 액세스하는 기능을 추가합니다.

collections.**namedtuple**(*typename, field_names, *, rename=False, defaults=None, module=None*)

*typename*이라는 이름의 새 튜플 서브 클래스를 반환합니다. 새로운 서브 클래스는 인덱싱되고 이터러블일 뿐만 아니라 어트리뷰트 조회로 액세스 할 수 있는 필드를 갖는 튜플류 객체를 만드는 데 사용됩니다. 서브 클래스의 인스턴스에는 유용한 독스트링(*typename*과 *field_names*를 포함합니다)과 튜플 내용을 `name=value` 형식으로 나열하는 유용한 `__repr__()` 메서드가 있습니다.

*field_names*는 `['x', 'y']`와 같은 문자열의 시퀀스입니다. 또는, *field_names*는 각 필드명이 공백 및/또는 쉼표로 구분된 단일 문자열일 수 있습니다, 예를 들어 `'x y'`나 `'x, y'`.

밑줄로 시작하는 이름을 제외한 모든 유효한 파이썬 식별자를 필드명에 사용할 수 있습니다. 유효한 식별자는 글자, 숫자 및 밑줄로 구성되지만, 숫자나 밑줄로 시작하지 않으며 *class*, *for*, *return*, *global*, *pass* 또는 *raise*와 같은 *keyword*일 수 없습니다.

*rename*이 참이면, 유효하지 않은 필드명은 위치 이름으로 자동 대체됩니다. 예를 들어, `['abc', 'def', 'ghi', 'abc']`는 `['abc', '_1', 'ghi', '_3']`으로 변환되어 키워드 `def`와 중복된 필드명 `abc`를 제거합니다.

*defaults*는 `None`이나 기본값의 이터러블 일 수 있습니다. 기본값이 있는 필드는 기본값이 없는 필드 뒤에 와야 하므로, *defaults*는 가장 오른쪽의 매개 변수에 적용됩니다. 예를 들어, field_names가 `['x', 'y', 'z']`이고 defaults가 `(1, 2)`이면 `x`는 필수 인자이고, `y`의 기본값은 `1`, `z`의 기본값은 `2`입니다.

*module*이 정의되면, 네임드 튜플의 `__module__` 어트리뷰트가 해당 값으로 설정됩니다.

네임드 튜플 인스턴스에는 인스턴스 별 딕셔너리가 없어서, 가볍고 일반 튜플보다 더 많은 메모리가 필요하지 않습니다.

피클링을 지원하려면, 네임드 튜플 클래스를 *typename*과 일치하는 변수에 대입해야 합니다.

버전 3.1에서 변경: *rename*에 대한 지원이 추가되었습니다.

버전 3.6에서 변경: *verbose*와 *rename* 매개 변수는 키워드 전용 인자가 되었습니다.

버전 3.6에서 변경: *module* 매개 변수를 추가했습니다.

버전 3.7에서 변경: *verbose* 매개 변수와 `_source` 어트리뷰트를 제거했습니다.

버전 3.7에서 변경: *defaults* 매개 변수와 *_field_defaults* 어트리뷰트가 추가되었습니다.

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # 위치와 키워드 인자로 인스턴스를 만듭니다
>>> p[0] + p[1]             # 일반 튜플 (11, 22) 처럼 인덱싱할 수 있습니다
33
>>> x, y = p                # 일반 튜플처럼 언팩합니다
>>> x, y
(11, 22)
>>> p.x + p.y               # 필드는 이름으로도 액세스할 수 있습니다
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
33
>>> p                          # name=value 스타일의 가독성 있는 __repr__
Point(x=11, y=22)
```

네임드 튜플은 *csv* 나 *sqlite3* 모듈이 반환한 결과 튜플에 필드 이름을 할당하는 데 특히 유용합니다:

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department,␣
↪paygrade')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

튜플에서 상속된 메서드 외에도 네임드 튜플은 세 가지 추가 메서드와 두 가지 어트리뷰트를 지원합니다. 필드 이름과의 충돌을 방지하기 위해, 메서드와 어트리뷰트 이름은 밑줄로 시작합니다.

classmethod somenamedtuple.**_make**(*iterable*)

기존 시퀀스나 이터러블로 새 인스턴스를 만드는 클래스 메서드.

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple.**_asdict**()

필드 이름을 해당 값으로 매핑하는 새 *dict*를 반환합니다:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
{'x': 11, 'y': 22}
```

버전 3.1에서 변경: 일반 *dict* 대신 *OrderedDict*를 반환합니다.

버전 3.8에서 변경: *OrderedDict* 대신 일반 *dict*를 반환합니다. 파이썬 3.7부터, 일반 딕셔너리의 순서가 유지되도록 보장합니다. *OrderedDict*의 추가 기능이 필요할 때, 제안하는 처방은 결과를 원하는 형으로 캐스트 하는 것입니다: `OrderedDict(nt._asdict())`.

somenamedtuple.**_replace**(***kwargs*)

지정된 필드들을 새로운 값으로 치환하는 네임드 튜플의 새 인스턴스를 반환합니다:

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum],␣
↪timestamp=time.now())
```

Named tuples are also supported by generic function *copy.replace()*.

버전 3.13에서 변경: Raise *TypeError* instead of *ValueError* for invalid keyword arguments.

somenamedtuple.**_fields**

필드 이름을 나열하는 문자열의 튜플. 인트로스펙션과 기존 네임드 튜플에서 새로운 네임드 튜플 형을 만드는 데 유용합니다.

```
>>> p._fields            # 필드 이름을 봅니다
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

somenamedtuple.**_field_defaults**

필드 이름을 기본값으로 매핑하는 딕셔너리.

```
>>> Account = namedtuple('Account', ['type', 'balance'], defaults=[0])
>>> Account._field_defaults
{'balance': 0}
>>> Account('premium')
Account(type='premium', balance=0)
```

이름이 문자열에 저장된 필드를 조회하려면 *getattr()* 함수를 사용하십시오.:

```
>>> getattr(p, 'x')
11
```

딕셔너리를 네임드 튜플로 변환하려면 이중 애스터리스크 연산자를 사용하십시오 (tut-unpacking-arguments 에서 설명합니다).:

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

네임드 튜플은 일반적인 파이썬 클래스이므로, 서브 클래스를 사용하여 기능을 쉽게 추가하거나 변경할 수 있습니다. 계산된 필드와 고정 너비 인쇄 포맷을 추가하는 방법은 다음과 같습니다:

```
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' % (self.x, self.y, self.
↪hypot)

>>> for p in Point(3, 4), Point(14, 5/7):
...     print(p)
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

위에 표시된 서브 클래스는 `__slots__`를 빈 튜플로 설정합니다. 이렇게 하면 인스턴스 딕셔너리 생성을 방지하여 메모리 요구 사항을 낮게 유지할 수 있습니다.

서브 클래싱은 저장된 새 필드를 추가하는 데는 유용하지 않습니다. 대신, *_fields* 어트리뷰트로 새로운 네임드 튜플 형을 만드십시오:

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

`__doc__` 필드에 직접 대입하여 독스트링을 사용자 정의할 수 있습니다:

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'
```

버전 3.5에서 변경: 프로퍼티 독스트링이 쓰기 가능하게 되었습니다.

더 보기

- 네임드 튜플에 형 힌트를 추가하는 방법은 *typing.NamedTuple*을 참조하십시오. 이것은 `class` 키워드를 사용하는 우아한 표기법도 제공합니다:

```
class Component(NamedTuple):
    part_number: int
    weight: float
    description: Optional[str] = None
```

- 튜플 대신 하부 딕셔너리를 기반으로 하는 가변 이름 공간에 대해서는 *types.SimpleNamespace()*를 참조하십시오.
- *dataclasses* 모듈은 사용자 정의 클래스에 생성된 특수 메서드를 자동으로 추가하는 데코레이터와 함수를 제공합니다.

8.4.6 OrderedDict 객체

순서 있는 딕셔너리는 일반 딕셔너리와 비슷하지만, 순서를 다루는 연산과 관련된 몇 가지 추가 기능이 있습니다. 내장 *dict* 클래스가 삽입 순서를 기억하는 기능을 얻었으므로 (이 새로운 동작은 파이썬 3.7에서 보장되었습니다), 이제 덜 중요해졌습니다.

몇 가지 *dict*와의 차이점은 여전히 남아 있습니다:

- 일반 *dict*는 매핑 연산에 매우 적합하도록 설계되었습니다. 삽입 순서 추적은 부차적입니다.
- *OrderedDict*는 순서를 바꾸는 연산에 적합하도록 설계되었습니다. 공간 효율성, 이터레이션 속도 및 갱신 연산의 성능은 부차적입니다.
- *OrderedDict* 알고리즘은 *dict*보다 빈번한 순서 변경 연산을 더 잘 처리할 수 있습니다. 아래의 조리법에서 볼 수 있듯이, 이는 다양한 종류의 LRU 캐시를 구현하는 데 적합합니다.
- *OrderedDict*의 동등 비교 연산은 순서의 일치를 확인합니다.

  A regular *dict* can emulate the order sensitive equality test with `p == q and all(k1 == k2 for k1, k2 in zip(p, q))`.

- *OrderedDict*의 *popitem()* 메서드는 서명이 다릅니다. 어떤 항목을 팝 할지는 지정하는 선택적 인자를 받아들입니다.

  A regular *dict* can emulate OrderedDict' s `od.popitem(last=True)` with `d.popitem()` which is guaranteed to pop the rightmost (last) item.

  A regular *dict* can emulate OrderedDict' s `od.popitem(last=False)` with `(k := next(iter(d)), d.pop(k))` which will return and remove the leftmost (first) item if it exists.

- *OrderedDict*에는 요소를 효율적으로 끝으로 재배치하는 *move_to_end()* 메서드가 있습니다.

  A regular *dict* can emulate OrderedDict' s `od.move_to_end(k, last=True)` with `d[k] = d.pop(k)` which will move the key and its associated value to the rightmost (last) position.

  A regular *dict* does not have an efficient equivalent for OrderedDict' s `od.move_to_end(k, last=False)` which moves the key and its associated value to the leftmost (first) position.

- 파이썬 3.8 이전에는, *dict*에 `__reversed__()` 메서드가 없었습니다.

class collections.**OrderedDict**([*items*])

딕셔너리 순서 재배치에 특화된 메서드가 있는 `dict` 서브 클래스의 인스턴스를 반환합니다.

Added in version 3.1.

popitem(*last=True*)

순서 있는 딕셔너리의 `popitem()` 메서드는 (키, 값) 쌍을 반환하고 제거합니다. *last*가 참이면 쌍이 LIFO 순서로 반환되고, 거짓이면 FIFO (first-in, first-out - 선입선출) 순서로 반환됩니다.

move_to_end(*key*, *last=True*)

기존 *key*를 순서 있는 딕셔너리의 한쪽 끝으로 옮깁니다. *last*가 참(기본값)이면 항목은 오른쪽 끝으로 이동하고, *last*가 거짓이면 처음으로 이동합니다. *key*가 존재하지 않으면 `KeyError`가 발생합니다:

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d)
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d)
'bacde'
```

Added in version 3.2.

일반적인 매핑 메서드 외에도 순서 있는 딕셔너리는 `reversed()`를 사용하는 역 이터레이션을 지원합니다.

`OrderedDict` 객체 간의 동등성 (equality) 테스트는 순서를 고려하며 대략 `list(od1.items())==list(od2.items())`와 동등합니다.

`OrderedDict` 객체와 다른 `Mapping` 객체 간의 동등성 테스트는 일반 딕셔너리처럼 순서를 고려하지 않습니다. 이 때문에 `OrderedDict` 객체를 일반 딕셔너리가 사용되는 모든 곳에 대체 할 수 있습니다.

버전 3.5에서 변경: `OrderedDict`의 items, keys 및 values 뷰는 이제 `reversed()`를 사용하는 역 이터레이션을 지원합니다.

버전 3.6에서 변경: **PEP 468**을 수락함에 따라, `OrderedDict` 생성자와 `update()` 메서드로 전달된 키워드 인자의 순서가 보존됩니다.

버전 3.9에서 변경: **PEP 584**에 지정된, 병합(`|`)과 업데이트(`|=`) 연산자가 추가되었습니다.

OrderedDict 예제와 조리법

키가 마지막에 삽입된 순서를 기억하는 순서 있는 딕셔너리 변형을 만드는 것은 간단합니다. 새 항목이 기존 항목을 덮어쓰면, 원래 삽입 위치가 변경되고 끝으로 이동합니다:

```
class LastUpdatedOrderedDict(OrderedDict):
    '키가 마지막에 삽입된 순서로 항목을 저장합니다'

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

`OrderedDict`는 `functools.lru_cache()`의 변형을 구현하는 데에도 유용합니다:

```
from collections import OrderedDict
from time import time

class TimeBoundedLRU:
    "오래된 항목을 무효화하고 새로 고치는 LRU 캐시."

    def __init__(self, func, maxsize=128, maxage=30):
        self.cache = OrderedDict()       # { args : (timestamp, result)}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        self.func = func
        self.maxsize = maxsize
        self.maxage = maxage

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            timestamp, result = self.cache[args]
            if time() - timestamp <= self.maxage:
                return result
        result = self.func(*args)
        self.cache[args] = time(), result
        if len(self.cache) > self.maxsize:
            self.cache.popitem(last=False)
        return result
```

```
class MultiHitLRUCache:
    """ 여러 번 요청될 때까지 결과 캐싱을
         지연시키는 LRU 캐시.

        일회성 요청으로 인해 LRU 캐시를 비우는 것을 방지하기 위해,
        요청이 두 번 이상 이루어질 때까지 개시하지 않습니다.

    """

    def __init__(self, func, maxsize=128, maxrequests=4096, cache_after=1):
        self.requests = OrderedDict()    # { uncached_key : request_count }
        self.cache = OrderedDict()       # { cached_key : function_result }
        self.func = func
        self.maxrequests = maxrequests   # 캐시되지 않은 요청의 최대 수
        self.maxsize = maxsize           # 저장된 반환값의 최대 수
        self.cache_after = cache_after

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            return self.cache[args]
        result = self.func(*args)
        self.requests[args] = self.requests.get(args, 0) + 1
        if self.requests[args] <= self.cache_after:
            self.requests.move_to_end(args)
            if len(self.requests) > self.maxrequests:
                self.requests.popitem(last=False)
        else:
            self.requests.pop(args, None)
            self.cache[args] = result
            if len(self.cache) > self.maxsize:
                self.cache.popitem(last=False)
        return result
```

8.4.7 UserDict 객체

UserDict 클래스는 딕셔너리 객체를 감싸는 래퍼 역할을 합니다. 이 클래스의 필요성은 *dict*에서 직접 서브 클래싱 할 수 있는 능력에 의해 부분적으로 대체되었습니다; 그러나 하부 딕셔너리를 어트리뷰트로 액세스 할 수 있어서, 이 클래스를 사용하면 작업하기가 더 쉬울 수 있습니다.

class collections.**UserDict**([*initialdata*])

딕셔너리를 시뮬레이트 하는 클래스. 인스턴스의 내용은 일반 딕셔너리에 보관되며, *UserDict* 인스턴스의 *data* 어트리뷰트를 통해 액세스 할 수 있습니다. *initialdata*가 제공되면 *data*는 그 내용으로 초기화됩니다; *initialdata*에 대한 참조는 유지되지 않아서, 다른 용도로 사용할 수 있습니다.

UserDict 인스턴스는 매핑의 메서드와 연산을 지원할 뿐만 아니라, 다음과 같은 어트리뷰트를 제공합니다:

data

UserDict 클래스의 내용을 저장하는 데 사용되는 실제 딕셔너리.

8.4.8 UserList 객체

이 클래스는 리스트 객체를 둘러싸는 래퍼 역할을 합니다. 여러분 자신의 리스트류 클래스가 상속하고 기존 메서드를 재정의하거나 새로운 메서드를 추가할 수 있는 유용한 베이스 클래스입니다. 이런 식으로 리스트에 새로운 동작을 추가 할 수 있습니다.

이 클래스의 필요성은 *list*에서 직접 서브 클래싱할 수 있는 능력에 의해 부분적으로 대체되었습니다; 그러나 하부 리스트에 어트리뷰트로 액세스할 수 있어서, 이 클래스를 사용하면 작업하기가 더 쉬울 수 있습니다.

class collections.**UserList**([*list*])

리스트를 시뮬레이트 하는 클래스. 인스턴스의 내용은 일반 리스트로 유지되며 *UserList* 인스턴스의 *data* 어트리뷰트를 통해 액세스 할 수 있습니다. 인스턴스의 내용은 초기에 *list*의 사본으로 설정되며, 기본값은 빈 목록 `[]` 입니다. *list*는 모든 이터러블일 수 있습니다, 예를 들어 실제 파이썬 리스트나 *UserList* 객체.

UserList 인스턴스는 가변 시퀀스의 메서드와 연산을 지원할 뿐만 아니라 다음 어트리뷰트를 제공합니다:

data

UserList 클래스의 내용을 저장하는 데 사용되는 실제 *list* 객체.

서브 클래싱 요구 사항: *UserList*의 서브 클래스는 인자가 없거나 하나의 인자로 호출 할 수 있는 생성자를 제공해야 합니다. 새 시퀀스를 반환하는 리스트 연산은 실제 구현 클래스의 인스턴스를 만들려고 시도합니다. 이를 위해, 데이터 소스로 사용되는 시퀀스 객체인 단일 매개 변수로 생성자를 호출할 수 있다고 가정합니다.

파생 클래스가 이 요구 사항을 준수하고 싶지 않다면, 이 클래스에서 지원하는 모든 특수 메서드를 재정의해야 합니다; 이때 제공해야 하는 메서드에 대한 정보는 소스를 참조하십시오.

8.4.9 UserString 객체

UserString 클래스는 문자열 객체를 둘러싸는 래퍼 역할을 합니다. 이 클래스의 필요성은 *str*에서 직접 서브 클래싱할 수 있는 능력에 의해 부분적으로 대체되었습니다; 그러나 하부 문자열을 어트리뷰트로 액세스할 수 있어서, 이 클래스를 사용하면 작업하기가 더 쉬울 수 있습니다.

class collections.**UserString**(*seq*)

문자열 객체를 시뮬레이트 하는 클래스. 인스턴스의 내용은 일반 문자열 객체로 유지되며, *UserString* 인스턴스의 *data* 어트리뷰트를 통해 액세스 할 수 있습니다. 인스턴스의 내용은 처음에 *seq*의 사본으로 설정됩니다. *seq* 인자는 내장 *str()* 함수를 사용하여 문자열로 변환 할 수 있는 모든 객체가 될 수 있습니다.

UserString 인스턴스는 문자열의 메서드와 연산을 지원할 뿐만 아니라 다음과 같은 어트리뷰트를 제공합니다:

data

UserString 클래스의 내용을 저장하는 데 사용되는 실제 *str* 객체.

버전 3.5에서 변경: 새로운 메서드 `__getnewargs__`, `__rmod__`, `casefold`, `format_map`, `isprintable` 및 `maketrans`.

8.5 collections.abc — 컨테이너의 추상 베이스 클래스

Added in version 3.3: 이전에는, 이 모듈이 *collections* 모듈의 일부였습니다.

소스 코드: Lib/_collections_abc.py

이 모듈은 클래스가 특정 인터페이스를 제공하는지를 검사하는 데 사용할 수 있는 추상 베이스 클래스를 제공합니다; 예를 들어, 해시 가능인지 또는 매핑인지입니다.

An *issubclass()* or *isinstance()* test for an interface works in one of three ways.

1) A newly written class can inherit directly from one of the abstract base classes. The class must supply the required abstract methods. The remaining mixin methods come from inheritance and can be overridden if desired. Other methods may be added as needed:

```
class C(Sequence):                          # 직접 상속
    def __init__(self): ...                 # ABC 가 요구하지 않는 추가 메서드
    def __getitem__(self, index):  ...      # 필수 추상 메서드
    def __len__(self):  ...                 # 필수 추상 메서드
    def count(self, value): ...             # 선택적으로 믹스인 메서드를 재정의합니다
```

```
>>> issubclass(C, Sequence)
True
>>> isinstance(C(), Sequence)
True
```

2) Existing classes and built-in classes can be registered as "virtual subclasses" of the ABCs. Those classes should define the full API including all of the abstract methods and all of the mixin methods. This lets users rely on *issubclass()* or *isinstance()* tests to determine whether the full interface is supported. The exception to this rule is for methods that are automatically inferred from the rest of the API:

```
class D:                                    # 상속 없음
    def __init__(self): ...                 # ABC 가 요구하지 않는 추가 메서드
    def __getitem__(self, index):  ...      # 추상 메서드
    def __len__(self):  ...                 # 추상 메서드
    def count(self, value): ...             # 믹스인 메서드
    def index(self, value): ...             # 믹스인 메서드

Sequence.register(D)                        # 상속하는 대신 등록합니다
```

```
>>> issubclass(D, Sequence)
True
>>> isinstance(D(), Sequence)
True
```

In this example, class `D` does not need to define `__contains__`, `__iter__`, and `__reversed__` because the in-operator, the *iteration* logic, and the *reversed()* function automatically fall back to using `__getitem__` and `__len__`.

3) Some simple interfaces are directly recognizable by the presence of the required methods (unless those methods have been set to *None*):

```
class E:
    def __iter__(self): ...
    def __next__(self): ...
```

```
>>> issubclass(E, Iterable)
True
>>> isinstance(E(), Iterable)
True
```

Complex interfaces do not support this last technique because an interface is more than just the presence of method names. Interfaces specify semantics and relationships between methods that cannot be inferred solely from the presence of specific method names. For example, knowing that a class supplies `__getitem__`, `__len__`, and `__iter__` is insufficient for distinguishing a *Sequence* from a *Mapping*.

Added in version 3.9: These abstract classes now support `[]`. See 제네릭 에일리어스 형 and **PEP 585**.

8.5.1 Collections 추상 베이스 클래스

collections 모듈은 다음과 같은 *ABC*를 제공합니다:

ABC	상속	추상 메서드	믹스인 메서드
Container[1]		__contains__	
Hashable[1]		__hash__	
Iterable[1,2]		__iter__	
Iterator[1]	*Iterable*	__next__	__iter__
Reversible[1]	*Iterable*	__reversed__	
Generator[1]	*Iterator*	send, throw	close, __iter__, __next__
Sized[1]		__len__	
Callable[1]		__call__	
Collection[1]	*Sized*, *Iterable*, *Containe*	__contains__, __iter__, __len__	
Sequence	*Reversib* *Collecti*	__getitem__, __len__	__contains__, __iter__, __reversed__, index 및 count
MutableSequence	*Sequence*	__getitem__, __setitem__, __delitem__, __len__, insert	상속된 *Sequence* 메서드와 append, clear, reverse, extend, pop, remove 및 __iadd__
ByteString	*Sequence*	__getitem__, __len__	상속된 *Sequence* 메서드
Set	*Collecti*	__contains__, __iter__, __len__	__le__, __lt__, __eq__, __ne__, __gt__, __ge__, __and__, __or__, __sub__, __rsub__, __xor__, __rxor__ 및 isdisjoint
MutableSet	*Set*	__contains__, __iter__, __len__, add, discard	상속된 *Set* 메서드와 clear, pop, remove, __ior__, __iand__, __ixor__ 및 __isub__
Mapping	*Collecti*	__getitem__, __iter__, __len__	__contains__, keys, items, values, get, __eq__ 및 __ne__
MutableMapping	*Mapping*	__getitem__, __setitem__, __delitem__, __iter__, __len__	상속된 *Mapping* 메서드와 pop, popitem, clear, update 및 setdefault
MappingView	*Sized*		__init__, __len__ 및 __repr__
ItemsView	*MappingV* *Set*		__contains__, __iter__
KeysView	*MappingV* *Set*		__contains__, __iter__
ValuesView	*MappingV* *Collecti*		__contains__, __iter__
Awaitable[1]		__await__	
Coroutine[1]	*Awaitabl*	send, throw	close
AsyncIterable[1]		__aiter__	
AsyncIterator[1]	*AsyncIte*	__anext__	__aiter__
AsyncGenerator[1]	*AsyncIte*	asend, athrow	aclose, __aiter__, __anext__
Buffer[1]		__buffer__	

[1] These ABCs override *__subclasshook__()* to support testing an interface by verifying the required methods are present and have not been set to *None*. This only works for simple interfaces. More complex interfaces require registration or direct subclassing.

[2] isinstance(obj, Iterable) 를 검사하면 *Iterable*로 등록되었거나 *__iter__()* 메서드가 있는 클래스를 감지하지만, __getitem__() 메서드로 이터레이트 하는 클래스는 감지하지 않습니다. 객체가 이터러블인지를 확인하는 유일하게 신뢰성 있는 방법은 iter(obj) 를 호출하는 것입니다.

8.5.2 Collections 추상 베이스 클래스 – 상세한 설명

class collections.abc.**Container**

`__contains__()` 메서드를 제공하는 클래스의 ABC.

class collections.abc.**Hashable**

`__hash__()` 메서드를 제공하는 클래스의 ABC.

class collections.abc.**Sized**

`__len__()` 메서드를 제공하는 클래스의 ABC.

class collections.abc.**Callable**

`__call__()` 메서드를 제공하는 클래스의 ABC.

See *Annotating callable objects* for details on how to use `Callable` in type annotations.

class collections.abc.**Iterable**

`__iter__()` 메서드를 제공하는 클래스의 ABC.

`isinstance(obj, Iterable)` 를 검사하면 *`Iterable`*로 등록되었거나 *`__iter__()`* 메서드가 있는 클래스를 감지하지만, `__getitem__()` 메서드로 이터레이트 하는 클래스는 감지하지 않습니다. 객체가 이터러블인지를 확인하는 유일하게 신뢰성 있는 방법은 `iter(obj)` 를 호출하는 것입니다.

class collections.abc.**Collection**

길이가 있는 이터러블 컨테이너 클래스의 ABC.

Added in version 3.6.

class collections.abc.**Iterator**

`__iter__()` 와 *`__next__()`* 메서드를 제공하는 클래스의 ABC. 이터레이터의 정의도 참조하십시오.

class collections.abc.**Reversible**

`__reversed__()` 메서드도 제공하는 이터러블 클래스의 ABC.

Added in version 3.6.

class collections.abc.**Generator**

`send()`,`throw()` 및 `close()` 메서드로 이터레이터를 확장하는 **PEP 342**에 정의된 프로토콜을 구현하는 제너레이터 클래스의 ABC.

See *Annotating generators and coroutines* for details on using `Generator` in type annotations.

Added in version 3.5.

class collections.abc.**Sequence**

class collections.abc.**MutableSequence**

class collections.abc.**ByteString**

읽기 전용과 가변 시퀀스의 ABC.

Implementation note: Some of the mixin methods, such as *`__iter__()`*, `__reversed__()`, and *`index()`* make repeated calls to the underlying `__getitem__()` method. Consequently, if `__getitem__()` is implemented with constant access speed, the mixin methods will have linear performance; however, if the underlying method is linear (as it would be with a linked list), the mixins will have quadratic performance and will likely need to be overridden.

index(*value*, *start=0*, *stop=None*)

Return first index of *value*.

Raises *`ValueError`* if the value is not present.

Supporting the *start* and *stop* arguments is optional, but recommended.

버전 3.5에서 변경: The *`index()`* method gained support for the *stop* and *start* arguments.

Deprecated since version 3.12, will be removed in version 3.17: The *ByteString* ABC has been deprecated.

Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use *Buffer* or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

`ByteString` was originally intended to be an abstract class that would serve as a supertype of both *bytes* and *bytearray*. However, since the ABC never had any methods, knowing that an object was an instance of `ByteString` never actually told you anything useful about the object. Other common buffer types such as *memoryview* were also never understood as subtypes of `ByteString` (either at runtime or by static type checkers).

See **PEP 688** for more details.

class `collections.abc.`**Set**

class `collections.abc.`**MutableSet**

읽기 전용과 가변 집합의 ABC.

class `collections.abc.`**Mapping**

class `collections.abc.`**MutableMapping**

읽기 전용과 가변 매핑의 ABC.

class `collections.abc.`**MappingView**

class `collections.abc.`**ItemsView**

class `collections.abc.`**KeysView**

class `collections.abc.`**ValuesView**

매핑, 항목, 키 및 값 뷰의 ABC.

class `collections.abc.`**Awaitable**

`await` 표현식에서 사용할 수 있는 어웨이터블 객체의 ABC. 사용자 정의 구현은 `__await__()` 메서드를 제공해야 합니다.

코루틴 객체와 *Coroutine* ABC의 인스턴스는 모두 이 ABC의 인스턴스입니다.

> **참고**
>
> CPython에서, 제너레이터 기반 코루틴(*@types.coroutine*으로 데코레이트 된 제너레이터)은, `__await__()` 메서드가 없어도 어웨이터블 입니다. 이들에 대해 `isinstance(gencoro, Awaitable)`를 사용하면 `False`가 반환됩니다. 이들을 감지하려면 *inspect.isawaitable()*을 사용하십시오.

Added in version 3.5.

class `collections.abc.`**Coroutine**

코루틴 호환 클래스의 ABC. coroutine-objects에 정의된 다음 메서드를 구현합니다: `send()`, `throw()` 및 `close()`. 사용자 정의 구현은 `__await__()`도 구현해야 합니다. 모든 *Coroutine* 인스턴스는 *Awaitable*의 인스턴스이기도 합니다.

> **참고**
>
> CPython에서, 제너레이터 기반 코루틴(*@types.coroutine*으로 데코레이트 된 제너레이터)은, `__await__()` 메서드가 없어도 어웨이터블 입니다. 이들에 대해 `isinstance(gencoro, Coroutine)`을 사용하면 `False`가 반환됩니다. 이들을 감지하려면 *inspect.isawaitable()*을 사용하십시오.

See *Annotating generators and coroutines* for details on using `Coroutine` in type annotations. The variance and order of type parameters correspond to those of *Generator*.

Added in version 3.5.

class collections.abc.**AsyncIterable**

__aiter__ 메서드를 제공하는 클래스의 ABC. 비동기 이터러블의 정의도 참조하십시오.

Added in version 3.5.

class collections.abc.**AsyncIterator**

__aiter__ 와 __anext__ 메서드를 제공하는 클래스의 ABC. 비동기 이터레이터의 정의도 참조하십시오.

Added in version 3.5.

class collections.abc.**AsyncGenerator**

PEP 525와 **PEP 492**에 정의된 프로토콜을 구현하는 비동기 제너레이터 클래스의 ABC.

See *Annotating generators and coroutines* for details on using `AsyncGenerator` in type annotations.

Added in version 3.6.

class collections.abc.**Buffer**

ABC for classes that provide the `__buffer__()` method, implementing the buffer protocol. See **PEP 688**.

Added in version 3.12.

8.5.3 예제와 조리법

ABC들은 클래스나 인스턴스가 특정 기능을 제공하는지 묻는 것을 허용합니다, 예를 들어:

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

ABC 중 일부는 믹스인으로도 유용하여 컨테이너 API를 지원하는 클래스를 쉽게 개발할 수 있게 합니다. 예를 들어, 전체 *Set* API를 지원하는 클래스를 작성하려면, `__contains__()`, *__iter__()* 및 `__len__()` 의 세 가지 하부 추상 메서드 만 제공하면 됩니다. ABC는 `__and__()` 와 *isdisjoint()*와 같은 나머지 메서드를 제공합니다:

```
class ListBasedSet(collections.abc.Set):
    ''' 속도보다 공간을 선호하고 집합 원소가
        해시 가능할 필요가 없는 대안 집합 구현. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)

    def __contains__(self, value):
        return value in self.elements

    def __len__(self):
        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # __and__() 메서드는 자동으로 지원됩니다
```

*Set*과 *MutableSet*을 믹스인으로 사용할 때의 주의 사항:

(1) 일부 집합 연산은 새로운 집합을 만들기 때문에, 기본 믹스인 메서드는 이터러블로부터 새 인스턴스를 만드는 방법이 필요합니다. 클래스 생성자가 `ClassName(iterable)` 형식의 서명을 가진 것으로 가

정합니다. 이 가정은 새로운 집합을 생성하기 위해 `cls(iterable)` 를 호출하는 `_from_iterable()` 이라는 내부 *classmethod*로 분리되었습니다. *Set* 믹스인이 다른 생성자 서명을 갖는 클래스에서 사용되고 있으면, 이터러블 인자로부터 새 인스턴스를 생성할 수 있는 클래스 메서드나 일반 메서드로 `_from_iterable()` 을 재정의해야 합니다.

(2) 비교를 재정의하려면 (의미는 고정되었으므로, 아마도 속도 때문에), `__le__()` 와 `__ge__()` 를 재정의하십시오, 그러면 다른 연산은 자동으로 맞춰집니다.

(3) *Set* 믹스인은 집합의 해시값을 계산하는 `_hash()` 메서드를 제공합니다; 그러나 모든 집합이 해시 가능하거나 불변이지는 않기 때문에 `__hash__()` 는 정의되지 않습니다. 믹스인을 사용하여 집합 해시 가능성을 추가하려면, *Set()* 와 *Hashable()* 을 모두 상속한 다음, `__hash__ = Set._hash` 를 정의하십시오.

더 보기

- *MutableSet* 으로 구축한 예제 OrderedSet 조리법.
- ABC에 대한 자세한 내용은, *abc* 모듈과 **PEP 3119**를 참조하십시오.

8.6 heapq — 힙 큐 알고리즘

소스 코드: Lib/heapq.py

이 모듈은 우선순위 큐 알고리즘이라고도 하는 힙(heap) 큐 알고리즘의 구현을 제공합니다.

Min-heaps are binary trees for which every parent node has a value less than or equal to any of its children. We refer to this condition as the heap invariant.

For min-heaps, this implementation uses lists for which `heap[k] <= heap[2*k+1]` and `heap[k] <= heap[2*k+2]` for all *k* for which the compared elements exist. Elements are counted from zero. The interesting property of a min-heap is that its smallest element is always the root, `heap[0]`.

Max-heaps satisfy the reverse invariant: every parent node has a value *greater* than any of its children. These are implemented as lists for which `maxheap[2*k+1] <= maxheap[k]` and `maxheap[2*k+2] <= maxheap[k]` for all *k* for which the compared elements exist. The root, `maxheap[0]`, contains the *largest* element; `heap.sort(reverse=True)` maintains the max-heap invariant.

The `heapq` API differs from textbook heap algorithms in two aspects: (a) We use zero-based indexing. This makes the relationship between the index for a node and the indexes for its children slightly less obvious, but is more suitable since Python uses zero-based indexing. (b) Textbooks often focus on max-heaps, due to their suitability for in-place sorting. Our implementation favors min-heaps as they better correspond to Python *lists*.

These two aspects make it possible to view the heap as a regular Python list without surprises: `heap[0]` is the smallest item, and `heap.sort()` maintains the heap invariant!

Like *list.sort()*, this implementation uses only the < operator for comparisons, for both min-heaps and max-heaps.

In the API below, and in this documentation, the unqualified term *heap* generally refers to a min-heap. The API for max-heaps is named using a `_max` suffix.

To create a heap, use a list initialized as `[]`, or transform an existing list into a min-heap or max-heap using the *heapify()* or *heapify_max()* functions, respectively.

The following functions are provided for min-heaps:

`heapq.`**`heappush`**(*heap*, *item*)

Push the value *item* onto the *heap*, maintaining the min-heap invariant.

heapq.**heappop**(*heap*)

Pop and return the smallest item from the *heap*, maintaining the min-heap invariant. If the heap is empty, `IndexError` is raised. To access the smallest item without popping it, use `heap[0]`.

heapq.**heappushpop**(*heap*, *item*)

힙에 *item*을 푸시한 다음, *heap*에서 가장 작은 항목을 팝하고 반환합니다. 결합한 액션은 `heappush()`한 다음 `heappop()`을 별도로 호출하는 것보다 더 효율적으로 실행합니다.

heapq.**heapify**(*x*)

Transform list *x* into a min-heap, in-place, in linear time.

heapq.**heapreplace**(*heap*, *item*)

*heap*에서 가장 작은 항목을 팝하고 반환하며, 새로운 *item*도 푸시합니다. 힙 크기는 변경되지 않습니다. 힙이 비어 있으면, `IndexError`가 발생합니다.

이 한 단계 연산은 `heappop()`한 다음 `heappush()`하는 것보다 더 효율적이며 고정 크기 힙을 사용할 때 더 적합 할 수 있습니다. 팝/푸시 조합은 항상 힙에서 요소를 반환하고 그것을 *item*으로 대체합니다.

반환된 값은 추가된 *item*보다 클 수 있습니다. 그것이 바람직하지 않다면, 대신 `heappushpop()` 사용을 고려하십시오. 푸시/팝 조합은 두 값 중 작은 값을 반환하여, 힙에 큰 값을 남겨 둡니다.

For max-heaps, the following functions are provided:

heapq.**heapify_max**(*x*)

Transform list *x* into a max-heap, in-place, in linear time.

Added in version 3.14.

heapq.**heappush_max**(*heap*, *item*)

Push the value *item* onto the max-heap *heap*, maintaining the max-heap invariant.

Added in version 3.14.

heapq.**heappop_max**(*heap*)

Pop and return the largest item from the max-heap *heap*, maintaining the max-heap invariant. If the max-heap is empty, `IndexError` is raised. To access the largest item without popping it, use `maxheap[0]`.

Added in version 3.14.

heapq.**heappushpop_max**(*heap*, *item*)

Push *item* on the max-heap *heap*, then pop and return the largest item from *heap*. The combined action runs more efficiently than `heappush_max()` followed by a separate call to `heappop_max()`.

Added in version 3.14.

heapq.**heapreplace_max**(*heap*, *item*)

Pop and return the largest item from the max-heap *heap* and also push the new *item*. The max-heap size doesn't change. If the max-heap is empty, `IndexError` is raised.

The value returned may be smaller than the *item* added. Refer to the analogous function `heapreplace()` for detailed usage notes.

Added in version 3.14.

이 모듈은 또한 힙 기반의 세 가지 범용 함수를 제공합니다.

heapq.**merge**(**iterables, key=None, reverse=False*)

여러 정렬된 입력을 단일 정렬된 출력으로 병합합니다 (예를 들어, 여러 로그 파일에서 타임 스탬프된 항목을 병합합니다). 정렬된 값에 대한 이터레이터를 반환합니다.

`sorted(itertools.chain(*iterables))`와 비슷하지만 이터러블을 반환하고, 데이터를 한 번에 메모리로 가져오지 않으며, 각 입력 스트림이 이미 (최소에서 최대로) 정렬된 것으로 가정합니다.

키워드 인자로 지정해야 하는 두 개의 선택적 인자가 있습니다.

*key*는 각 입력 요소에서 비교 키를 추출하는 데 사용되는 단일 인자의 키 함수를 지정합니다. 기본값은 `None`입니다 (요소를 직접 비교합니다).

*reverse*는 불리언 값입니다. `True`로 설정하면, 각 비교가 반대로 된 것처럼 입력 요소가 병합됩니다. `sorted(itertools.chain(*iterables), reverse=True)`와 유사한 동작을 달성하려면 모든 이터러블이 최대에서 최소로 정렬되어 있어야 합니다.

버전 3.5에서 변경: 선택적 *key*와 *reverse* 매개 변수를 추가했습니다.

`heapq.`**`nlargest`**(*n*, *iterable*, *key=None*)

*iterable*에 의해 정의된 데이터 집합에서 *n* 개의 가장 큰 요소로 구성된 리스트를 반환합니다. *key*가 제공되면 *iterable*의 각 요소에서 비교 키를 추출하는 데 사용되는 단일 인자 함수를 지정합니다 (예를 들어, `key=str.lower`). 다음과 동등합니다: `sorted(iterable, key=key, reverse=True)[:n]`.

`heapq.`**`nsmallest`**(*n*, *iterable*, *key=None*)

*iterable*에 의해 정의된 데이터 집합에서 *n* 개의 가장 작은 요소로 구성된 리스트를 반환합니다. *key*가 제공되면 *iterable*의 각 요소에서 비교 키를 추출하는 데 사용되는 단일 인자 함수를 지정합니다 (예를 들어, `key=str.lower`). 다음과 동등합니다: `sorted(iterable, key=key)[:n]`.

마지막 두 함수는 작은 *n* 값에서 가장 잘 동작합니다. 값이 크면, *`sorted()`* 기능을 사용하는 것이 더 효율적입니다. 또한, `n==1`일 때는, 내장 *`min()`*과 *`max()`* 함수를 사용하는 것이 더 효율적입니다. 이 함수를 반복해서 사용해야 하면, iterable을 실제 힙으로 바꾸는 것이 좋습니다.

8.6.1 기본 예

힙 정렬은 모든 값을 힙으로 푸시한 다음 한 번에 하나씩 가장 작은 값을 팝 하여 구현할 수 있습니다:

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

이것은 `sorted(iterable)`과 비슷하지만, *`sorted()`*와 달리, 이 구현은 안정적(stable)이지 않습니다.

힙 요소는 튜플일 수 있습니다. 추적하는 기본 레코드와 함께 비교 값(가령 작업 우선순위)을 지정하는 데 유용합니다:

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.6.2 우선순위 큐 구현 참고 사항

우선순위 큐는 힙의 일반적인 사용이며, 몇 가지 구현 과제가 있습니다:

- 정렬 안정성: 우선순위가 같은 두 개의 작업을 어떻게 원래 추가된 순서대로 반환합니까?
- 우선순위가 같고 작업에 기본 비교 순서가 없으면 (우선순위, 작업) 쌍에 대한 튜플 비교가 성립하지 않습니다.
- 작업의 우선순위가 변경되면, 어떻게 힙의 새로운 위치로 옮깁니까?
- 또는 계류 중인 작업을 삭제해야 하면, 작업을 어떻게 찾고 큐에서 제거합니까?

처음 두 가지 과제에 대한 해결책은 항목을 우선순위, 항목 수 및 작업을 포함하는 3-요소 리스트로 저장하는 것입니다. 항목 수는 순위 결정자 역할을 하므로 우선순위가 같은 두 작업이 추가된 순서대로 반환됩니다. 두 항목 수가 같은 경우는 없어서, 튜플 비교는 두 작업을 직접 비교하려고 하지 않습니다.

비교할 수 없는 작업의 문제에 대한 또 다른 해결책은 작업 항목을 무시하고 우선순위 필드만 비교하는 래퍼 클래스를 만드는 것입니다:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

나머지 과제는 계류 중인 작업을 찾고 우선순위를 변경하거나 완전히 제거하는 것과 관련이 있습니다. 작업을 찾는 것은 큐에 있는 항목을 가리키는 딕셔너리를 사용해서 해결할 수 있습니다.

힙 구조 불변성을 깨뜨리기 때문에 항목을 제거하거나 우선순위를 변경하는 것은 더 어렵습니다. 따라서, 가능한 해결책은 항목을 제거된 것으로 표시하고 우선순위가 수정된 새 항목을 추가하는 것입니다:

```
pq = []                         # 1힙에 배치된 항목의 리스트
entry_finder = {}               # 작업에서 항목으로의 매핑
REMOVED = '<removed-task>'      # 삭제된 작업을 위한 자리 표시기
counter = itertools.count()     # 고유한 시퀀스 카운트

def add_task(task, priority=0):
    '새 작업을 추가하거나 기존 작업의 우선순위를 갱신합니다'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    '기존 작업을 REMOVED로 표시합니다.  발견되지 않으면 KeyError를 발생시킵니다.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    '가장 낮은 우선순위 작업을 삭제하고 반환합니다. 비어있으면 KeyError를 발생시킵니다.
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

8.6.3 이론

힙은 0부터 요소를 셀 때, 모든 k에 대해 `a[k] <= a[2*k+1]`와 `a[k] <= a[2*k+2]`가 유지되는 배열입니다. 비교를 위해, 존재하지 않는 요소는 무한인 것으로 간주합니다. 힙의 흥미로운 특성은 `a[0]`이 항상 가장 작은 요소라는 것입니다.

위의 특이한 불변성은 토너먼트를 위한 효율적인 메모리 표현을 위한 것입니다. 아래 숫자는 `a[k]`가 아니라 k입니다:

```
0

              1                                 2

      3                 4                 5                 6
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

위의 트리에서, 각 셀 *k*는 `2*k+1`과 `2*k+2`위에 있습니다. 우리가 스포츠에서 볼 수 있는 일반적인 이진 토너먼트에서, 각 셀은 아래에 있는 두 개의 셀의 승자가 되며, 트리 아래로 승자를 추적하여 모든 상대를 볼 수 있습니다. 그러나, 이러한 토너먼트의 많은 컴퓨터 응용에서 승자의 이력을 추적할 필요는 없습니다. 메모리 효율성을 높이기 위해, 승자가 승격될 때, 하위 수준에서 다른 것으로 대체하려고 시도합니다. 규칙은 셀과 셀 아래의 두 셀이 세 개의 다른 항목을 포함하지만, 위의 셀은 아래의 두 셀에 "이기는" 것입니다.

이 힙 불변성이 항상 보호된다면, 인덱스 0은 분명히 최종 승자입니다. 이것을 제거하고 "다음" 승자를 찾는 가장 간단한 알고리즘 적인 방법은 어떤 패자(위의 도표에서 셀 30이라고 합시다)를 0 위치로 옮기고, 불변성을 다시 만족할 때까지 값을 교환하면서 이 새로운 0을 트리 아래로 침투시키는 것입니다. 이것은 트리의 총 항목 수에 대해 분명히 로그 함수적(logarithmic) 입니다. 모든 항목에 대해 반복하면, $O(n \log n)$ 정렬을 얻게 됩니다.

이 정렬의 멋진 기능은 삽입된 항목이 추출한 마지막 0번째 요소보다 "더 나은" 항목이 아니라면, 정렬이 진행되는 동안 새 항목을 효율적으로 삽입 할 수 있다는 것입니다. 이는 트리가 들어오는 모든 이벤트를 담고, "승리" 조건이 가장 작은 예약 시간을 의미하는 시뮬레이션 문맥에서 특히 유용합니다. 이벤트가 실행을 위해 다른 이벤트를 예약하면, 이들은 미래에 예약되어서, 쉽게 힙에 들어갈 수 있습니다. 따라서, 힙은 스케줄러를 구현하기에 좋은 구조입니다 (이것이 제가 MIDI 시퀀서에 사용한 것입니다 :-).

스케줄러를 구현하기 위한 다양한 구조가 광범위하게 연구되었으며, 힙은 합리적으로 빠르며, 속도가 거의 일정합니다, 최악의 경우는 평균 경우와 크게 다르지 않기 때문에 스케줄러에 좋습니다. 하지만, 최악의 경우는 끔찍할 수 있습니다만, 전반적으로 더 효율적인 다른 표현이 있기는 합니다.

힙은 큰 디스크 정렬에도 매우 유용합니다. 여러분은 아마도 큰 정렬은 "런(runs)" (크기가 일반적으로 CPU 메모리 크기와 관련된 사전 정렬된 시퀀스)을 생성한 후에 이러한 런들에 대한 병합 패스가 따라옴을 의미하며, 이러한 병합은 종종 매우 영리하게 조직됨을 알고 있을 겁니다[1]. 초기 정렬이 가능한 한 가장 긴 런을 생성하는 것이 매우 중요합니다. 토너먼트는 이를 달성하기 위한 좋은 방법입니다. 토너먼트를 개최하는데 사용할 수 있는 모든 메모리를 사용하여 현재 런에 맞는 항목들을 교체하고 침투시키면, 무작위 입력을 위한 메모리 크기의 두 배인 런을 생성하게 되고, 적당히 정렬된 입력에 대해서는 더 좋습니다.

더 나아가, 또한 디스크에 0번째 항목을 출력하고 현재 토너먼트에 맞지 않는 입력을 받으면 (그 값이 마지막 출력값을 "이기기" 때문에), 힙에 넣을 수 없어서 힙의 크기가 줄어듭니다. 해제된 메모리는 두 번째 힙을 점진적으로 구축하는데 즉시 영리하게 재사용될 수 있고, 두 번째 힙이 자라는 속도는 첫 번째 힙이 줄어드는 것과 같습니다. 첫 번째 힙이 완전히 사라지면, 힙을 전환하고 새 런을 시작합니다. 영리하고 매우 효과적입니다!

한마디로, 힙은 알아두어야 할 유용한 메모리 구조입니다. 저는 몇 가지 응용 프로그램에서 사용하며, '힙' 모듈을 근처에 두는 것이 좋다고 생각합니다. :-)

8.7 `bisect` — 배열 이진 분할 알고리즘

소스 코드: Lib/bisect.py

이 모듈은 정렬된 리스트를 삽입 후에 다시 정렬할 필요 없도록 관리할 수 있도록 지원합니다. 값비싼 비교 연산이 포함된 항목의 긴 리스트의 경우, 이는 선형 검색이나 빈번한 재정렬에 비해 개선된 것입니다.

이 모듈은 기본적인 이진 분할 알고리즘을 사용하기 때문에 *`bisect`*라고 불립니다. 특정 값을 검색하는 다른 이진 분할 도구와 달리, 이 모듈의 함수는 삽입 지점을 찾도록 설계되었습니다. 따라서, 함수는 값을 찾았는지 판단하기 위해 `__eq__()` 메서드를 호출하지 않습니다. 대신, 함수는 `__lt__()` 메서드만 호출하고 배열에 있는 값 사이의 삽입 지점을 반환합니다.

[1] 요즘 최신 디스크 밸런싱 알고리즘은 영리하다기보다는 성가시며, 이는 디스크의 탐색(seek) 기능으로 인한 결과입니다. 큰 테이프 드라이브와 같이 탐색할 수 없는 장치에서는, 이야기가 상당히 달랐으며, 각 테이프 움직임이 가장 효과적일 수 있도록 (즉, 병합을 "진행하는데" 최대한 참여할 수 있도록) (일찌감치) 계획하기 위해 아주 영리해야 했습니다. 일부 테이프는 반대 방향으로 읽을 수도 있었으며, 이것은 되감기 시간을 피하는 데 사용되기도 했습니다. 저를 믿으십시오, 진짜 훌륭한 테이프 정렬은 장관이었습니다! 언제나, 정렬은 항상 위대한 예술이었습니다! :-)

> **참고**
>
> The functions in this module are not thread-safe. If multiple threads concurrently use `bisect` functions on the same sequence, this may result in undefined behaviour. Likewise, if the provided sequence is mutated by a different thread while a `bisect` function is operating on it, the result is undefined. For example, using `insort_left()` on the same list from multiple threads may result in the list becoming unsorted.

다음과 같은 함수가 제공됩니다:

bisect.**bisect_left**(*a, x, lo=0, hi=len(a), *, key=None*)

정렬된 순서를 유지하도록 *a*에 *x*를 삽입할 위치를 찾습니다. 매개 변수 *lo* 와 *hi*는 고려해야 할 리스트의 부분집합을 지정하는 데 사용될 수 있습니다; 기본적으로 전체 리스트가 사용됩니다. *x*가 *a*에 이미 있으면, 삽입 위치는 기존 항목 앞(왼쪽)이 됩니다. 반환 값은 *a*가 이미 정렬되었다고 가정할 때 `list.insert()`의 첫 번째 매개 변수로 사용하기에 적합합니다.

반환된 삽입 위치 *ip*는 배열 *a*를 두 개의 조각으로 분할하여, 왼쪽에 대해서는 `all(elem < x for elem in a[lo : ip])`이 참이고, 오른쪽에 대해서는 `all(elem >= x for elem in a[ip : hi])`이 참이 되도록 만듭니다.

key specifies a *key function* of one argument that is used to extract a comparison key from each element in the array. To support searching complex records, the key function is not applied to the *x* value.

If *key* is `None`, the elements are compared directly and no key function is called.

버전 3.10에서 변경: *key* 매개 변수를 추가했습니다.

bisect.**bisect_right**(*a, x, lo=0, hi=len(a), *, key=None*)

bisect.**bisect**(*a, x, lo=0, hi=len(a), *, key=None*)

`bisect_left()`와 비슷하지만, *a*에 있는 *x*의 기존 항목 뒤(오른쪽)에 오는 삽입 위치를 반환합니다.

반환된 삽입 위치 *ip*는 배열 *a*를 두 개의 조각으로 분할하여, 왼쪽에 대해서는 `all(elem <= x for elem in a[lo : ip])`이 참이고, 오른쪽에 대해서는 `all(elem > x for elem in a[ip : hi])`이 참이 되도록 만듭니다.

버전 3.10에서 변경: *key* 매개 변수를 추가했습니다.

bisect.**insort_left**(*a, x, lo=0, hi=len(a), *, key=None*)

Insert *x* in *a* in sorted order.

This function first runs `bisect_left()` to locate an insertion point. Next, it runs the `insert()` method on *a* to insert *x* at the appropriate position to maintain sort order.

To support inserting records in a table, the *key* function (if any) is applied to *x* for the search step but not for the insertion step.

Keep in mind that the $O(\log n)$ search is dominated by the slow $O(n)$ insertion step.

버전 3.10에서 변경: *key* 매개 변수를 추가했습니다.

bisect.**insort_right**(*a, x, lo=0, hi=len(a), *, key=None*)

bisect.**insort**(*a, x, lo=0, hi=len(a), *, key=None*)

`insort_left()`와 비슷하지만, *a*에 *x*를 *x*의 기존 항목 다음에 삽입합니다.

This function first runs `bisect_right()` to locate an insertion point. Next, it runs the `insert()` method on *a* to insert *x* at the appropriate position to maintain sort order.

To support inserting records in a table, the *key* function (if any) is applied to *x* for the search step but not for the insertion step.

Keep in mind that the $O(\log n)$ search is dominated by the slow $O(n)$ insertion step.

버전 3.10에서 변경: *key* 매개 변수를 추가했습니다.

8.7.1 Performance Notes

When writing time sensitive code using *bisect*() and *insort*(), keep these thoughts in mind:

- Bisection is effective for searching ranges of values. For locating specific values, dictionaries are more performant.
- The *insort*() functions are $O(n)$ because the logarithmic search step is dominated by the linear time insertion step.
- The search functions are stateless and discard key function results after they are used. Consequently, if the search functions are used in a loop, the key function may be called again and again on the same array elements. If the key function isn' t fast, consider wrapping it with `functools.cache()` to avoid duplicate computations. Alternatively, consider searching an array of precomputed keys to locate the insertion point (as shown in the examples section below).

더 보기

- Sorted Collections is a high performance module that uses *bisect* to managed sorted collections of data.
- bisect를 사용하여 직접적인 검색 메서드와 키 함수 지원을 포함하는 완전한 기능을 갖춘 컬렉션 클래스를 만드는 SortedCollection recipe. 검색 중에 불필요한 키 함수 호출을 피하고자 키는 미리 계산됩니다.

8.7.2 정렬된 리스트 검색하기

위의 *bisect functions*는 삽입 위치를 찾는 데 유용하지만, 일반적인 검색 작업에 사용하기가 까다롭거나 어색할 수 있습니다. 다음 다섯 함수는 정렬된 리스트에 대한 표준 조회로 변환하는 방법을 보여줍니다:

```
def index(a, x):
    'x 와 정확히 같은 가장 왼쪽의 값을 찾습니다'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'x보다 작은 가장 오른쪽 값을 찾습니다'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'x보다 작거나 같은 가장 오른쪽 값을 찾습니다'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'x보다 큰 가장 왼쪽 값을 찾습니다'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'x보다 크거나 같은 가장 왼쪽 항목을 찾습니다'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.7.3 예제

bisect() 함수는 숫자 테이블 조회에 유용할 수 있습니다. 이 예제는 *bisect()*를 사용하여 (가령) 시험 점수에 대한 문자 등급을 조회하는데, 정렬된 숫자 경계점 집합에 기반합니다: 90 이상은 'A' , 80에서 89는 'B' 등입니다:

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

The *bisect()* and *insort()* functions also work with lists of tuples. The *key* argument can serve to extract the field used for ordering records in a table:

```
>>> from collections import namedtuple
>>> from operator import attrgetter
>>> from bisect import bisect, insort
>>> from pprint import pprint

>>> Movie = namedtuple('Movie', ('name', 'released', 'director'))

>>> movies = [
...     Movie('Jaws', 1975, 'Spielberg'),
...     Movie('Titanic', 1997, 'Cameron'),
...     Movie('The Birds', 1963, 'Hitchcock'),
...     Movie('Aliens', 1986, 'Cameron')
... ]

>>> # 1960년 이후 개봉한 최초의 영화를 찾습니다
>>> by_year = attrgetter('released')
>>> movies.sort(key=by_year)
>>> movies[bisect(movies, 1960, key=by_year)]
Movie(name='The Birds', released=1963, director='Hitchcock')

>>> # 정렬 순서를 유지하면서 영화를 삽입합니다
>>> romance = Movie('Love Story', 1970, 'Hiller')
>>> insort(movies, romance, key=by_year)
>>> pprint(movies)
[Movie(name='The Birds', released=1963, director='Hitchcock'),
 Movie(name='Love Story', released=1970, director='Hiller'),
 Movie(name='Jaws', released=1975, director='Spielberg'),
 Movie(name='Aliens', released=1986, director='Cameron'),
 Movie(name='Titanic', released=1997, director='Cameron')]
```

키 함수가 비싸면, 미리 계산된 키 목록을 검색하여 레코드의 인덱스를 찾으면 반복되는 함수 호출을 피할 수 있습니다:

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])        # 또는 operator.itemgetter(1) 를 사용하세요.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> keys = [r[1] for r in data]         # 미리 계산된 키 리스트.
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.8 array — 효율적인 숫자 배열

이 모듈은 문자, 정수, 부동 소수점 숫자와 같은 기본적인 값의 배열을 간결하게 표현할 수 있는 객체 형을 정의합니다. 배열은 시퀀스 형이며 리스트와 매우 비슷하게 행동합니다만, 그곳에 저장되는 객체의 형이 제약된다는 점이 다릅니다. 형은 객체 생성 시에 단일 문자인 형 코드(*type code*)를 사용하여 지정됩니다. 다음 형 코드가 정의됩니다:

형 코드	C 형	파이썬 형	최소 크기(바이트)	노트
'b'	signed char	int	1	
'B'	unsigned char	int	1	
'u'	wchar_t	유니코드 문자	2	(1)
'w'	Py_UCS4	유니코드 문자	4	(2)
'h'	signed short	int	2	
'H'	unsigned short	int	2	
'i'	signed int	int	2	
'I'	unsigned int	int	2	
'l'	signed long	int	4	
'L'	unsigned long	int	4	
'q'	signed long long	int	8	
'Q'	unsigned long long	int	8	
'f'	float	float	4	
'd'	double	float	8	

노트:

(1) 플랫폼에 따라 16비트 또는 32비트 일 수 있습니다.

버전 3.9에서 변경: `array('u')`는 이제 폐지된 `Py_UNICODE` 대신 `wchar_t`를 C형으로 사용합니다. `Py_UNICODE`는 파이썬 3.3부터 `wchar_t`의 별칭이라서 이 변경은 동작에 영향을 미치지 않습니다.

Deprecated since version 3.3, will be removed in version 3.16: Please migrate to `'w'` typecode.

(2) Added in version 3.13.

값의 실제 표현은 기계 아키텍처에 의해(엄격히 말하자면 C 구현에 의해) 결정됩니다. 실제 크기는 *array.itemsize* 어트리뷰트를 통해 액세스할 수 있습니다.

모듈은 다음 항목을 정의합니다:

`array.`**`typecodes`**

사용 가능한 모든 형 코드가 있는 문자열.

모듈은 다음 형을 정의합니다:

class array.**array**(*typecode*[, *initializer*])

항목이 *typecode*에 의해 제한되는 새 배열, 선택적인 *initializer* 값으로 초기화되는데, `bytes` 나 `bytearray` 객체, 유니코드 문자열 또는 적절한 형의 요소에 대한 이터러블이어야 합니다.

`bytes` 나 `bytearray` 객체가 주어지면, initializer는 새 배열의 `frombytes()` 메서드로 전달됩니다; 유니코드 문자열이 주어지면, initializer는 `fromunicode()` 메서드로 전달됩니다; 그렇지 않으면, initializer의 이터레이터가 `extend()` 메서드에 전달되어 배열에 초기 항목들을 추가합니다.

배열 객체는 인덱싱, 슬라이싱, 이어붙이기 및 곱셈과 같은 일반적인 시퀀스 연산을 지원합니다. 슬라이스 대입을 사용할 때, 대입되는 값은 같은 형 코드의 배열 객체여야 합니다; 다른 모든 경우에는, `TypeError`가 발생합니다. 배열 객체는 버퍼 인터페이스도 구현하며, 바이트열류 객체가 지원되는 곳이면 어디에서나 사용될 수 있습니다.

`typecode`, `initializer` 인자로 감사 이벤트(*auditing event*) `array.__new__`를 발생시킵니다.

typecode

배열을 만드는 데 사용된 typecode 문자.

itemsize

내부 표현에서 하나의 배열 항목의 길이 (바이트).

append(*x*)

배열의 끝에 값 *x*로 새 항목을 추가합니다.

buffer_info()

배열의 내용을 담는 데 사용된 버퍼의 현재 메모리 주소와 요소의 수로 표현한 길이를 제공하는 튜플 `(address, length)`를 반환합니다. 바이트 단위의 메모리 버퍼 크기는 `array.buffer_info()[1] * array.itemsize`로 계산할 수 있습니다. 이것은 특정 `ioctl()` 연산과 같은 메모리 주소가 필요한 저수준(그리고 근본적으로 안전하지 않은) I/O 인터페이스로 작업할 때 간혹 유용합니다. 반환된 숫자는 배열이 존재하고 길이 변경 연산이 적용되지 않는 한 유효합니다.

> **참고**
>
> C나 C++로 작성된 코드(이 정보를 효율적으로 사용하는 유일한 방법)에서 배열 객체를 사용할 때, 배열 객체가 지원하는 버퍼 인터페이스를 사용하는 것이 좋습니다. 이 메서드는 이전 버전과의 호환성을 위해 유지되며 새 코드에서는 사용하지 않아야 합니다. 버퍼 인터페이스는 bufferobjects에 설명되어 있습니다.

byteswap()

배열의 모든 항목을 "바이트 스와프(byteswap)" 합니다. 1, 2, 4 또는 8바이트 크기의 값에 대해서만 지원됩니다; 다른 형의 값이면 `RuntimeError`가 발생합니다. 바이트 순서가 다른 컴퓨터에서 작성된 파일에서 데이터를 읽을 때 유용합니다.

count(*x*)

배열 내에서 *x*가 등장하는 횟수를 반환합니다.

extend(*iterable*)

*iterable*의 항목을 배열의 끝에 추가합니다. *iterable*이 다른 배열이면, 정확히 같은 형 코드를 가져야 합니다; 그렇지 않으면, `TypeError`가 발생합니다. *iterable*이 배열이 아니면, 이터러블이어야 하며 요소는 배열에 추가할 올바른 형이어야 합니다.

frombytes(*buffer*)

바이트열류 객체에서 항목을 추가합니다. 내용을 기곗값(machine value)의 배열로 해석합니다 (마치 `fromfile()` 메서드를 사용하여 파일에서 읽은 것처럼).

Added in version 3.2: `fromstring()`은 명확하게 하려고 `frombytes()`로 이름을 바꿨습니다.

fromfile(*f*, *n*)

파일 객체 *f*에서 (기곗값으로) *n* 항목을 읽고 배열의 끝에 추가합니다. *n* 미만의 항목만 사용할 수 있으면 `EOFError`가 발생하지만, 사용 가능한 항목은 여전히 배열에 삽입됩니다.

fromlist(*list*)

리스트에서 항목을 추가합니다. 이것은 형 에러가 있으면 배열이 변경되지 않는다는 점만 제외하면 `for x in list: a.append(x)` 와 동등합니다.

fromunicode(*s*)

주어진 유니코드 문자열의 데이터로 이 배열을 확장합니다. 배열은 형 코드가 `'u'` 나 `'w'` 이어야 합니다; 그렇지 않으면 `ValueError`가 발생합니다. 다른 형의 배열에 유니코드 데이터를 추가하려면 `array.frombytes(unicodestring.encode(enc))` 를 사용하십시오.

index(*x*[, *start*[, *stop*]])

Return the smallest *i* such that *i* is the index of the first occurrence of *x* in the array. The optional arguments *start* and *stop* can be specified to search for *x* within a subsection of the array. Raise `ValueError` if *x* is not found.

버전 3.10에서 변경: Added optional *start* and *stop* parameters.

insert(*i*, *x*)

i 위치 앞에 값이 *x*인 새 항목을 배열에 삽입합니다. 음수 값은 배열 끝에 상대적인 값으로 처리됩니다.

pop([*i*])

배열에서 인덱스 *i*에 있는 항목을 제거하고 이를 반환합니다. 선택적 인자의 기본값은 `-1` 이므로, 기본적으로 마지막 항목이 제거되고 반환됩니다.

remove(*x*)

배열에서 첫 번째 *x*를 제거합니다.

clear()

배열에서 모든 항목을 제거합니다.

Added in version 3.13.

reverse()

배열의 항목 순서를 뒤집습니다.

tobytes()

배열을 기곗값 배열로 변환하고 바이트열 표현(`tofile()` 메서드로 파일에 기록될 바이트 시퀀스와 같습니다)을 반환합니다.

Added in version 3.2: `tostring()` 은 명확하게 하려고 `tobytes()` 로 이름을 바꿨습니다.

tofile(*f*)

모든 항목을 (기곗값으로) 파일 객체 *f*에 씁니다.

tolist()

배열을 같은 항목이 있는 일반 리스트로 변환합니다.

tounicode()

배열을 유니코드 문자열로 변환합니다. 배열은 형 코드가 `'u'` 나 `'w'` 이어야 합니다; 그렇지 않으면 `ValueError`가 발생합니다. 다른 형의 배열로부터 유니코드 문자열을 얻으려면 `array.tobytes().decode(enc)` 를 사용하십시오.

배열 객체의 문자열 표현은 `array(typecode, initializer)` 형태를 취합니다. 배열이 비어 있으면 *initializer*가 생략되고, 그렇지 않으면 *typecode*가 `'u'` 나 `'w'` 인 경우 유니코드 문자열이 되고, 그렇지 않으면 숫자 리스트가 됩니다. 문자열 표현은 `eval()`을 사용하여 같은 형과 값을 갖는 배열로 다시 변환될 수 있음이 보장됩니다. 단 `from array import array`를 사용하여 `array` 클래스를 임포트 한다고 가정합니다. 변수 `inf` 와 `nan` 도 해당 부동소수점 값을 포함하는 경우 정의해야합니다. 예:

```
array('l')
array('w', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14, -inf, nan])
```

더 보기

모듈 *struct*
이질적인(heterogeneous) 바이너리 데이터의 패킹과 언 패킹.

NumPy
NumPy 패키지는 다른 배열형을 정의합니다.

8.9 weakref — 약한 참조

소스 코드: Lib/weakref.py

---

weakref 모듈은 파이썬 프로그래머가 객체에 대한 약한 참조 (*weak references*)를 만들 수 있도록 합니다.

이하에서, 용어 참조대상(*referent*)은 약한 참조로 참조되는 객체를 의미합니다.

객체에 대한 약한 참조만으로는 객체를 살아있게 유지할 수 없습니다: 참조대상에 대한 유일한 남은 참조가 약한 참조면, 가비지 수거는 자유롭게 참조대상을 파괴하고 메모리를 다른 용도로 재사용할 수 있습니다. 그러나 객체가 실제로 파괴될 때까지 약한 참조는 강한 참조가 없어도 객체를 반환할 수 있습니다.

약한 참조의 주요 용도는 큰 객체를 보유하는 캐시나 매핑을 구현하는 것입니다. 큰 객체는 캐시나 매핑에 등장한다는 이유만으로 살아 있지 않아야 합니다.

예를 들어, 큰 바이너리 이미지 객체가 여러 개 있을 때, 이름을 각 객체와 연관 지을 수 있습니다. 파이썬 딕셔너리를 사용하여 이름을 이미지에 매핑하거나 이미지를 이름에 매핑하면, 이미지 객체는 딕셔너리에 값이나 키로 등장하기 때문에 계속 살아있게 됩니다. *weakref* 모듈에서 제공하는 *WeakKeyDictionary*와 *WeakValueDictionary* 클래스는 대안이며, 약한 참조를 사용하여 매핑을 구축하기 때문에 매핑 객체에 등장한다는 이유만으로 객체를 살려두지 않습니다. 예를 들어 이미지 객체가 *WeakValueDictionary* 의 값이면, 해당 이미지 객체에 대한 마지막 남은 참조가 약한 매핑에 들어 있는 약한 참조이면, 가비지 수거는 객체를 회수할 수 있으며, 약한 매핑의 해당 항목은 간단히 삭제됩니다.

*WeakKeyDictionary*와 *WeakValueDictionary*는 구현에 약한 참조를 사용하여, 가비지 수거에서 키나 값이 회수될 때, 약한 딕셔너리에 알리는 약한 참조에 대한 콜백 함수를 설정합니다. *WeakSet*은 *set* 인터페이스를 구현하지만, *WeakKeyDictionary* 처럼 원소에 대한 약한 참조를 유지합니다.

*finalize*는 객체가 가비지 수거될 때 호출될 정리 함수를 등록하는 간단한 방법을 제공합니다. 이 모듈은 원시 약한 참조에 콜백 함수를 설정하는 것보다 사용하기가 더 쉽습니다. 모듈은 객체가 수거될 때까지 자동으로 파이널라이저가 활성 상태로 유지되도록 하기 때문입니다.

대부분의 프로그램은 이러한 약한 컨테이너형이나 *finalize*를 사용하는 것으로 충분합니다 – 일반적으로 여러분 스스로 약한 참조를 직접 만들 필요는 없습니다. 저수준 장치는 고급 용도를 위해 *weakref* 모듈이 노출합니다.

모든 객체를 약하게 참조할 수 있는 것은 아닙니다. 약한 참조를 지원하는 객체에는 클래스 인스턴스, 파이썬으로 작성된 함수 (C로 작성된 함수는 아닙니다), 인스턴스 메서드, 집합, 불변 집합(frozenset), 일부 파일 객체, 제너레이터, 형 객체, 소켓, 배열, 데크(deque), 정규식 패턴 객체 및 코드 객체가 포함됩니다.

버전 3.2에서 변경: thread.lock, threading.Lock 및 코드 객체에 대한 지원이 추가되었습니다.

*list*와 *dict*와 같은 여러 내장형은 약한 참조를 직접 지원하지 않지만, 서브 클래싱을 통해 지원을 추가할 수 있습니다:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # 이 객체는 약한 참조할 수 있습니다
```

*tuple*과 *int*와 같은 다른 내장형은 서브 클래싱 될 때도 약한 참조를 지원하지 않습니다.

확장형은 쉽게 약한 참조를 지원하도록 만들 수 있습니다; weakref-support을 참조하십시오.

When __slots__ are defined for a given type, weak reference support is disabled unless a '__weakref__' string is also present in the sequence of strings in the __slots__ declaration. See __slots__ documentation for details.

class weakref.**ref**(*object*[, *callback*])

*object*에 대한 약한 참조를 반환합니다. 참조대상이 아직 살아있으면 참조 객체를 호출하여 원래 객체를 얻을 수 있습니다. 참조대상이 더는 존재하지 않으면 참조 객체를 호출할 때 *None*이 반환됩니다. *None*이 아닌 *callback*이 제공되고, 반환된 약한 참조 객체가 여전히 살아있으면, 객체가 파이널라이즈 되려고 할 때 콜백이 호출됩니다; 약한 참조 객체는 콜백에 유일한 매개 변수로 전달됩니다; 참조대상은 더는 사용할 수 없습니다.

같은 객체에 대해 여러 개의 약한 참조를 구성할 수 있습니다. 각 약한 참조에 등록된 콜백은 가장 최근에 등록된 콜백에서 가장 오래전에 등록된 콜백 순으로 호출됩니다.

콜백에 의해 발생한 예외는 표준 에러 출력에 표시되지만, 전파될 수는 없습니다; 객체의 __del__() 메서드에서 발생한 예외와 정확히 같은 방식으로 처리됩니다.

약한 참조는 *object*가 해시 가능하면 해시 가능입니다. *object*가 삭제된 후에도 해시값을 유지합니다. 오직 *object*가 삭제된 후에 *hash()*를 처음 호출하면, 호출은 *TypeError*를 발생시킵니다.

약한 참조는 동등 검사를 지원하지만, 순서는 지원하지 않습니다. 참조대상이 여전히 살아 있다면, 두 참조는 (*callback*과 관계없이) 참조대상과 같은 동등 관계를 갖습니다. 참조대상이 삭제되었으면, 참조 객체가 같은 객체일 때만 참조가 같습니다.

이것은 팩토리 함수가 아니라 서브 클래싱 할 수 있는 형입니다.

__callback__

이 읽기 전용 어트리뷰트는 현재 약한 참조와 연관된 콜백을 반환합니다. 콜백이 없거나 약한 참조의 참조대상이 더는 살아있지 않으면 이 어트리뷰트의 값은 None이 됩니다.

버전 3.4에서 변경: *__callback__* 어트리뷰트를 추가했습니다.

weakref.**proxy**(*object*[, *callback*])

*object*로의 약한 참조를 사용하는 프락시를 반환합니다. 이는 약한 참조 객체에서 사용되는 명시적 역참조를 요구하는 대신 대부분의 문맥에서 프락시의 사용을 지원합니다. 반환된 객체는 *object*가 콜러블인지에 따라 ProxyType이나 CallableProxyType 형을 갖습니다. 프락시 객체는 참조대상에 관계없이 해시 가능하지 않습니다; 이것은 그들의 근본적인 가변 특성과 관련된 여러 가지 문제를 피하고, 딕셔너리 키로 사용하는 것을 방지합니다. *callback*은 *ref()* 함수의 같은 이름의 매개 변수와 같습니다.

Accessing an attribute of the proxy object after the referent is garbage collected raises *ReferenceError*.

버전 3.8에서 변경: 행렬 곱셈 연산자 @와 @=을 포함하도록 프락시 객체에 대한 연산자 지원을 확장했습니다.

weakref.**getweakrefcount**(*object*)

*object*를 참조하는 약한 참조와 프락시의 개수를 반환합니다.

weakref.**getweakrefs**(*object*)

*object*를 참조하는 모든 약한 참조와 프락시 객체의 리스트를 반환합니다.

class weakref.**WeakKeyDictionary**([*dict*])

키를 약하게 참조하는 매핑 클래스. 더는 키에 대한 강한 참조가 없으면 딕셔너리의 항목이 삭제됩니다. 이것은 응용 프로그램의 다른 부분이 소유한 객체에 어트리뷰트를 추가하지 않고도 추가 데이터를 연결하는 데 사용될 수 있습니다. 어트리뷰트 액세스를 재정의하는 객체에 특히 유용할 수 있습니다.

Note that when a key with equal value to an existing key (but not equal identity) is inserted into the dictionary, it replaces the value but does not replace the existing key. Due to this, when the reference to the original key is deleted, it also deletes the entry in the dictionary:

```
>>> class T(str): pass
...
>>> k1, k2 = T(), T()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> d = weakref.WeakKeyDictionary()
>>> d[k1] = 1    # d = {k1: 1}
>>> d[k2] = 2    # d = {k1: 2}
>>> del k1       # d = {}
```

A workaround would be to remove the key prior to reassignment:

```
>>> class T(str): pass
...
>>> k1, k2 = T(), T()
>>> d = weakref.WeakKeyDictionary()
>>> d[k1] = 1    # d = {k1: 1}
>>> del d[k1]
>>> d[k2] = 2    # d = {k2: 2}
>>> del k1       # d = {k2: 2}
```

버전 3.9에서 변경: **PEP 584**에 지정된 대로, | 와 |= 연산자에 대한 지원이 추가되었습니다.

WeakKeyDictionary 객체에는 내부 참조를 직접 노출하는 추가 메서드가 있습니다. 참조는 사용되는 시점에 "살아있다고" 보장되지 않아서, 참조를 호출한 결과를 사용하기 전에 확인해야 합니다. 가비지 수거기가 키를 필요 이상으로 길게 유지하도록 하는 참조를 만들지 않도록 하는 데 사용할 수 있습니다.

WeakKeyDictionary.**keyrefs**()

키에 대한 약한 참조의 이터러블을 반환합니다.

class weakref.**WeakValueDictionary**([*dict*])

값을 약하게 참조하는 매핑 클래스. 값에 대한 강한 참조가 더는 존재하지 않을 때 딕셔너리의 항목이 삭제됩니다.

버전 3.9에서 변경: **PEP 584**에 지정된 대로, | 와 |= 연산자에 대한 지원이 추가되었습니다.

WeakValueDictionary 객체에는 *WeakKeyDictionary.keyrefs* 메서드와 같은 문제가 있는 추가 메서드가 있습니다.

WeakValueDictionary.**valuerefs**()

값에 대한 약한 참조의 이터러블을 반환합니다.

class weakref.**WeakSet**([*elements*])

원소에 대한 약한 참조를 유지하는 집합 클래스. 원소에 대한 강한 참조가 더는 존재하지 않을 때 원소가 삭제됩니다.

class weakref.**WeakMethod**(*method*[, *callback*])

연결된 메서드(즉, 클래스에 정의되고 인스턴스에서 조회된 메서드)에 대한 약한 참조를 시뮬레이트하는 사용자 지정 *ref* 서브 클래스. 연결된 메서드는 일시적이므로, 표준 약한 참조는 유지할 수 없습니다. *WeakMethod*에는 객체나 원래 함수가 죽을 때까지 연결된 메서드를 다시 만드는 특별한 코드가 있습니다:

```
>>> class C:
...     def method(self):
...         print("method called!")
...
>>> c = C()
>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
<bound method C.method of <__main__.C object at 0x7fc859830220>>
>>> r()()
method called!
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

callback is the same as the parameter of the same name to the `ref()` function.

Added in version 3.4.

class weakref.**finalize**(*obj*, *func*, */*, **args*, ***kwargs*)

*obj*가 가비지 수거될 때 호출되는 콜러블 파이널라이저 객체를 반환합니다. 일반적인 약한 참조와 달리, 파이널라이저는 참조 객체가 수집될 때까지 항상 생존하므로, 수명 주기 관리가 크게 간소화됩니다.

파이널라이저는 (명시적으로나 가비지 수거에서) 호출될 때까지 살아있다고 간주하며, 그 후 죽습니다. 살아있는 파이널라이저를 호출하면 `func(*arg, **kwargs)`를 평가한 결과가 반환되고, 죽은 파이널라이저를 호출하면 `None`이 반환됩니다.

가비지 수거 중에 파이널라이저 콜백에서 발생한 예외는 표준 에러 출력에 표시되지만, 전파할 수는 없습니다. 객체의 `__del__()` 메서드나 약한 참조의 콜백에서 발생하는 예외와 같은 방식으로 처리됩니다.

프로그램이 종료될 때, `atexit` 어트리뷰트가 거짓으로 설정되지 않은 한 각 남은 살아있는 파이널라이저가 호출됩니다. 만들어진 순서와 반대 순서로 호출됩니다.

모듈 전역이 `None`으로 교체된 경우 인터프리터 종료의 후반 동안에는 파이널라이저가 콜백을 호출하지 않습니다.

__call__()

*self*가 살아 있으면 이를 죽은 것으로 표시하고 `func(*args, **kwargs)` 호출 결과를 반환합니다. *self*가 죽었으면 `None`을 반환합니다.

detach()

*self*가 살아 있으면 이를 죽은 것으로 표시하고 튜플 `(obj, func, args, kwargs)`를 반환합니다. *self*가 죽었으면 `None`을 반환합니다.

peek()

*self*가 살아 있으면 튜플 `(obj, func, args, kwargs)`를 반환합니다. *self*가 죽었으면 `None`을 반환합니다.

alive

파이널라이저가 살아 있으면 참이고, 그렇지 않으면 거짓인 프로퍼티.

atexit

기본값이 참인, 쓰기 가능한 불리언 프로퍼티. 프로그램이 종료할 때, `atexit`가 참인 남은 모든 살아있는 파이널라이저를 호출합니다. 그것들은 만들어진 순서와 반대 순서로 호출됩니다.

> **참고**
>
> *func*, *args* 및 *kwargs*가 직접이나 간접적으로 *obj*에 대한 참조를 소유하지 않는 것이 중요합니다. 그렇지 않으면 *obj*는 가비지 수거되지 않습니다. 특히, *func*는 *obj*의 연결된 메서드가 아니어야 합니다.

Added in version 3.4.

weakref.**ReferenceType**

약한 참조 객체의 형 객체.

weakref.**ProxyType**

콜러블이 아닌 객체의 프락시를 위한 형 객체.

weakref.**CallableProxyType**

콜러블 객체의 프락시를 위한 형 객체.

weakref.**ProxyTypes**

프락시의 모든 형 객체를 포함하는 시퀀스. 이것은 두 프락시 형 모두의 이름 지정에 의존하지 않고 객체가 프락시인지 검사하기 더 쉽게 만들 수 있습니다.

더 보기

PEP 205 - 약한 참조
이전 구현에 대한 링크와 다른 언어의 유사한 기능에 대한 정보를 포함하는, 이 기능에 대한 제안과 근거.

8.9.1 약한 참조 객체

약한 참조 객체에는 *ref.__callback__* 외에 메서드와 어트리뷰트가 없습니다. 약한 참조 객체는 참조대상이 아직 존재한다면 호출함으로써 얻을 수 있도록 합니다:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

참조대상이 더는 존재하지 않을 때, 참조 객체를 호출하면 *None*을 반환합니다:

```
>>> del o, o2
>>> print(r())
None
```

약한 참조 객체가 여전히 살아있는지를 검사하는 것은 `ref() is not None` 표현식을 사용하여 수행해야 합니다. 일반적으로, 참조 객체를 사용할 필요가 있는 응용 프로그램 코드는 다음 패턴을 따라야 합니다:

```
# r은 약한 참조 객체입니다
o = r()
if o is None:
    # 참조대상이 가비지 수거되었습니다
    print("Object has been deallocated; can't frobnicate.")
else:
    print("Object is still live!")
    o.do_something_useful()
```

"생존"에 대해 별도의 검사를 사용하면 스레드 응용 프로그램에서 경쟁 조건이 발생합니다; 약한 참조가 호출되기 전에 다른 스레드가 약한 참조를 무효화 할 수 있습니다; 위에 표시된 관용구는 단일 스레드 응용 프로그램뿐만 아니라 다중 스레드 응용 프로그램에서도 안전합니다.

서브 클래싱을 통해 *ref* 객체의 특수한 버전을 만들 수 있습니다. 이는 *WeakValueDictionary* 구현에 사용되어 매핑의 각 항목에 대한 메모리 오버헤드를 줄입니다. 이는 추가 정보를 참조와 연관시키는 데 가장 유용 할 수 있지만, 참조대상을 꺼내기 위한 호출에 추가 처리를 삽입하는 데 사용될 수도 있습니다.

이 예제는 *ref*의 서브 클래스를 사용하여 객체에 대한 추가 정보를 저장하고 참조대상에 액세스할 때 반환되는 값에 영향을 주는 방법을 보여줍니다:

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, /, **annotations):
        super().__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """참조 대상과 참조가 호출된 횟수를 포함하는 쌍을 반환합니다.
        """
        ob = super().__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.9.2 예

이 간단한 예제는 응용 프로그램이 객체 ID를 사용하여 이전에 본 객체를 조회하는 방법을 보여줍니다. 그런 다음 객체를 강제로 살아있도록 하지 않으면서 다른 자료 구조에서 객체의 ID를 사용할 수 있지만, 살아있다면 객체를 여전히 ID로 조회할 수 있습니다.

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

8.9.3 파이널라이저 객체

*finalize*를 사용해서 얻을 수 있는 주요 이점은 반환된 파이널라이저 객체를 보존할 필요 없이 콜백을 간단하게 등록할 수 있다는 것입니다. 예를 들어

```
>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

파이널라이저를 직접 호출할 수도 있습니다. 그러나 파이널라이저는 콜백을 최대 한 번 호출합니다.

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f()                     # callback not called because finalizer dead
>>> del obj                 # callback not called because finalizer dead
```

detach() 메서드를 사용하여 파이널라이저를 등록 취소할 수 있습니다. 그러면 파이널라이저를 죽이고 만들어질 때 생성자에 전달된 인자가 반환됩니다.

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<...Object object ...>, <function callback ...>, (1, 2), {'z': 3})
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

atexit 어트리뷰트를 *False*로 설정하지 않는 한, 파이널라이저가 살아있다면 프로그램이 종료될 때 호출됩니다. 예를 들어

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

8.9.4 파이널라이저와 `__del__()` 메서드의 비교

인스턴스가 임시 디렉터리를 나타내는 클래스를 만들고 싶다고 가정하십시오. 다음 이벤트 중 첫 번째 것이 발생할 때 디렉터리는 내용과 함께 삭제되어야 합니다:

- 객체가 가비지 수거됩니다,
- 객체의 `remove()` 메서드가 호출됩니다, 또는
- 프로그램이 종료합니다.

다음과 같이 `__del__()` 메서드를 사용하여 클래스를 구현하려고 시도할 수 있습니다:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
        return self.name is None

    def __del__(self):
        self.remove()
```

파이썬 3.4부터, `__del__()` 메서드는 더는 참조 순환이 가비지 수거되는 것을 막지 않으며, 인터프리터 종료 중에 모듈 전역이 더는 *None*으로 강제되지 않습니다. 따라서 이 코드는 CPython에서 아무런 문제 없이 작동해야 합니다.

그러나, `__del__()` 메서드의 처리는 인터프리터의 가비지 수거기 구현에 대한 내부 세부 사항에 의존하기 때문에 구현에 따라 다르기로 악명 높습니다.

더욱 강인한 대안은 객체의 전체 상태에 액세스하기보다 필요한 특정 함수와 객체만 참조하는 파이널라이저를 정의하는 것일 수 있습니다:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.name)

    def remove(self):
        self._finalizer()

    @property
    def removed(self):
        return not self._finalizer.alive
```

이처럼 정의된 파이널라이저는 디렉터리를 적절히 정리하는 데 필요한 세부 사항에 대한 참조만 받습니다. 객체가 가비지 수거되지 않으면 종료 시에 파이널라이저는 여전히 호출됩니다.

약한 참조 기반 파이널라이저의 다른 장점은 제삼자가 정의를 제어하는 클래스에 대해 파이널라이저를 등록하는 데 사용할 수 있다는 것입니다, 가령 모듈이 언로드 될 때 코드 실행하기:

```
import weakref, sys
def unloading_module():
    # 함수 바디에서 모듈 전역으로의 묵시적 참조
weakref.finalize(sys.modules[__name__], unloading_module)
```

> **참고**
>
> 프로그램이 종료될 때 데몬 스레드에서 파이널라이저 객체를 만들면 종료 시에 파이널라이저가 호출되지 않을 가능성이 있습니다. 그러나, 데몬 스레드 *atexit.register()*에서, `try: ... finally: ...`와 `with: ...`는 정리가 발생한다고 보장하지 않습니다.

8.10 types — 동적 형 생성과 내장형 이름

소스 코드: Lib/types.py

이 모듈은 새로운 형의 동적 생성을 지원하는 유틸리티 함수를 정의합니다.

표준 파이썬 인터프리터가 사용하지만, *int*나 *str*처럼 내장으로 노출되지 않는 일부 객체 형의 이름도 정의합니다.

마지막으로, 내장되기에 충분히 기본적이지 않은 몇 가지 추가 형 관련 유틸리티 클래스와 함수를 제공합니다.

8.10.1 동적 형 생성

types.**new_class**(*name*, *bases=()*, *kwds=None*, *exec_body=None*)

적절한 메타 클래스를 사용하여 동적으로 클래스 객체를 만듭니다.

처음 세 개의 인자는 클래스 정의 헤더를 구성하는 요소들입니다: 클래스 이름, 베이스 클래스 (순서대로), 키워드 인자 (가령 `metaclass`).

exec_body 인자는 새로 만들어진 클래스 이름 공간을 채우는 데 사용되는 콜백입니다. 클래스 이름 공간을 유일한 인자로 받아들이고 클래스 내용으로 이름 공간을 직접 갱신해야 합니다. 콜백이 제공되지 않으면, `lambda ns: None`를 전달하는 것과 같은 효과가 있습니다.

Added in version 3.3.

`types.`**`prepare_class`**(*name*, *bases*=(), *kwds*=*None*)

적절한 메타 클래스를 계산하고 클래스 이름 공간을 만듭니다.

인자는 클래스 정의 헤더를 구성하는 요소들입니다: 클래스 이름, 베이스 클래스 (순서대로) 및 키워드 인자 (가령 `metaclass`).

반환 값은 3-튜플입니다: `metaclass, namespace, kwds`

*metaclass*는 적절한 메타 클래스이고, *namespace*는 준비된 클래스 이름 공간이며 *kwds*는 `'metaclass'` 항목이 제거된 전달된 *kwds* 인자의 갱신된 사본입니다. *kwds* 인자가 전달되지 않으면, 빈 딕셔너리가 됩니다.

Added in version 3.3.

버전 3.6에서 변경: 반환된 튜플의 `namespace` 요소의 기본값이 변경되었습니다. 이제 메타 클래스에 `__prepare__` 메서드가 없으면 삽입 순서 보존 맵핑이 사용됩니다.

더 보기

metaclasses
이 함수들이 지원하는 클래스 생성 절차에 대한 자세한 내용

PEP 3115 - 파이썬 3000의 메타 클래스
`__prepare__` 이름 공간 훅을 도입했습니다

`types.`**`resolve_bases`**(*bases*)

PEP 560의 명세에 따라 MRO 항목을 동적으로 결정합니다.

이 함수는 `type`의 인스턴스가 아닌 항목들을 *bases*에서 찾고, `__mro_entries__()` 메서드가 있는 각 객체가 이 메서드 호출의 언 패킹 된 결과로 대체된 튜플을 반환합니다. *bases* 항목이 `type`의 인스턴스이거나, `__mro_entries__()` 메서드가 없으면, 반환 튜플에 변경되지 않은 상태로 포함됩니다.

Added in version 3.7.

`types.`**`get_original_bases`**(*cls*, /)

Return the tuple of objects originally given as the bases of *cls* before the `__mro_entries__()` method has been called on any bases (following the mechanisms laid out in **PEP 560**). This is useful for introspecting *Generics*.

For classes that have an `__orig_bases__` attribute, this function returns the value of `cls.__orig_bases__`. For classes without the `__orig_bases__` attribute, `cls.__bases__` is returned.

예제:

```
from typing import TypeVar, Generic, NamedTuple, TypedDict

T = TypeVar("T")
class Foo(Generic[T]): ...
class Bar(Foo[int], float): ...
class Baz(list[str]): ...
Eggs = NamedTuple("Eggs", [("a", int), ("b", str)])
Spam = TypedDict("Spam", {"a": int, "b": str})

assert Bar.__bases__ == (Foo, float)
assert get_original_bases(Bar) == (Foo[int], float)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
assert Baz.__bases__ == (list,)
assert get_original_bases(Baz) == (list[str],)

assert Eggs.__bases__ == (tuple,)
assert get_original_bases(Eggs) == (NamedTuple,)

assert Spam.__bases__ == (dict,)
assert get_original_bases(Spam) == (TypedDict,)

assert int.__bases__ == (object,)
assert get_original_bases(int) == (object,)
```

Added in version 3.12.

더 보기

PEP 560 - typing 모듈과 제네릭 형에 대한 코어 지원

8.10.2 표준 인터프리터 형

이 모듈은 파이썬 인터프리터를 구현하는 데 필요한 많은 형의 이름을 제공합니다. `listiterator` 형과 같이 처리 중에 우연히 발생하는 일부 형은 의도적으로 포함하지 않았습니다.

이러한 이름의 일반적인 용도는 *isinstance()* 나 *issubclass()* 검사입니다.

이러한 형을 인스턴스 화할 때는 서명이 파이썬 버전마다 다를 수 있음에 유의해야 합니다.

다음과 같은 형들에 대해 표준 이름이 정의됩니다:

`types.`**`NoneType`**

The type of *None*.

Added in version 3.10.

`types.`**`FunctionType`**

`types.`**`LambdaType`**

사용자 정의 함수와 `lambda` 표현식이 만든 함수의 형.

인자 `code`로 감사 이벤트 `function.__new__`를 발생시킵니다.

감사 이벤트는 함수 객체의 직접 인스턴스 화에서만 발생하며, 일반 컴파일에서는 발생하지 않습니다.

`types.`**`GeneratorType`**

제너레이터 함수가 만든, 제너레이터-이터레이터 객체의 형.

`types.`**`CoroutineType`**

`async def` 함수가 만든 코루틴 객체의 형.

Added in version 3.5.

`types.`**`AsyncGeneratorType`**

비동기 제너레이터 함수가 만든, 비동기 제너레이터-이터레이터 객체의 형.

Added in version 3.6.

class `types.`**`CodeType`**(***kwargs*)

compile() 이 반환하는 것과 같은 코드 객체의 형.

인자 `code`, `filename`, `name`, `argcount`, `posonlyargcount`, `kwonlyargcount`, `nlocals`, `stacksize`, `flags`로 감사 이벤트 `code.__new__`를 발생시킵니다.

감사된 인자는 초기화자가 요구하는 이름이나 위치와 일치하지 않을 수 있음에 유의하십시오. 감사 이벤트는 코드 객체의 직접 인스턴스 화에서만 발생하며, 일반 컴파일에서는 발생하지 않습니다.

types.**CellType**

셀 객체의 형: 이러한 객체는 함수의 클로저 변수에 대한 컨테이너로 사용됩니다.

Added in version 3.8.

types.**MethodType**

사용자 정의 클래스 인스턴스의 메서드 형.

types.**BuiltinFunctionType**

types.**BuiltinMethodType**

len() 이나 *sys.exit()* 와 같은 내장 함수와 내장 클래스의 메서드의 형. (여기서, "내장" 이라는 용어는 "C로 작성된" 을 의미합니다.)

types.**WrapperDescriptorType**

`object.__init__()` 나 `object.__lt__()` 와 같은, 일부 내장 데이터형과 베이스 클래스의 메서드의 형.

Added in version 3.7.

types.**MethodWrapperType**

일부 내장 데이터형과 베이스 클래스의 연결된(*bound*) 메서드의 형. 예를 들어 `object().__str__` 의 형입니다.

Added in version 3.7.

types.**NotImplementedType**

The type of *NotImplemented*.

Added in version 3.10.

types.**MethodDescriptorType**

str.join() 과 같은 일부 내장 데이터형의 메서드의 형.

Added in version 3.7.

types.**ClassMethodDescriptorType**

`dict.__dict__['fromkeys']` 와 같은 일부 내장 데이터형의 연결되지 않은(*unbound*) 클래스 메서드의 형.

Added in version 3.7.

class types.**ModuleType**(*name*, *doc=None*)

모듈의 형. 생성자는 만들 모듈의 이름과 선택적으로 독스트링을 취합니다.

> **더 보기**
>
> **Documentation on module objects**
> Provides details on the special attributes that can be found on instances of `ModuleType`.
>
> ***importlib.util.module_from_spec()***
> Modules created using the `ModuleType` constructor are created with many of their special attributes unset or set to default values. `module_from_spec()` provides a more robust way of creating `ModuleType` instances which ensures the various attributes are set appropriately.

types.**EllipsisType**

The type of *Ellipsis*.

Added in version 3.10.

class types.**GenericAlias**(*t_origin*, *t_args*)

list[int]와 같은 매개 변수화된 제네릭의 형입니다.

t_origin은 list, tuple 또는 dict와 같이 매개 변수화되지 않은 제네릭 클래스여야 합니다. t_args는 t_origin을 매개 변수화하는 형의 *tuple*(길이 1일 수 있습니다)이어야 합니다:

```
>>> from types import GenericAlias

>>> list[int] == GenericAlias(list, (int,))
True
>>> dict[str, int] == GenericAlias(dict, (str, int))
True
```

Added in version 3.9.

버전 3.9.2에서 변경: 이 형은 이제 서브 클래싱 할 수 있습니다.

더 보기

Generic Alias Types
In-depth documentation on instances of types.GenericAlias

PEP 585 - Type Hinting Generics In Standard Collections
Introducing the types.GenericAlias class

class types.**UnionType**

The type of *union type expressions*.

Added in version 3.10.

버전 3.14에서 변경: This is now an alias for *typing.Union*.

class types.**TracebackType**(*tb_next*, *tb_frame*, *tb_lasti*, *tb_lineno*)

sys.exception().__traceback__에서 발견되는 것과 같은 트레이스백 객체의 형.

사용 가능한 어트리뷰트와 연산에 대한 세부 사항 및 동적으로 트레이스백을 만드는 것에 대한 지침은 언어 레퍼런스를 참조하십시오.

types.**FrameType**

tb가 트레이스백 객체일 때 tb.tb_frame에서 발견되는 것과 같은 프레임 객체의 형.

types.**GetSetDescriptorType**

FrameType.f_locals나 array.array.typecode와 같은, PyGetSetDef가 있는 확장 모듈에서 정의된 객체의 형. 이 형은 객체 어트리뷰트에 대한 디스크립터로 사용됩니다. *property* 형과 같은 목적을 갖지만, 확장 모듈에 정의된 클래스에 사용됩니다.

types.**MemberDescriptorType**

datetime.timedelta.days와 같은, PyMemberDef가 있는 확장 모듈에서 정의된 객체의 형. 이 형은 표준 변환 함수를 사용하는 간단한 C 데이터 멤버의 디스크립터로 사용됩니다; *property* 형과 같은 목적을 갖지만, 확장 모듈에 정의된 클래스를 위한 것입니다.

In addition, when a class is defined with a __slots__ attribute, then for each slot, an instance of MemberDescriptorType will be added as an attribute on the class. This allows the slot to appear in the class' s __dict__.

파이썬의 다른 구현에서, 이 형은 GetSetDescriptorType과 같을 수 있습니다.

class types.**MappingProxyType**(*mapping*)

매핑의 읽기 전용 프락시. 매핑 항목에 대한 동적 뷰를 제공하는데, 매핑이 변경될 때 뷰가 이러한 변경 사항을 반영함을 의미합니다.

Added in version 3.3.

버전 3.9에서 변경: **PEP 584**의 새 병합 (|) 연산자를 지원하도록 갱신되었습니다. 단순히 하부 매핑에 위임합니다.

key in proxy

하부 매핑에 키 *key*가 있으면 `True`를, 그렇지 않으면 `False`를 반환합니다.

proxy[key]

키 *key*를 사용하여 하부 매핑의 항목을 반환합니다. *key*가 하부 매핑에 없으면 `KeyError`를 발생시킵니다.

iter(proxy)

하부 매핑의 키에 대한 이터레이터를 반환합니다. 이것은 `iter(proxy.keys())`의 줄임 표현입니다.

len(proxy)

하부 매핑의 항목 수를 반환합니다.

copy()

하부 매핑의 얕은 사본을 반환합니다.

get(*key*[, *default*])

*key*가 하부 매핑에 있으면 *key*의 값을, 그렇지 않으면 *default*를 반환합니다. *default*를 지정하지 않으면, 기본적으로 `None`으로 설정되므로, 이 메서드는 절대 `KeyError`를 발생시키지 않습니다.

items()

하부 매핑의 항목(items)(`(key, value)` 쌍)의 새 뷰를 반환합니다.

keys()

하부 매핑의 키(keys)의 새로운 뷰를 반환합니다.

values()

하부 매핑의 값(values)의 새 뷰를 반환합니다.

reversed(proxy)

하부 매핑의 키(keys)에 대한 역 이터레이터를 반환합니다.

Added in version 3.9.

hash(proxy)

하부 매핑의 해시를 반환합니다.

Added in version 3.12.

class types.**CapsuleType**

The type of capsule objects.

Added in version 3.13.

8.10.3 추가 유틸리티 클래스와 함수

class types.**SimpleNamespace**

이름 공간에 대한 어트리뷰트 액세스와 의미 있는 repr을 제공하는 간단한 `object` 서브 클래스.

Unlike `object`, with `SimpleNamespace` you can add and remove attributes.

`SimpleNamespace` objects may be initialized in the same way as `dict`: either with keyword arguments, with a single positional argument, or with both. When initialized with keyword arguments, those are directly added to the underlying namespace. Alternatively, when initialized with a positional argument, the underlying namespace will be updated with key-value pairs from that argument (either a mapping object or an *iterable* object producing key-value pairs). All such keys must be strings.

형은 다음 코드와 대략 동등합니다:

```
class SimpleNamespace:
    def __init__(self, mapping_or_iterable=(), /, **kwargs):
        self.__dict__.update(mapping_or_iterable)
        self.__dict__.update(kwargs)

    def __repr__(self):
        items = (f"{k}={v!r}" for k, v in self.__dict__.items())
        return "{}({})".format(type(self).__name__, ", ".join(items))

    def __eq__(self, other):
        if isinstance(self, SimpleNamespace) and isinstance(other,␣
↪SimpleNamespace):
            return self.__dict__ == other.__dict__
        return NotImplemented
```

`SimpleNamespace`는 `class NS: pass`의 대체품으로 유용할 수 있습니다. 하지만, 구조화된 레코드형에는 *namedtuple()*을 대신 사용하십시오.

`SimpleNamespace` objects are supported by *copy.replace()*.

Added in version 3.3.

버전 3.9에서 변경: repr의 어트리뷰트 순서가 알파벳순에서 삽입 순으로 변경되었습니다 (`dict` 처럼).

버전 3.13에서 변경: Added support for an optional positional argument.

types.**DynamicClassAttribute**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

클래스의 어트리뷰트 액세스를 __getattr__ 로 보냅니다.

디스크립터이며, 인스턴스와 클래스를 통해 액세스할 때 다르게 작동하는 어트리뷰트를 정의하는데 사용됩니다. 인스턴스 액세스는 정상적으로 유지되지만, 클래스를 통한 어트리뷰트 액세스는 클래스의 __getattr__ 메서드로 보냅니다; 이는 AttributeError를 발생 시켜 수행됩니다.

이를 통해 인스턴스에서 활성화된 프로퍼티를 가짐과 동시에, 클래스에서 같은 이름을 가진 가상 어트리뷰트를 가질 수 있습니다 (예제는 *enum.Enum*을 참조하십시오).

Added in version 3.4.

8.10.4 코루틴 유틸리티 함수

types.**coroutine**(*gen_func*)

이 함수는 제너레이터 함수를 제너레이터 기반 코루틴을 반환하는 코루틴 함수로 변환합니다. 제너레이터 기반 코루틴은 여전히 제너레이터 이터레이터이지만, 코루틴 객체로도 간주하며 어웨이터블입니다. 그러나, 반드시 `__await__()` 메서드를 구현할 필요는 없습니다.

*gen_func*가 제너레이터 함수면 제자리(in-place)에서 수정됩니다.

*gen_func*가 제너레이터 함수가 아니면, 래핑 됩니다. *collections.abc.Generator*의 인스턴스를 반환하면, 인스턴스는 어웨이터블 프락시 객체로 래핑 됩니다. 다른 모든 형의 객체는 그대로 반환됩니다.

Added in version 3.5.

8.11 copy — 얕은 복사와 깊은 복사 연산

소스 코드: Lib/copy.py

파이썬에서 대입문은 객체를 복사하지 않고, 대상과 객체 사이에 바인딩을 만듭니다. 가변(mutable) 컬렉션 또는 가변(mutable) 항목들을 포함한 컬렉션의 경우때로 컬렉션을 변경하지 않고 사본을 변경하기 위해 복사가 필요합니다. 이 모듈은 일반적인 얕은 복사와 깊은 복사 연산을 제공합니다. (아래 설명 참고)

인터페이스 요약:

copy.**copy**(*obj*)

*obj*의 얕은 사본을 반환합니다.

copy.**deepcopy**(*obj*[, *memo*])

*obj*의 깊은 사본을 반환합니다.

copy.**replace**(*obj*, /, **changes*)

Creates a new object of the same type as *obj*, replacing fields with values from *changes*.

Added in version 3.13.

exception copy.**Error**

모듈 특정 에러의 경우 발생됩니다.

얕은 복사와 깊은 복사의 차이점은복합 객체(리스트 또는 클래스 인스턴스들과 같은 다른 객체를 포함한 객체)에만 유효합니다.

- 얕은 복사는 새로운 복합 객체를 만들고,(가능한 범위까지) 원본 객체를 가리키는 참조를 새로운 복합 객체에 삽입합니다.
- 깊은 복사는 새로운 복합 객체를 만들고,재귀적으로 원본 객체의 사본을 새로 만든 복합 객체에 삽입합니다.

깊은 복사 연산은 얕은 복사 연산에는 없는 두 가지 문제가 있습니다:

- 재귀 객체(직접적 또는 간접적으로 자신에 대한 참조를 포함하는 복합 객체)는 순환 루프의 원인이 될 수 있습니다.
- 깊은 복사는 모든 것을 복사하기 때문에, 지나치게 많이 복사할 수 있습니다. 가령, 복사본 간에 공유할 의도가 있는 것까지도.

deepcopy() 함수는 다음과 같은 방법으로 이 문제들을 피합니다:

- 현재 복사 패스 중에 이미 복사된 객체의 memo 딕셔너리를 가지고 있습니다; 그리고
- 사용자 정의 클래스가 복사 연산 또는 복사된 구성요소 집합을 재정의하도록 합니다.

이 모듈은 모듈, 메서드, 스택 트레이스, 스택 프레임, 파일, 소켓, 윈도우나 그 비슷한 유형들은 복사하지 않습니다. 원래 객체를 변화시키지 않고 반환함으로써 함수와 클래스를 (얕고 깊게) "복사" 합니다; 이것은 *pickle* 모듈로 처리되는 방식과 호환됩니다.

딕셔너리의 얕은 복사는 *dict.copy()*를 사용하여 복사할 수 있습니다.그리고 리스트의 얕은 복사는 예를 들어 copied_list = original_list[:] 처럼전체 리스트의 슬라이스를 대입하여 리스트를 복사할 수도 있습니다.

클래스는 피클링을 제어하기 위해 사용하는 것과 같은 인터페이스를 사용하여 복사를 제어할 수 있습니다. 이러한 메서드들의 정보는 *pickle* 모듈 설명을 참고하세요.실제로 *copy* 모듈은 *copyreg* 모듈에 등록된 피클 함수를 사용합니다.

클래스가 자체적으로 복사 구현을 정의하기 위해선, *__copy__()* 와 *__deepcopy__()* 같은 특수 메서드를 정의할 수 있습니다.

object.**__copy__**(*self*)

얕은 복사 연산을 실행하기 위해 호출됩니다; 추가적인 인자를 전달 하지 않습니다.

object.**__deepcopy__**(*self*, *memo*)

깊은 복사 연산을 실행하기 위해 호출됩니다; *memo* 딕셔너리가 하나의 인자로 전달됩니다. __deepcopy__ 구현에서 구성요소의 깊은 복사를 만들기 위해선, 구성요소를 첫 번째 인자로 하고 *memo* 딕셔너리를 두 번째 인자로 하여 *deepcopy()* 함수를 호출해야 합니다. *memo* 딕셔너리는 불투명 객체로 취급해야 합니다.

Function copy.replace() is more limited than *copy()* and *deepcopy()*, and only supports named tuples created by *namedtuple()*, *dataclasses*, and other classes which define method *__replace__()*.

object.**__replace__**(*self*, /, **changes)

This method should create a new object of the same type, replacing fields with values from *changes*.

Added in version 3.13.

더 보기

모듈 *pickle*
객체 상태 조회와 복원을 지원하는데 사용되는 특수 메서드에 관한 논의

8.12 pprint — 예쁜 데이터 인쇄기

소스 코드: Lib/pprint.py

---

pprint 모듈은 임의의 파이썬 데이터 구조를 인터프리터의 입력으로 사용할 수 있는 형태로 "예쁘게 인쇄" 할 수 있는 기능을 제공합니다. 포맷된 구조에 기본 파이썬 형이 아닌 객체가 포함되면, 표현은 로드되지 않을 수 있습니다. 파일, 소켓 또는 클래스와 같은 객체뿐만 아니라 파이썬 리터럴로 표현할 수 없는 다른 많은 객체가 포함된 경우입니다.

포맷된 표현은 할 수 있다면 객체를 한 줄에 유지하고, 허용된 너비에 맞지 않으면 여러 줄로 나눕니다. 허용된 너비는 *width* 매개 변수로 조정할 수 있고, 기본 값은 80 자 입니다.

딕셔너리는 디스플레이를 계산하기 전에 키로 정렬됩니다.

버전 3.9에서 변경: *types.SimpleNamespace*를 예쁘게 인쇄하는 지원이 추가되었습니다.

버전 3.10에서 변경: *dataclasses.dataclass*를 예쁘게 인쇄하는 지원이 추가되었습니다.

8.12.1 함수

pprint.**pp**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, *, *compact=False*, *sort_dicts=False*, *underscore_numbers=False*)

*object*의 포맷된 표현에 줄 바꿈을 추가해서 인쇄합니다. 이 함수는 *print()* 함수 대신 대화형 인터프리터에서 값을 검사하는 데 사용할 수 있습니다. 팁: 스코프 내에서 사용하기 위해 `print = pprint.pp`를 다시 대입할 수도 있습니다.

매개변수

- **object** – The object to be printed.
- **stream** (*file-like object* | None) – A file-like object to which the output will be written by calling its `write()` method. If `None` (the default), *sys.stdout* is used.
- **indent** (`int`) – 각 중첩 수준에 대해 추가되는 들여쓰기 양.
- **width** (`int`) – The desired maximum number of characters per line in the output. If a structure cannot be formatted within the width constraint, a best effort will be made.
- **depth** (`int` | *None*) – The number of nesting levels which may be printed. If the data structure being printed is too deep, the next contained level is replaced by `....` If `None` (the default), there is no constraint on the depth of the objects being formatted.
- **compact** (`bool`) – Control the way long *sequences* are formatted. If `False` (the default), each item of a sequence will be formatted on a separate line, otherwise as many items as will fit within the *width* will be formatted on each output line.
- **sort_dicts** (`bool`) – If `True`, dictionaries will be formatted with their keys sorted, otherwise they will be displayed in insertion order (the default).
- **underscore_numbers** (`bool`) – If `True`, integers will be formatted with the _ character for a thousands separator, otherwise underscores are not displayed (the default).

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pp(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

Added in version 3.8.

pprint.**pprint**(*object, stream=None, indent=1, width=80, depth=None, *, compact=False, sort_dicts=True, underscore_numbers=False*)

Alias for `pp()` with *sort_dicts* set to `True` by default, which would automatically sort the dictionaries' keys, you might want to use `pp()` instead where it is `False` by default.

pprint.**pformat**(*object, indent=1, width=80, depth=None, *, compact=False, sort_dicts=True, underscore_numbers=False*)

*object*의 포맷된 표현을 문자열로 반환합니다. *indent*, *width*, *depth*, *compact*, *sort_dicts* 및 *underscore_numbers*는 포매팅 매개 변수로 `PrettyPrinter` 생성자에 전달되고, 그 의미는 위 설명서에서 기술된 대입니다.

pprint.**isreadable**(*object*)

*object*의 포맷된 표현이 "읽을 수 있는"지, 즉 `eval()`을 사용하여 값을 재구성하는 데 사용할 수 있는지 판단합니다. 재귀적 객체에 대해서는 항상 `False`를 반환합니다.

```
>>> pprint.isreadable(stuff)
False
```

pprint.**isrecursive**(*object*)

Determine if *object* requires a recursive representation. This function is subject to the same limitations as noted in `saferepr()` below and may raise an `RecursionError` if it fails to detect a recursive object.

pprint.**saferepr**(*object*)

일부 일반적인 자료 구조(`dict`, `list`, `tuple`과 `__repr__`이 재정의되지 않은 서브클래스의 인스턴스)에서의 재귀에 대해 보호되는, *object*의 문자열 표현을 반환합니다. 객체의 표현이 재귀적 항목을 노출하면, 재귀적 참조는 `<Recursion on typename with id=number>`로 표시됩니다. 표현에는 이외의 다른 포매팅이 적용되지 않습니다.

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni
↪']"
```

8.12.2 PrettyPrinter 객체

class pprint.**PrettyPrinter**(*indent=1, width=80, depth=None, stream=None, *, compact=False, sort_dicts=True, underscore_numbers=False*)

`PrettyPrinter` 인스턴스를 만듭니다.

Arguments have the same meaning as for `pp()`. Note that they are in a different order, and that *sort_dicts* defaults to `True`.

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> pp.pprint(stuff)
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
>>> pp = pprint.PrettyPrinter(width=41, compact=True)
>>> pp.pprint(stuff)
[['spam', 'eggs', 'lumberjack',
  'knights', 'ni'],
 'spam', 'eggs', 'lumberjack', 'knights',
 'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',))))))))
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

버전 3.4에서 변경: *compact* 매개 변수가 추가되었습니다.

버전 3.8에서 변경: *sort_dicts* 매개 변수가 추가되었습니다.

버전 3.10에서 변경: *underscore_numbers* 매개 변수가 추가되었습니다.

버전 3.11에서 변경: No longer attempts to write to `sys.stdout` if it is `None`.

`PrettyPrinter` 인스턴스에는 다음과 같은 메서드가 있습니다:

`PrettyPrinter.`**`pformat`**(*object*)

*object*의 포맷된 표현을 반환합니다. `PrettyPrinter` 생성자에 전달된 옵션을 고려합니다.

`PrettyPrinter.`**`pprint`**(*object*)

구성된 스트림에 *object*의 포맷된 표현과 불 넘김을 인쇄합니다.

다음 메서드는 같은 이름의 해당 함수에 대한 구현을 제공합니다. 새로운 `PrettyPrinter` 객체를 만들 필요가 없으므로, 인스턴스에서 이러한 메서드를 사용하는 것이 약간 더 효율적입니다.

`PrettyPrinter.`**`isreadable`**(*object*)

object의 포맷된 표현이 "읽을 수 있는"지, 즉 `eval()`을 사용하여 값을 재구성하는 데 사용할 수 있는지 판단합니다. 재귀 객체에 대해 `False`를 반환함에 유의하십시오. `PrettyPrinter`의 *depth* 매개 변수가 설정되고 객체가 허용된 것보다 더 깊으면, `False`를 반환합니다.

`PrettyPrinter.`**`isrecursive`**(*object*)

object가 재귀적 표현을 요구하는지 판단합니다.

이 메서드는 서브 클래스가 객체가 문자열로 변환되는 방식을 수정할 수 있도록 하는 훅으로 제공됩니다. 기본 구현은 `saferepr()` 구현의 내부를 사용합니다.

`PrettyPrinter.`**`format`**(*object*, *context*, *maxlevels*, *level*)

세 가지 값을 반환합니다: 포맷된 버전의 *object*를 문자열로, 결과가 읽을 수 있는지를 나타내는 플래그와 재귀가 감지되었는지를 나타내는 플래그. 첫 번째 인자는 표시할 객체입니다. 두 번째는 현재 표현 컨텍스트(표현에 영향을 주는 *object*의 직접 및 간접 컨테이너)의 일부인 객체의 `id()`를 키로 포함하는 딕셔너리입니다; 이미 *context*에 표현된 객체가 표현되어야 할 필요가 있으면, 세 번째 반환 값은 `True`이어야 합니다. `format()` 메서드에 대한 재귀 호출은 컨테이너에 대한 추가 항목을 이 딕셔너리에 추가해야 합니다. 세 번째 인자 *maxlevels*는 재귀에 요청된 제한을 줍니다; 요청된 제한이 없으면 `0`입니다. 이 인자는 재귀 호출에 수정되지 않은 채 전달되어야 합니다. 네 번째 인자 *level*은 현재 수준을 제공합니다; 재귀 호출은 현재 호출보다 작은 값으로 전달되어야 합니다.

8.12.3 예제

pp() 함수와 매개 변수의 여러 용도를 예시하기 위해, PyPI에서 프로젝트에 대한 정보를 가져옵시다:

```
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('https://pypi.org/pypi/sampleproject/1.2.0/json') as resp:
...     project_info = json.load(resp)['info']
```

기본적인 형태에서, *pp()*는 전체 객체를 보여줍니다:

```
>>> pprint.pp(project_info)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': ['Development Status :: 3 - Alpha',
                 'Intended Audience :: Developers',
                 'License :: OSI Approved :: MIT License',
                 'Programming Language :: Python :: 2',
                 'Programming Language :: Python :: 2.6',
                 'Programming Language :: Python :: 2.7',
                 'Programming Language :: Python :: 3',
                 'Programming Language :: Python :: 3.2',
                 'Programming Language :: Python :: 3.3',
                 'Programming Language :: Python :: 3.4',
                 'Topic :: Software Development :: Build Tools'],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {'last_day': -1, 'last_month': -1, 'last_week': -1},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
 'project_urls': {'Download': 'UNKNOWN',
                  'Homepage': 'https://github.com/pypa/sampleproject'},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

결과는 특정 *depth*로 제한될 수 있습니다 (더 깊은 내용에는 줄임표가 사용됩니다):

```
>>> pprint.pp(project_info, depth=1)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

또한, 최대 문자 *width*를 제안할 수 있습니다. 긴 객체를 분할 할 수 없으면, 지정된 너비를 초과합니다:

```
>>> pprint.pp(project_info, depth=1, width=60)
{'author': 'The Python Packaging Authority',
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
'author_email': 'pypa-dev@googlegroups.com',
'bugtrack_url': None,
'classifiers': [...],
'description': 'A sample Python project\n'
               '=======================\n'
               '\n'
               'This is the description file for the '
               'project.\n'
               '\n'
               'The file should use UTF-8 encoding and be '
               'written using ReStructured Text. It\n'
               'will be used to generate the project '
               'webpage on PyPI, and should be written '
               'for\n'
               'that purpose.\n'
               '\n'
               'Typical contents for this file would '
               'include an overview of the project, '
               'basic\n'
               'usage examples, etc. Generally, including '
               'the project changelog in here is not\n'
               'a good idea, although a simple "What\'s '
               'New" section for the most recent version\n'
               'may be appropriate.',
'description_content_type': None,
'docs_url': None,
'download_url': 'UNKNOWN',
'downloads': {...},
'home_page': 'https://github.com/pypa/sampleproject',
'keywords': 'sample setuptools development',
'license': 'MIT',
'maintainer': None,
'maintainer_email': None,
'name': 'sampleproject',
'package_url': 'https://pypi.org/project/sampleproject/',
'platform': 'UNKNOWN',
'project_url': 'https://pypi.org/project/sampleproject/',
'project_urls': {...},
'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
'requires_dist': None,
'requires_python': None,
'summary': 'A sample Python project',
'version': '1.2.0'}
```

8.13 reprlib — 대안 repr() 구현

소스 코드: Lib/reprlib.py

---

reprlib 모듈은 결과 문자열의 크기에 제한이 있는 객체 표현을 생성하는 수단을 제공합니다. 파이썬 디버거에서 사용되며 다른 문맥에서도 유용할 수 있습니다.

이 모듈은 클래스, 인스턴스 및 함수를 제공합니다.:

class reprlib.**Repr**(*, *maxlevel=6*, *maxtuple=6*, *maxlist=6*, *maxarray=5*, *maxdict=4*, *maxset=6*, *maxfrozenset=6*, *maxdeque=6*, *maxstring=30*, *maxlong=40*, *maxother=30*, *fillvalue='...'*, *indent=None*)

내장 `repr()`과 유사한 함수를 구현하는 데 유용한 포매팅 서비스를 제공하는 클래스; 과도하게 긴 표현의 생성을 피하고자 객체 형별로 크기 제한이 추가됩니다.

The keyword arguments of the constructor can be used as a shortcut to set the attributes of the `Repr` instance. Which means that the following initialization:

```
aRepr = reprlib.Repr(maxlevel=3)
```

Is equivalent to:

```
aRepr = reprlib.Repr()
aRepr.maxlevel = 3
```

See section *Repr Objects* for more information about `Repr` attributes.

버전 3.12에서 변경: Allow attributes to be set via keyword arguments.

`reprlib.`**`aRepr`**

아래에 설명된 `repr()`로 함수를 제공하는 데 사용되는 `Repr`의 인스턴스입니다. 이 객체의 어트리뷰트를 변경하면 `repr()`과 파이썬 디버거에서 사용되는 크기 제한에 영향을 줍니다.

`reprlib.`**`repr`**(*obj*)

`aRepr`의 `repr()` 메서드입니다. 같은 이름의 내장 함수에 의해 반환된 것과 비슷한 문자열을 반환하지만, 대부분의 크기에는 제한이 있습니다.

크기 제한 도구 외에도, 모듈은 `__repr__()`에 대한 재귀 호출을 감지하고 대신 자리 표시자 문자열을 치환하는 데코레이터를 제공합니다.

`@reprlib.`**`recursive_repr`**(*fillvalue*='...')

같은 스레드 내에서의 재귀 호출을 감지하는 `__repr__()` 메서드용 데코레이터. 재귀 호출이 이루어지면, *fillvalue*가 반환되고, 그렇지 않으면 평상시의 `__repr__()` 호출이 수행됩니다. 예를 들어:

```
>>> from reprlib import recursive_repr
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

Added in version 3.2.

8.13.1 Repr 객체

`Repr` 인스턴스는 여러 객체 형의 표현에 대한 크기 제한과 특정 객체 형을 포맷하는 메서드를 제공하는데 사용될 수 있습니다.

`Repr.`**`fillvalue`**

This string is displayed for recursive references. It defaults to `....`

Added in version 3.11.

`Repr.`**`maxlevel`**

재귀적 표현의 생성에 대한 심도 한계. 기본값은 `6`입니다.

`Repr.`**`maxdict`**

`Repr.`**`maxlist`**

Repr.**maxtuple**

Repr.**maxset**

Repr.**maxfrozenset**

Repr.**maxdeque**

Repr.**maxarray**

명명된 객체 형을 표현하는 항목 수 제한. 기본값은 *maxdict*은 4, *maxarray*는 5 이고 그 외는 6 입니다.

Repr.**maxlong**

정수 표현의 최대 문자 수입니다. 숫자는 가운데에서 삭제됩니다. 기본값은 40 입니다.

Repr.**maxstring**

문자열 표현의 문자 수 제한. 문자열의 "통상" 표현이 문자 소스로써 사용되는 것에 주의해 주세요: 표현에 이스케이프 시퀀스가 필요하면, 표현이 짧아질 때 이것이 망가질 수 있습니다. 기본값은 30 입니다.

Repr.**maxother**

이 제한은 *Repr* 객체에서 구체적인 포맷 메서드를 사용할 수 없는 객체 형의 크기를 제어하는 데 사용됩니다. *maxstring*과 비슷한 방식으로 적용됩니다. 기본값은 20 입니다.

Repr.**indent**

If this attribute is set to None (the default), the output is formatted with no line breaks or indentation, like the standard *repr()*. For example:

```
>>> example = [
...     1, 'spam', {'a': 2, 'b': 'spam eggs', 'c': {3: 4.5, 6: []}}, 'ham']
>>> import reprlib
>>> aRepr = reprlib.Repr()
>>> print(aRepr.repr(example))
[1, 'spam', {'a': 2, 'b': 'spam eggs', 'c': {3: 4.5, 6: []}}, 'ham']
```

If *indent* is set to a string, each recursion level is placed on its own line, indented by that string:

```
>>> aRepr.indent = '-->'
>>> print(aRepr.repr(example))
[
-->1,
-->'spam',
-->{
-->-->'a': 2,
-->-->'b': 'spam eggs',
-->-->'c': {
-->-->-->3: 4.5,
-->-->-->6: [],
-->-->},
-->},
-->'ham',
]
```

Setting *indent* to a positive integer value behaves as if it was set to a string with that number of spaces:

```
>>> aRepr.indent = 4
>>> print(aRepr.repr(example))
[
    1,
    'spam',
    {
        'a': 2,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        'b': 'spam eggs',
        'c': {
            3: 4.5,
            6: [],
        },
    },
    'ham',
]
```

Added in version 3.12.

`Repr.repr(obj)`

인스턴스에 의해 부과된 포매팅을 사용하는 내장 `repr()`와 등등합니다.

`Repr.repr1(obj, level)`

`repr()`에서 사용되는 재귀적 구현. *obj*의 형을 사용하여 호출할 포매팅 메서드를 결정하고, *obj* 와 *level*을 전달합니다. 형별 메서드는 재귀적 포매팅을 수행하기 위해 `repr1()`을 호출해야 하는데, 재귀 호출에서 *level* 값으로 `level - 1`을 사용합니다.

`Repr.repr_TYPE(obj, level)`

특정 형의 포매팅 메서드는 형 이름에 기반하는 이름의 메서드로 구현됩니다. 메서드 이름에서, **TYPE** 은 `'_'.join(type(obj).__name__.split())`으로 치환됩니다. 이 메서드로의 디스패치는 `repr1()`에 의해 처리됩니다. 재귀적으로 값을 포맷해야 하는 형별 메서드는 `self.repr1(subobj, level - 1)`을 호출해야 합니다.

8.13.2 Repr 객체 서브 클래싱

`Repr.repr1()`에 의한 동적 디스패치의 사용은 `Repr`의 서브 클래스가 추가 내장 객체 형에 대한 지원을 추가하거나 이미 지원되는 형의 처리를 수정할 수 있도록 합니다. 이 예제는 파일 객체에 대한 특별한 지원이 어떻게 추가될 수 있는지 보여줍니다:

```
import reprlib
import sys

class MyRepr(reprlib.Repr):

    def repr_TextIOWrapper(self, obj, level):
        if obj.name in {'<stdin>', '<stdout>', '<stderr>'}:
            return obj.name
        return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))          # '<stdin>' 을 인쇄합니다
```

```
<stdin>
```

8.14 enum — 열거형 지원

Added in version 3.4.

소스 코드: Lib/enum.py

> **Important**
>
> This page contains the API reference information. For tutorial information and discussion of more advanced topics, see

- Basic Tutorial
- Advanced Tutorial
- Enum Cookbook

열거형은:

- is a set of symbolic names (members) bound to unique values
- can be iterated over to return its canonical (i.e. non-alias) members in definition order
- uses *call* syntax to return members by value
- uses *index* syntax to return members by name

Enumerations are created either by using `class` syntax, or by using function-call syntax:

```
>>> from enum import Enum

>>> # 클래스 문법
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3

>>> # 함수형 문법
>>> Color = Enum('Color', [('RED', 1), ('GREEN', 2), ('BLUE', 3)])
```

`class` 문법을 사용하여 Enum을 만들 수 있기는 하지만, Enum은 일반적인 파이썬 클래스가 아닙니다. 자세한 내용은 열거형은 어떻게 다릅니까? 를 참조하십시오.

참고

명명법

- `Color` 클래스는 열거형 (*enumeration*) (또는 *enum*) 입니다.
- `Color.RED`, `Color.GREEN` 등의 어트리뷰트는 열거형 멤버 (*enumeration members*) (또는 *members*) 이며 기능상 상수입니다.
- 열거형 멤버에는 이름 (*names*) 과 값 (*values*) 이 있습니다 (`Color.RED` 의 이름은 `RED`, `Color.BLUE` 의 값은 `3`, 등)

8.14.1 모듈 내용

EnumType

The `type` for Enum and its subclasses.

Enum

열거형 상수를 만들기 위한 베이스 클래스.

IntEnum

*int*의 서브 클래스이기도 한 열거형 상수를 만들기 위한 베이스 클래스. (*Notes*)

StrEnum

*str*의 서브 클래스이기도 한 열거형 상수를 만들기 위한 베이스 클래스. (*Notes*)

Flag

Flag 멤버십을 잃지 않고 비트 연산을 사용하여 결합할 수 있는 열거형 상수를 만들기 위한 베이스 클래스.

IntFlag

IntFlag 멤버십을 잃지 않고 비트 연산자를 사용하여 결합할 수 있는 열거형 상수를 만들기 위한 베이스 클래스. *IntFlag* 멤버도 *int*의 서브 클래스입니다. (*Notes*)

ReprEnum

Used by *IntEnum*, *StrEnum*, and *IntFlag* to keep the *str()* of the mixed-in type.

EnumCheck

An enumeration with the values `CONTINUOUS`, `NAMED_FLAGS`, and `UNIQUE`, for use with *verify()* to ensure various constraints are met by a given enumeration.

FlagBoundary

An enumeration with the values `STRICT`, `CONFORM`, `EJECT`, and `KEEP` which allows for more fine-grained control over how invalid values are dealt with in an enumeration.

EnumDict

A subclass of *dict* for use when subclassing *EnumType*.

auto

Instances are replaced with an appropriate value for Enum members. *StrEnum* defaults to the lower-cased version of the member name, while other Enums default to 1 and increase from there.

property()

Allows *Enum* members to have attributes without conflicting with member names. The `value` and `name` attributes are implemented this way.

unique()

한 값에 하나의 이름 만 연결되도록 하는 Enum 클래스 데코레이터.

verify()

Enum class decorator that checks user-selectable constraints on an enumeration.

member()

Make `obj` a member. Can be used as a decorator.

nonmember()

Do not make `obj` a member. Can be used as a decorator.

global_enum()

Modify the *str()* and *repr()* of an enum to show its members as belonging to the module instead of its class, and export the enum members to the global namespace.

show_flag_values()

Return a list of all power-of-two integers contained in a flag.

Added in version 3.6: `Flag`, `IntFlag`, `auto`

Added in version 3.11: `StrEnum`, `EnumCheck`, `ReprEnum`, `FlagBoundary`, `property`, `member`, `nonmember`, `global_enum`, `show_flag_values`

Added in version 3.13: `EnumDict`

8.14.2 Data Types

class enum.**EnumType**

EnumType is the *metaclass* for *enum* enumerations. It is possible to subclass *EnumType* – see Subclassing EnumType for details.

`EnumType` is responsible for setting the correct `__repr__()`, `__str__()`, `__format__()`, and `__reduce__()` methods on the final *enum*, as well as creating the enum members, properly handling duplicates, providing iteration over the enum class, etc.

Added in version 3.11: Before 3.11 `EnumType` was called `EnumMeta`, which is still available as an alias.

__call__(*cls*, *value*, *names=None*, **, module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

This method is called in two different ways:

- to look up an existing member:

  cls
  The enum class being called.

  value
  The value to lookup.

- to use the `cls` enum to create a new enum (only if the existing enum does not have any members):

  cls
  The enum class being called.

  value
  The name of the new Enum to create.

  names
  The names/values of the members for the new Enum.

  module
  새로운 Enum 이 만들어지는 모듈의 이름.

  qualname
  이 Enum 이 모듈에서 실제로 위치한 곳.

  type
  A mix-in type for the new Enum.

  start
  The first integer value for the Enum (used by `auto`).

  boundary
  How to handle out-of-range values from bit operations (`Flag` only).

__contains__(*cls*, *member*)

Returns `True` if member belongs to the `cls`:

```
>>> some_var = Color.RED
>>> some_var in Color
True
>>> Color.RED.value in Color
True
```

버전 3.12에서 변경: Before Python 3.12, a `TypeError` is raised if a non-Enum-member is used in a containment check.

__dir__(*cls*)

Returns `['__class__', '__doc__', '__members__', '__module__']` and the names of the members in *cls*:

```
>>> dir(Color)
['BLUE', 'GREEN', 'RED', '__class__', '__contains__', '__doc__', '__
↪getitem__', '__init_subclass__', '__iter__', '__len__', '__members__', '_
↪_module__', '__name__', '__qualname__']
```

__getitem__(*cls*, *name*)

Returns the Enum member in *cls* matching *name*, or raises a `KeyError`:

```
>>> Color['BLUE']
<Color.BLUE: 3>
```

__iter__(*cls*)

cls 의 각 멤버를 정의 순서로 반환합니다:

```
>>> list(Color)
[<Color.RED: 1>, <Color.GREEN: 2>, <Color.BLUE: 3>]
```

__len__(*cls*)

Returns the number of member in *cls*:

```
>>> len(Color)
3
```

__members__

Returns a mapping of every enum name to its member, including aliases

__reversed__(*cls*)

Returns each member in *cls* in reverse definition order:

```
>>> list(reversed(Color))
[<Color.BLUE: 3>, <Color.GREEN: 2>, <Color.RED: 1>]
```

class enum.**Enum**

Enum is the base class for all *enum* enumerations.

name

The name used to define the `Enum` member:

```
>>> Color.BLUE.name
'BLUE'
```

value

The value given to the `Enum` member:

```
>>> Color.RED.value
1
```

멤버의 값, `__new__()` 에서 설정할 수 있습니다.

> **참고**
>
> Enum 멤버 값
>
> 멤버 값은 아무것이나 될 수 있습니다: `int`, `str` 등. 정확한 값이 중요하지 않다면, `auto` 인스턴스를 사용할 수 있으며 적절한 값이 선택됩니다. 자세한 내용은 `auto`를 참조하세요.
>
> While mutable/unhashable values, such as `dict`, `list` or a mutable `dataclass`, can be used, they will have a quadratic performance impact during creation relative to the total number of mutable/unhashable values in the enum.

name

멤버의 이름.

value

멤버의 값, `__new__()`에서 설정할 수 있습니다.

order

더는 사용되지 않습니다, 하위 호환성을 위해 유지됩니다. (클래스 생성 중 제거되는 클래스 어트리뷰트)

ignore

`_ignore_` is only used during creation and is removed from the enumeration once creation is complete.

`_ignore_` is a list of names that will not become members, and whose names will also be removed from the completed enumeration. See TimePeriod for an example.

__dir__(*self*)

Returns `['__class__', '__doc__', '__module__', 'name', 'value']` and any public methods defined on *self.__class__*:

```
>>> from datetime import date
>>> class Weekday(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
...     @classmethod
...     def today(cls):
...         print('today is %s' % cls(date.today().isoweekday()).name)
...
>>> dir(Weekday.SATURDAY)
['__class__', '__doc__', '__eq__', '__hash__', '__module__', 'name', 'today
↪', 'value']
```

_generate_next_value_(*name*, *start*, *count*, *last_values*)

name
The name of the member being defined (e.g. 'RED').

start
The start value for the Enum; the default is 1.

count
The number of members currently defined, not including this one.

last_values
A list of the previous values.

A *staticmethod* that is used to determine the next value returned by `auto`:

```
>>> from enum import auto
>>> class PowersOfThree(Enum):
...     @staticmethod
...     def _generate_next_value_(name, start, count, last_values):
...         return 3 ** (count + 1)
...     FIRST = auto()
...     SECOND = auto()
...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> PowersOfThree.SECOND.value
9
```

__init__(*self*, **args*, ***kwds*)

By default, does nothing. If multiple values are given in the member assignment, those values become separate arguments to `__init__`; e.g.

```
>>> from enum import Enum
>>> class Weekday(Enum):
...     MONDAY = 1, 'Mon'
```

`Weekday.__init__()` would be called as `Weekday.__init__(self, 1, 'Mon')`

__init_subclass__(*cls*, ***kwds*)

A *classmethod* that is used to further configure subsequent subclasses. By default, does nothing.

missing(*cls*, *value*)

A *classmethod* for looking up values not found in *cls*. By default it does nothing, but can be overridden to implement custom search behavior:

```
>>> from enum import StrEnum
>>> class Build(StrEnum):
...     DEBUG = auto()
...     OPTIMIZED = auto()
...     @classmethod
...     def _missing_(cls, value):
...         value = value.lower()
...         for member in cls:
...             if member.value == value:
...                 return member
...         return None
...
>>> Build.DEBUG.value
'debug'
>>> Build('deBUG')
<Build.DEBUG: 'debug'>
```

__new__(*cls*, **args*, ***kwds*)

By default, doesn' t exist. If specified, either in the enum class definition or in a mixin class (such as `int`), all values given in the member assignment will be passed; e.g.

```
>>> from enum import Enum
>>> class MyIntEnum(int, Enum):
...     TWENTYSIX = '1a', 16
```

results in the call `int('1a', 16)` and a value of `26` for the member.

> **참고**
>
> When writing a custom `__new__`, do not use `super().__new__` – call the appropriate `__new__` instead.

__repr__(*self*)

Returns the string used for *repr*() calls. By default, returns the *Enum* name, member name, and value, but can be overridden:

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __repr__(self):
...         cls_name = self.__class__.__name__
...         return f'{cls_name}.{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.
↪ALTERNATE}"
(OtherStyle.ALTERNATE, 'OtherStyle.ALTERNATE', 'OtherStyle.ALTERNATE')
```

__str__(*self*)

Returns the string used for *str*() calls. By default, returns the *Enum* name and member name, but can be overridden:

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __str__(self):
...         return f'{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.
↪ALTERNATE}"
(<OtherStyle.ALTERNATE: 1>, 'ALTERNATE', 'ALTERNATE')
```

__format__(*self*)

Returns the string used for *format*() and *f-string* calls. By default, returns `__str__()` return value, but can be overridden:

```
>>> class OtherStyle(Enum):
...     ALTERNATE = auto()
...     OTHER = auto()
...     SOMETHING_ELSE = auto()
...     def __format__(self, spec):
...         return f'{self.name}'
...
>>> OtherStyle.ALTERNATE, str(OtherStyle.ALTERNATE), f"{OtherStyle.
↪ALTERNATE}"
(<OtherStyle.ALTERNATE: 1>, 'OtherStyle.ALTERNATE', 'ALTERNATE')
```

참고

Using `auto` with `Enum` results in integers of increasing value, starting with `1`.

버전 3.12에서 변경: Added enum-dataclass-support

_add_alias_()

Adds a new name as an alias to an existing member:

```
>>> Color.RED._add_alias_("ERROR")
>>> Color.ERROR
<Color.RED: 1>
```

Raises a `NameError` if the name is already assigned to a different member.

Added in version 3.13.

_add_value_alias_()

Adds a new value as an alias to an existing member:

```
>>> Color.RED._add_value_alias_(42)
>>> Color(42)
<Color.RED: 1>
```

Raises a `ValueError` if the value is already linked with a different member.

Added in version 3.13.

class enum.**IntEnum**

IntEnum is the same as `Enum`, but its members are also integers and can be used anywhere that an integer can be used. If any integer operation is performed with an *IntEnum* member, the resulting value loses its enumeration status.

```
>>> from enum import IntEnum
>>> class Number(IntEnum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...
>>> Number.THREE
<Number.THREE: 3>
>>> Number.ONE + Number.TWO
3
>>> Number.THREE + 5
8
>>> Number.THREE == 3
True
```

참고

Using `auto` with `IntEnum` results in integers of increasing value, starting with `1`.

버전 3.11 에서 변경: `__str__()` is now `int.__str__()` to better support the *replacement of existing constants* use-case. `__format__()` was already `int.__format__()` for that same reason.

class enum.**StrEnum**

StrEnum is the same as `Enum`, but its members are also strings and can be used in most of the same places that a string can be used. The result of any string operation performed on or with a *StrEnum* member is not part of the enumeration.

```
>>> from enum import StrEnum, auto
>>> class Color(StrEnum):
...     RED = 'r'
...     GREEN = 'g'
...     BLUE = 'b'
...     UNKNOWN = auto()
...
>>> Color.RED
<Color.RED: 'r'>
>>> Color.UNKNOWN
<Color.UNKNOWN: 'unknown'>
>>> str(Color.UNKNOWN)
'unknown'
```

> **참고**
>
> There are places in the stdlib that check for an exact `str` instead of a `str` subclass (i.e. `type(unknown) == str` instead of `isinstance(unknown, str)`), and in those locations you will need to use `str(MyStrEnum.MY_MEMBER)`.

> **참고**
>
> Using `auto` with `StrEnum` results in the lower-cased member name as the value.

> **참고**
>
> `__str__()` is `str.__str__()` to better support the *replacement of existing constants* use-case. `__format__()` is likewise `str.__format__()` for that same reason.

Added in version 3.11.

class enum.**Flag**

`Flag` is the same as `Enum`, but its members support the bitwise operators `&` (*AND*), `|` (*OR*), `^` (*XOR*), and `~` (*INVERT*); the results of those operations are (aliases of) members of the enumeration.

__contains__(*self*, *value*)

Returns *True* if value is in self:

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> purple = Color.RED | Color.BLUE
>>> white = Color.RED | Color.GREEN | Color.BLUE
>>> Color.GREEN in purple
False
>>> Color.GREEN in white
True
>>> purple in white
True
>>> white in purple
False
```

__iter__(self):

Returns all contained non-alias members:

```
>>> list(Color.RED)
[<Color.RED: 1>]
>>> list(purple)
[<Color.RED: 1>, <Color.BLUE: 4>]
```

Added in version 3.11.

__len__(self):

Returns number of members in flag:

```
>>> len(Color.GREEN)
1
>>> len(white)
3
```

Added in version 3.11.

__bool__(self):

Returns *True* if any members in flag, *False* otherwise:

```
>>> bool(Color.GREEN)
True
>>> bool(white)
True
>>> black = Color(0)
>>> bool(black)
False
```

__or__(*self*, *other*)

Returns current flag binary or' ed with other:

```
>>> Color.RED | Color.GREEN
<Color.RED|GREEN: 3>
```

__and__(*self*, *other*)

Returns current flag binary and' ed with other:

```
>>> purple & white
<Color.RED|BLUE: 5>
>>> purple & Color.GREEN
<Color: 0>
```

__xor__(*self*, *other*)

Returns current flag binary xor' ed with other:

```
>>> purple ^ white
<Color.GREEN: 2>
>>> purple ^ Color.GREEN
<Color.RED|GREEN|BLUE: 7>
```

__invert__(self):

Returns all the flags in *type*(*self*) that are not in *self*:

```
>>> ~white
<Color: 0>
>>> ~purple
<Color.GREEN: 2>
>>> ~Color.RED
<Color.GREEN|BLUE: 6>
```

_numeric_repr_()

Function used to format any remaining unnamed numeric values. Default is the value' s repr; common choices are `hex()` and `oct()`.

> **참고**
>
> Using `auto` with `Flag` results in integers that are powers of two, starting with `1`.

버전 3.11에서 변경: The *repr*() of zero-valued flags has changed. It is now:

```
>>> Color(0)
<Color: 0>
```

class enum.**IntFlag**

`IntFlag` is the same as `Flag`, but its members are also integers and can be used anywhere that an integer can be used.

```
>>> from enum import IntFlag, auto
>>> class Color(IntFlag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> Color.RED & 2
<Color: 0>
>>> Color.RED | 2
<Color.RED|GREEN: 3>
```

If any integer operation is performed with an *IntFlag* member, the result is not an *IntFlag*:

```
>>> Color.RED + 2
3
```

If a `Flag` operation is performed with an *IntFlag* member and:

- the result is a valid *IntFlag*: an *IntFlag* is returned
- the result is not a valid *IntFlag*: the result depends on the `FlagBoundary` setting

The `repr()` of unnamed zero-valued flags has changed. It is now:

```
>>> Color(0)
<Color: 0>
```

> **참고**
>
> Using `auto` with `IntFlag` results in integers that are powers of two, starting with `1`.

버전 3.11에서 변경: `__str__()` is now `int.__str__()` to better support the *replacement of existing constants* use-case. `__format__()` was already `int.__format__()` for that same reason.

Inversion of an `IntFlag` now returns a positive value that is the union of all flags not in the given flag, rather than a negative value. This matches the existing `Flag` behavior.

class enum.**ReprEnum**

`ReprEnum` uses the `repr()` of `Enum`, but the `str()` of the mixed-in data type:

- `int.__str__()` for `IntEnum` and `IntFlag`
- `str.__str__()` for `StrEnum`

Inherit from `ReprEnum` to keep the `str()` / `format()` of the mixed-in data type instead of using the `Enum`-default `str()`.

Added in version 3.11.

class enum.**EnumCheck**

EnumCheck contains the options used by the `verify()` decorator to ensure various constraints; failed constraints result in a `ValueError`.

UNIQUE

Ensure that each value has only one name:

```
>>> from enum import Enum, verify, UNIQUE
>>> @verify(UNIQUE)
... class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     CRIMSON = 1
Traceback (most recent call last):
...
ValueError: aliases found in <enum 'Color'>: CRIMSON -> RED
```

CONTINUOUS

Ensure that there are no missing values between the lowest-valued member and the highest-valued member:

```
>>> from enum import Enum, verify, CONTINUOUS
>>> @verify(CONTINUOUS)
... class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 5
Traceback (most recent call last):
...
ValueError: invalid enum 'Color': missing values 3, 4
```

NAMED_FLAGS

Ensure that any flag groups/masks contain only named flags – useful when values are specified instead of being generated by `auto()`:

```
>>> from enum import Flag, verify, NAMED_FLAGS
>>> @verify(NAMED_FLAGS)
... class Color(Flag):
...     RED = 1
...     GREEN = 2
...     BLUE = 4
...     WHITE = 15
...     NEON = 31
Traceback (most recent call last):
...
ValueError: invalid Flag 'Color': aliases WHITE and NEON are missing␣
↪combined values of 0x18 [use enum.show_flag_values(value) for details]
```

> **참고**
>
> CONTINUOUS and NAMED_FLAGS are designed to work with integer-valued members.

Added in version 3.11.

class enum.**FlagBoundary**

`FlagBoundary` controls how out-of-range values are handled in `Flag` and its subclasses.

STRICT

Out-of-range values cause a `ValueError` to be raised. This is the default for `Flag`:

```
>>> from enum import Flag, STRICT, auto
>>> class StrictFlag(Flag, boundary=STRICT):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> StrictFlag(2**2 + 2**4)
Traceback (most recent call last):
...
ValueError: <flag 'StrictFlag'> invalid value 20
    given 0b0 10100
  allowed 0b0 00111
```

CONFORM

Out-of-range values have invalid values removed, leaving a valid *Flag* value:

```
>>> from enum import Flag, CONFORM, auto
>>> class ConformFlag(Flag, boundary=CONFORM):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> ConformFlag(2**2 + 2**4)
<ConformFlag.BLUE: 4>
```

EJECT

Out-of-range values lose their *Flag* membership and revert to *int*.

```
>>> from enum import Flag, EJECT, auto
>>> class EjectFlag(Flag, boundary=EJECT):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> EjectFlag(2**2 + 2**4)
20
```

KEEP

Out-of-range values are kept, and the *Flag* membership is kept. This is the default for *IntFlag*:

```
>>> from enum import Flag, KEEP, auto
>>> class KeepFlag(Flag, boundary=KEEP):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...
>>> KeepFlag(2**2 + 2**4)
<KeepFlag.BLUE|16: 20>
```

Added in version 3.11.

class enum.**EnumDict**

EnumDict is a subclass of *dict* that is used as the namespace for defining enum classes (see prepare). It is exposed to allow subclasses of *EnumType* with advanced behavior like having multiple values per member. It should be called with the name of the enum class being created, otherwise private names and internal classes will not be handled correctly.

Note that only the *MutableMapping* interface (`__setitem__()` and *update()*) is overridden. It may be possible to bypass the checks using other `dict` operations like `|=`.

member_names

멤버 이름의 리스트.

Added in version 3.13.

지원되는 __dunder__ 이름

*__members__*는 member_name:member 항목의 읽기 전용 순서 있는 매핑입니다. 클래스에서만 이용할 수 있습니다.

지정된다면, *__new__()*는 열거형 멤버를 만들고 반환해야 합니다; 멤버의 _value_를 적절하게 설정하는 것도 좋습니다. 일단 모든 멤버가 만들어지면 더는 사용되지 않습니다.

지원되는 _sunder_ 이름

- *_name_* – 멤버의 이름
- *_value_* – 멤버의 값; __new__에서 설정할 수 있습니다
- *_missing_()* – 값을 찾을 수 없을 때 사용되는 조회 함수; 재정의할 수 있습니다
- *_ignore_* – 멤버로 변환되지 않고 최종 클래스에서 제거될 *list*나 *str* 형의 이름 목록
- *_order_* – 더는 사용되지 않습니다, 하위 호환성을 위해 유지됩니다 (클래스 생성 중 제거되는 클래스 어트리뷰트)
- *_generate_next_value_()* – 열거형 멤버에 대한 적절한 값을 얻기 위해 사용합니다; 재정의할 수 있습니다

> **참고**
>
> For standard *Enum* classes the next value chosen is the highest value seen incremented by one.
>
> For *Flag* classes the next value chosen will be the next highest power-of-two.

- *_add_alias_()* – adds a new name as an alias to an existing member.
- *_add_value_alias_()* – adds a new value as an alias to an existing member.
- While _sunder_ names are generally reserved for the further development of the *Enum* class and can not be used, some are explicitly allowed:
  - _repr_* (e.g. _repr_html_), as used in IPython' s rich display

Added in version 3.6: _missing_, _order_, _generate_next_value_

Added in version 3.7: _ignore_

Added in version 3.13: _add_alias_, _add_value_alias_, _repr_*

8.14.3 Utilities and Decorators

class enum.**auto**

auto can be used in place of a value. If used, the *Enum* machinery will call an *Enum*' s *_generate_next_value_()* to get an appropriate value. For *Enum* and *IntEnum* that appropriate value will be the last value plus one; for *Flag* and *IntFlag* it will be the first power-of-two greater than the highest value; for *StrEnum* it will be the lower-cased version of the member' s name. Care must be taken if mixing *auto*() with manually specified values.

auto instances are only resolved when at the top level of an assignment:

- FIRST = auto() will work (auto() is replaced with 1);

- `SECOND = auto(), -2` will work (auto is replaced with `2`, so `2, -2` is used to create the `SECOND` enum member;
- `THREE = [auto(), -3]` will *not* work (`<auto instance>, -3` is used to create the `THREE` enum member)

버전 3.11.1에서 변경: In prior versions, `auto()` had to be the only thing on the assignment line to work properly.

`_generate_next_value_` can be overridden to customize the values used by *auto*.

> **참고**
>
> in 3.13 the default `_generate_next_value_` will always return the highest member value incremented by 1, and will fail if any member is an incompatible type.

`@enum.`**`property`**

A decorator similar to the built-in *property*, but specifically for enumerations. It allows member attributes to have the same names as members themselves.

> **참고**
>
> the *property* and the member must be defined in separate classes; for example, the *value* and *name* attributes are defined in the *Enum* class, and *Enum* subclasses can define members with the names `value` and `name`.

Added in version 3.11.

`@enum.`**`unique`**

열거형 용 `class` 데코레이터입니다. 열거형의 *__members__*를 검색하여, 별칭을 수집합니다; 발견되면 *ValueError*가 세부 정보와 함께 발생합니다:

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

`@enum.`**`verify`**

A `class` decorator specifically for enumerations. Members from *EnumCheck* are used to specify which constraints should be checked on the decorated enumeration.

Added in version 3.11.

`@enum.`**`member`**

A decorator for use in enums: its target will become a member.

Added in version 3.11.

`@enum.`**`nonmember`**

A decorator for use in enums: its target will not become a member.

Added in version 3.11.

@enum.**global_enum**

A decorator to change the *str()* and *repr()* of an enum to show its members as belonging to the module instead of its class. Should only be used when the enum members are exported to the module global namespace (see *re.RegexFlag* for an example).

Added in version 3.11.

enum.**show_flag_values**(*value*)

Return a list of all power-of-two integers contained in a flag *value*.

Added in version 3.11.

8.14.4 참고 사항

IntEnum, *StrEnum*, 및 *IntFlag*

These three enum types are designed to be drop-in replacements for existing integer- and string-based values; as such, they have extra limitations:

- `__str__` uses the value and not the name of the enum member
- `__format__`, because it uses `__str__`, will also use the value of the enum member instead of its name

If you do not need/want those limitations, you can either create your own base class by mixing in the `int` or `str` type yourself:

```
>>> from enum import Enum
>>> class MyIntEnum(int, Enum):
...     pass
```

or you can reassign the appropriate *str()*, etc., in your enum:

```
>>> from enum import Enum, IntEnum
>>> class MyIntEnum(IntEnum):
...     __str__ = Enum.__str__
```

8.15 graphlib — 그래프와 유사한 구조에 작동하는 기능

소스 코드: Lib/graphlib.py

---

class graphlib.**TopologicalSorter**(*graph=None*)

해시 가능 노드의 그래프(graph)를 위상 정렬(topological sort)하는 기능을 제공합니다.

위상 순서(topological order)는 그래프(graph)에서 꼭짓점(vertex)의 선형 순서로, 꼭짓점 u에서 꼭짓점 v로 가는 모든 유향 변(directed edge) u -> v 에 대해, 꼭짓점 u가 꼭짓점 v보다 앞에 옵니다. 예를 들어, 그래프의 꼭짓점은 수행될 작업을 나타낼 수 있고, 변은 하나의 작업이 다른 작업보다 먼저 수행되어야 한다는 제약을 나타낼 수 있습니다; 이 예에서, 위상 순서는 유효한 작업 순서입니다. 그래프에 유향 순환이 없는 경우, 즉 유향 비순환 그래프인 경우에만 완전한 위상 정렬이 가능합니다.

선택적 *graph* 인자가 제공되면 키가 노드이고 값이 그래프에서 해당 노드의 모든 선행 노드(키의 값을 가리키는 변이 있는 노드)의 이터러블인 비순환 그래프를 나타내는 딕셔너리이어야 합니다). *add()* 메서드를 사용하여 그래프에 추가 노드를 추가할 수 있습니다.

일반적으로, 주어진 그래프의 정렬을 수행하는 데 필요한 단계는 다음과 같습니다:

- 선택적 초기 그래프를 사용하여 *TopologicalSorter*의 인스턴스를 만듭니다.
- 그래프에 노드를 추가합니다.

- 그래프에서 *prepare()*를 호출합니다.
- *is_active()*가 True인 동안, *get_ready()*가 반환하는 노드를 이터레이트하고 이들을 처리합니다. 처리가 완료됨에 따라, 각 노드에 *done()*을 호출합니다.

그래프에서 노드의 즉각적인 정렬이 필요하고 병렬화가 개입하지 않으면, 편의 메서드 *TopologicalSorter.static_order()*를 직접 사용할 수 있습니다:

```
>>> graph = {"D": {"B", "C"}, "C": {"A"}, "B": {"A"}}
>>> ts = TopologicalSorter(graph)
>>> tuple(ts.static_order())
('A', 'C', 'B', 'D')
```

이 클래스는 노드가 준비됨에 따라 병렬 처리를 쉽게 지원하도록 설계되었습니다. 예를 들어:

```
topological_sorter = TopologicalSorter()

# 'topological_sorter' 에 노드를 추가합니다...

topological_sorter.prepare()
while topological_sorter.is_active():
    for node in topological_sorter.get_ready():
        # 작업자 스레드나 프로세스는 'task_queue' 큐에서 작업할 노드를 꺼냅니다.
        task_queue.put(node)

    # 노드의 작업이 완료되면, 작업자는 노드를 'finalized_tasks_queue' 에 넣어서
    # 작업할 더 많은 노드를 얻을 수 있도록 합니다. 'is_active()'의 정의는, 이 시점에
서,
    # 아직 'done()'으로 전달되지 않은 하나 이상의 노드가 'task_queue'에 있다고 보장
해서,
    # 이 블로킹 'get()'은 (결국) 성공합니다. 'done()'을 호출한 후, 다시 'get_
↪ready()'를
    # 호출하도록 루프로 돌아가서, 새로 풀려난 노드를 논리적으로 가능한 한 빨리
↪'task_queue'에
    # 넣습니다.
    node = finalized_tasks_queue.get()
    topological_sorter.done(node)
```

add(*node*, **predecessors*)

새 노드와 그 선행 노드를 그래프에 추가합니다. *node*와 *predecessors*의 모든 요소는 모두 해시 가능해야 합니다.

같은 노드 인자로 여러 번 호출되면, 종속성 집합은 전달된 모든 종속성의 합집합입니다.

종속성이 없는 노드를 추가하거나(*predecessors*가 제공되지 않는 경우) 종속성을 두 번 제공할 수 있습니다. 이전에 제공되지 않은 노드가 *predecessors*에 포함되면, 노드는 그 자신의 선행 노드 없이 자동으로 그래프에 추가됩니다.

prepare() 이후에 호출되면 *ValueError*가 발생합니다.

prepare()

그래프를 완료로 표시하고 그래프에서 순환을 검사합니다. 순환이 감지되면, *CycleError*가 발생하지만, 순환이 더 진행하는 것을 차단할 때까지 *get_ready()*를 사용하여 여전히 가능한 많은 노드를 얻을 수 있습니다. 이 함수를 호출한 후에는, 그래프를 수정할 수 없어서, *add()*를 사용하여 더는 노드를 추가할 수 없습니다.

A *ValueError* will be raised if the sort has been started by *static_order()* or *get_ready()*.

버전 3.14에서 변경: prepare() can now be called more than once as long as the sort has not started. Previously this raised *ValueError*.

is_active()

더 진행할 수 있으면 `True`를, 그렇지 않으면 `False`를 반환합니다. 순환이 결정을 차단하지 않고 *TopologicalSorter.get_ready()*에 의해 아직 반환되지 않은 준비된 노드가 아직 있거나 *TopologicalSorter.done()*으로 표시된 노드 수가 *TopologicalSorter.get_ready()*에 의해 반환된 수보다 작으면 진행할 수 있습니다.

이 클래스의 `__bool__()` 메서드는 이 함수로 위임됩니다, 그래서 다음 대신:

```
if ts.is_active():
    ...
```

다음처럼 간단하게 할 수 있습니다:

```
if ts:
    ...
```

이전에 *prepare()*를 호출하지 않고 호출되면 *ValueError*가 발생합니다.

done(**nodes*)

*TopologicalSorter.get_ready()*에 의해 반환된 노드 집합이 처리된 것으로 표시하여, *nodes*에 있는 각 노드의 모든 후속 노드들이 *TopologicalSorter.get_ready()*에 대한 호출로 나중에 반환되도록 차단 해제합니다.

*nodes*에 있는 노드가 이 메서드에 대한 이전 호출에 의해 이미 처리된 것으로 표시되었거나 *TopologicalSorter.add()*를 사용하여 그래프에 추가되지 않았거나, *prepare()*를 호출하지 않고 호출되었거나, 또는 *get_ready()*가 아직 노드를 반환하지 않았으면 *ValueError*를 발생시킵니다.

get_ready()

준비된 모든 노드가 담긴 `tuple`을 반환합니다. 처음에는 선행 노드가 없는 모든 노드를 반환하며, 일단 *TopologicalSorter.done()*을 호출하여 처리된 것으로 표시되면, 추가 호출은 모든 선행 노드가 이미 처리된 모든 새 노드를 반환합니다. 더는 진행할 수 없으면, 빈 튜플이 반환됩니다.

이전에 *prepare()*를 호출하지 않고 호출되면 *ValueError*가 발생합니다.

static_order()

노드들을 위상 순서로 이터레이트하는 이터레이터 객체를 반환합니다. 이 메서드를 사용할 때, *prepare()*와 *done()*은 호출하지 않아야 합니다. 이 메서드는 다음과 동등합니다:

```
def static_order(self):
    self.prepare()
    while self.is_active():
        node_group = self.get_ready()
        yield from node_group
        self.done(*node_group)
```

반환되는 특정 순서는 항목이 그래프에 삽입된 특정 순서에 따라 달라질 수 있습니다. 예를 들면:

```
>>> ts = TopologicalSorter()
>>> ts.add(3, 2, 1)
>>> ts.add(1, 0)
>>> print([*ts.static_order()])
[2, 0, 1, 3]

>>> ts2 = TopologicalSorter()
>>> ts2.add(1, 0)
>>> ts2.add(3, 2, 1)
>>> print([*ts2.static_order()])
[0, 2, 1, 3]
```

이것은 그래프에서 "0" 과 "2" 가 같은 수준에 있고 (*get_ready()* 에 대한 같은 호출에서 반환됩니다) 이들 간의 순서는 삽입 순서에 따라 결정되기 때문입니다.

순환이 감지되면 *CycleError* 가 발생합니다.

Added in version 3.9.

8.15.1 예외

graphlib 모듈은 다음 예외를 정의합니다:

exception graphlib.**CycleError**

작업 그래프에 순환이 있으면 *TopologicalSorter.prepare()* 가 발생시키는 *ValueError* 의 서브클래스. 여러 순환이 존재하면, 그들 중 오직 하나의 정의되지 않은 선택만 보고되고 예외에 포함됩니다.

감지된 순환은 예외 인스턴스의 *args* 어트리뷰트에서 두 번째 요소를 통해 액세스 할 수 있으며 각 노드가 그래프에서 리스트에 있는 다음 노드의 직전 선행 노드가 되도록 노드 리스트로 구성됩니다. 보고된 리스트에서, 순환임을 분명히 하기 위해, 처음과 마지막 노드는 같습니다.

CHAPTER 9

숫자와 수학 모듈

이 장에 나와있는 모듈들은 숫자와 수학에 관련된 함수와 데이터 타입을 제공합니다. *numbers* 모듈은 숫자 데이터 타입을 위한 추상 계층 구조를 정의합니다. *math* 와 *cmath* 모듈은 부동소수와 복소수를 위한 여러 수학 함수를 가지고 있습니다. *decimal* 모듈은 임의의 정밀도 계산을 사용하여 정확한 10진수 표현을 지원합니다.

이 장에는 다음과 같은 모듈이 설명되어 있습니다:

9.1 numbers — 숫자 추상 베이스 클래스

소스 코드: Lib/numbers.py

---

numbers 모듈(**PEP 3141**)은 숫자에 대한 추상 베이스 클래스 의 계층 구조를 정의합니다. 계층 구조가 깊어질수록 더 많은 연산이 정의되어 있습니다. 이 모듈에 정의된 모든 형은 인스턴스로 만들 수 없습니다.

class numbers.**Number**

숫자 계층의 최상위 클래스입니다. 형에 상관없이 인자 x 가 숫자인지 확인하려면 `isinstance(x, Number)` 를 사용하세요.

9.1.1 숫자 계층

class numbers.**Complex**

이 서브 클래스는 복소수를 표현하고 내장 *complex* 형에 사용되는 연산을 포함합니다. 여기에는 *complex* 와 *bool* 형으로의 변환과 *real*, *imag*, +, -, *, /, **, *abs()*, *conjugate()*, ==, != 이 포함됩니다. - 와 != 를 제외하고는 모두 추상입니다.

real

추상. 복소수의 실수부를 반환합니다.

imag

추상. 복소수의 허수부를 반환합니다.

abstractmethod conjugate()

추상 메서드. 켤레 복소수를 반환합니다. 예를 들어 `(1+3j).conjugate() == (1-3j)` 입니다.

class numbers.**Real**

Real 클래스는 *Complex* 클래스에 실수 연산을 추가합니다.

요약하면 *float* 로의 변환과 *math.trunc()*, *round()*, *math.floor()*, *math.ceil()*, *divmod()*, //, %, <, <=, >, >= 가 포함됩니다.

이 클래스는 또한 *complex()*, *real*, *imag*, *conjugate()* 를 위한 기본값을 제공합니다.

class numbers.**Rational**

Real 의 하위 형이고 *numerator* 와 *denominator* 프로퍼티가 추가됩니다. 또한 *float()* 함수를 위한 기본값을 제공합니다.

The *numerator* and *denominator* values should be instances of *Integral* and should be in lowest terms with *denominator* positive.

numerator

프로퍼티(추상 메서드)

denominator

프로퍼티(추상 메서드)

class numbers.**Integral**

Rational 의 하위 형이고 *int* 클래스로 변환 기능이 추가됩니다. *float()*, *numerator*, *denominator* 를 위한 기본값을 제공합니다. 모듈러스가 있는 *pow()*를 위한 추상 메서드와 비트 연산 <<, >>, &, ^, |, ~ 를 추가합니다.

9.1.2 형 구현자를 위한 주의 사항

구현자는 동일한 숫자가 같게 취급되고 같은 값으로 해싱되도록 해야 합니다. 만약 종류가 다른 실수의 하위 형이 있는 경우 조금 까다로울 수 있습니다. 예를 들어 *fractions.Fraction* 클래스는 *hash()* 함수를 다음과 같이 구현합니다:

```
def __hash__(self):
    if self.denominator == 1:
        # 정수입니다.
        return hash(self.numerator)
    # 비싼 검사이지만, 확실히 맞습니다.
    if self == float(self):
        return hash(float(self))
    else:
        # 간단한 분수에서의 높은 충돌률을 피하기위해 튜플의 해시를 사용합니다.
        return hash((self.numerator, self.denominator))
```

더 많은 숫자 추상 베이스 클래스(ABC) 추가

물론 숫자를 위한 ABC를 추가하는 것이 가능합니다. 그렇지 않으면 엉망으로 상속 계층이 구현될 것입니다. *Complex* 와 *Real* 사이에 다음과 같이 MyFoo 를 추가할 수 있습니다:

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

산술 연산 구현

다른 형에 대한 연산은 두 인자의 형에 관해 알고 있는 구현을 호출하거나 두 인자를 가장 비슷한 내장형으로 변환하여 연산하도록 산술 연산을 구현하는 것이 좋습니다. *Integral* 클래스의 하위 형일 경우에 __add__() 와 __radd__() 메서드는 다음과 같이 정의되어야 함을 의미합니다:

```
class MyIntegral(Integral):

    def __add__(self, other):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

Complex 클래스의 서브클래스에는 다섯 가지의 서로 다른 혼합형 연산이 있습니다. 위의 코드에서 `MyIntegral` 와 `OtherTypeIKnowAbout` 를 제외한 나머지를 기본구조라고 하겠습니다. `a` 는 *Complex* 의 하위 형인 `A` 의 인스턴스입니다(즉 `a : A <: Complex` 입니다). 비슷하게 `b : B <: Complex` 입니다. `a + b` 인 경우를 생각해 보겠습니다:

1. 만약 `A` 가 `b` 를 받는 `__add__()` 메서드를 정의했다면 모든 것이 문제없이 처리됩니다.
2. `A` 가 기본구조 코드로 진입하고 `__add__()` 로 부터 어떤 값을 반환한다면 `B` 가 똑똑하게 정의한 `__radd__()` 메서드를 놓칠 수 있습니다. 이를 피하려면 기본구조는 `__add__()` 에서 *NotImplemented* 를 반환해야 합니다. (또는 `A` 가 `__add__()` 메서드를 전혀 구현하지 않을 수도 있습니다.)
3. 그다음 `B` 의 `__radd__()` 메서드가 기회를 얻습니다. 이 메서드가 `a` 를 받을 수 있다면 모든 것이 문제없이 처리됩니다.
4. 기본구조 코드로 돌아온다면 더 시도해 볼 수 있는 메서드가 없으므로 기본적으로 수행될 구현을 작성해야 합니다.
5. 만약 `B <: A` 라면 파이썬은 `A.__add__` 메서드 전에 `B.__radd__` 를 시도합니다. `A` 에 대해서 알고 `B` 가 구현되었기 때문에 이런 행동은 문제없습니다. 따라서 *Complex* 에 위임하기 전에 이 인스턴스를 처리할 수 있습니다.

만약 어떤 것도 공유하지 않는 `A <: Complex` 와 `B <: Real` 라면 적절한 공유 연산(shared operation)은 내장 *complex* 클래스에 연관된 것입니다. 양쪽의 `__radd__()` 메서드가 여기에 해당하므로 `a+b == b+a` 가 됩니다.

대부분 주어진 어떤 형에 대한 연산은 매우 비슷하므로, 주어진 연산자의 정방향(forward) 인스턴스와 역방향(reverse) 인스턴스를 생성하는 헬퍼 함수를 정의하는 것이 유용합니다. 예를 들어 *fractions.Fraction* 클래스는 다음과 같이 사용합니다:

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # 정수를 포함합니다.
            return monomorphic_operator(a, b)
        elif isinstance(a, Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

9.2 math — 수학 함수

This module provides access to common mathematical functions and constants, including those defined by the C standard.

이 함수는 복소수와 함께 사용할 수 없습니다; 복소수를 지원해야 하면 *cmath* 모듈에 있는 같은 이름의 함수를 사용하십시오. 대부분 사용자는 복소수를 이해하는 데 필요한 수준의 수학을 배우고 싶어 하지 않기 때문에 복소수를 지원하는 함수와 그렇지 않은 함수를 구별했습니다. 복소수 결과 대신 예외를 수신하면 매개 변수로 사용된 예상치 못한 복소수를 조기에 감지할 수 있기 때문에, 프로그래머는 처음 위치에서 생성된 경로와 원인을 파악할 수 있습니다.

이 모듈에서 제공하는 함수는 다음과 같습니다. 달리 명시되지 않는 한 모든 반환 값은 float 입니다.

수론적 함수	
comb(n, k)	반복과 순서 없이 n 개의 항목에서 k 개의 항목을 선택하는 방법의 수
factorial(n)	n 계승
*gcd(*integers)*	정수 인자의 최대 공약수
isqrt(n)	음이 아닌 정수 n의 정수 제곱근
*lcm(*integers)*	정수 인자의 최소 공배수
perm(n, k)	반복 없고 순서 있게 n 개의 항목에서 k 개의 항목을 선택하는 방법의
부동 소수점 산술	
ceil(x)	x의 천장값, x보다 크거나 같은 가장 작은 정수
fabs(x)	x의 절댓값
floor(x)	x의 바닥값, x보다 작거나 같은 가장 큰 정수

다음 페이지에 계

표 1 – 이전 페이지에서 계속

`fma(x, y, z)`	Fused multiply-add operation: `(x * y) + z`
`fmod(x, y)`	나누기 `x / y`의 나머지
`modf(x)`	*x*의 소수와 정수 부분
`remainder(x, y)`	*y*에 대한 *x*의 나머지
`trunc(x)`	*x*의 정수부
부동 소수점 조작 함수	
`copysign(x, y)`	*x*의 크기(절댓값)와 *y*의 부호를 갖는 값
`frexp(x)`	*x*의 가수(mantissa)와 지수(exponent)
`isclose(a, b, rel_tol, abs_tol)`	값 *a*와 *b*가 서로 가까이 있는지 검사합니다
`isfinite(x)`	*x*가 무한대나 NaN이 아닌지 검사합니다
`isinf(x)`	*x*가 양 또는 음의 무한대인지 검사합니다
`isnan(x)`	*x*가 NaN(not a number)인지 검사합니다
`ldexp(x, i)`	`x * (2**i)`, 함수 `frexp()`의 역
`nextafter(x, y, steps)`	*x* 에서 *y* 방향으로 *steps* 스텝 다음의 부동 소수점 값
`ulp(x)`	*x*의 최하위 비트 값
거듭제곱, 지수 및 로그 함수	
`cbrt(x)`	*x*의 세제곱근
`exp(x)`	*e* 의 *x* 거듭제곱
`exp2(x)`	2 의 *x* 거듭제곱
`expm1(x)`	*e* 의 *x* 거듭제곱 빼기 1
`log(x, base)`	주어진 밑(기본값 *e*)의 *x* 의 로그
`log1p(x)`	*x*의 자연로그 (밑 *e*)
`log2(x)`	*x*의 밑이 2인 로그
`log10(x)`	*x*의 밑이 10인 로그
`pow(x, y)`	*x*의 *y* 거듭제곱
`sqrt(x)`	*x*의 제곱근
합과 곱 함수	
`dist(p, q)`	좌표의 이터러블로 제공되는, 두 점 *p*와 *q* 사이의 유클리드 거리
`fsum(iterable)`	입력 *iterable*에 있는 값들의 합
`hypot(*coordinates)`	좌표 이터러블의 유클리드 크기(norm)
`prod(iterable, start)`	*start* 값으로 시작하는, 입력 *iterable*에 있는 요소들의 곱
`sumprod(p, q)`	Sum of products from two iterables *p* and *q*
각도 변환	
`degrees(x)`	각도 *x*를 라디안에서 도(degree)로 변환합니다
`radians(x)`	각도 *x*를 도(degree)에서 라디안으로 변환합니다
삼각 함수	
`acos(x)`	*x*의 아크 코사인(arc cosine)
`asin(x)`	*x*의 아크 사인(arc sine)
`atan(x)`	*x*의 아크 탄젠트(arc tangent)
`atan2(y, x)`	`atan(y / x)`
`cos(x)`	*x*의 코사인
`sin(x)`	*x*의 사인
`tan(x)`	*x*의 탄젠트
쌍곡선 함수	
`acosh(x)`	*x*의 역 쌍곡 코사인(inverse hyperbolic cosine)
`asinh(x)`	*x*의 역 쌍곡 사인(inverse hyperbolic sine)
`atanh(x)`	*x*의 역 쌍곡 탄젠트(inverse hyperbolic tangent)
`cosh(x)`	*x*의 쌍곡 코사인(hyperbolic cosine)
`sinh(x)`	*x*의 쌍곡 사인(hyperbolic sine)
`tanh(x)`	*x*의 쌍곡 탄젠트(hyperbolic tangent)
특수 함수	
`erf(x)`	*x*의 오차 함수(error function)
`erfc(x)`	*x*의 여오차 함수(complementary error function)
`gamma(x)`	*x*의 감마 함수(Gamma function)
`lgamma(x)`	*x*의 감마 함수(Gamma function)의 절댓값의 자연로그
상수	

다음 페이지에 계

표 1 – 이전 페이지에서 계속

pi	$\pi = 3.141592\cdots$
e	$e = 2.718281\cdots$
tau	$\tau = 2\pi = 6.283185\cdots$
inf	양의 무한대
nan	"Not a number" (NaN)

9.2.1 수론적 함수

`math.comb(n, k)`

반복과 순서 없이 n 개의 항목에서 k 개의 항목을 선택하는 방법의 수를 반환합니다.

`k <= n`이면 `n! / (k! * (n - k)!)`로 평가되고, `k > n`이면 0으로 평가됩니다.

`(1 + x)ⁿ`의 다항식 전개에서 k 번째 항의 계수와 같기 때문에 이항 계수(binomial coefficient)라고도 합니다.

인자 중 어느 하나라도 정수가 아니면 *TypeError*를 발생시킵니다. 인자 중 어느 하나라도 음수이면 *ValueError*를 발생시킵니다.

Added in version 3.8.

`math.factorial(n)`

Return factorial of the nonnegative integer n.

버전 3.10에서 변경: 정숫값 부동 소수점(`5.0`과 같은)은 더는 허용되지 않습니다.

`math.gcd(*integers)`

지정된 정수 인자의 최대 공약수를 반환합니다. 인자 중 하나가 0이 아니면, 반환된 값은 모든 인자를 나누는 가장 큰 양의 정수입니다. 모든 인자가 0이면, 반환 값은 `0`입니다. 인자가 없는 `gcd()`는 `0`을 반환합니다.

Added in version 3.5.

버전 3.9에서 변경: 임의의 개수 인자에 대한 지원이 추가되었습니다. 이전에는, 단지 두 개의 인자만 지원되었습니다.

`math.isqrt(n)`

음이 아닌 정수 n의 정수 제곱근을 반환합니다. 이것은 n의 정확한 제곱근의 바닥(floor)입니다, 또는, 동등하게, $a^2 \leq n$을 만족하는 가장 큰 정수 a입니다.

일부 응용 프로그램에서는, $n \leq a^2$ 을 만족하는 가장 작은 정수 a, 즉 n의 정확한 제곱근의 천장(ceiling)을 구하는 것이 더 편리합니다. 양의 n에 대해, 이것은 `a = 1 + isqrt(n - 1)`을 사용하여 계산할 수 있습니다.

Added in version 3.8.

`math.lcm(*integers)`

지정된 정수 인자의 최소 공배수를 반환합니다. 모든 인자가 0이 아니면, 반환 값은 모든 인자의 배수인 가장 작은 양의 정수입니다. 인자 중 어느 하나가 0이면, 반환 값은 `0`입니다. 인자가 없는 `lcm()`은 `1`을 반환합니다.

Added in version 3.9.

`math.perm(n, k=None)`

반복 없고 순서 있게 n 개의 항목에서 k 개의 항목을 선택하는 방법의 수를 반환합니다.

`k <= n`이면 `n! / (n - k)!`로 평가되고, `k > n`이면 0으로 평가됩니다.

k가 지정되지 않거나 `None`이면, k의 기본값은 n이고 함수는 `n!`을 반환합니다.

인자 중 어느 하나라도 정수가 아니면 *TypeError*를 발생시킵니다. 인자 중 어느 하나라도 음수이면 *ValueError*를 발생시킵니다.

Added in version 3.8.

9.2.2 부동 소수점 산술

math.**ceil**(*x*)

*x*보다 크거나 같은 가장 작은 정수인 *x*의 천장값(ceiling)을 반환합니다. *x*가 float가 아니면, `x.__ceil__`에 위임하고, 이것은 *Integral* 값을 반환해야 합니다.

math.**fabs**(*x*)

*x*의 절댓값을 반환합니다.

math.**floor**(*x*)

*x*보다 작거나 같은 가장 큰 정수인 *x*의 바닥값(floor)을 반환합니다. *x*가 float가 아니면, `x.__floor__`에 위임하고, 이것은 *Integral* 값을 반환해야 합니다.

math.**fma**(*x*, *y*, *z*)

Fused multiply-add operation. Return `(x * y) + z`, computed as though with infinite precision and range followed by a single round to the `float` format. This operation often provides better accuracy than the direct expression `(x * y) + z`.

This function follows the specification of the fusedMultiplyAdd operation described in the IEEE 754 standard. The standard leaves one case implementation-defined, namely the result of `fma(0, inf, nan)` and `fma(inf, 0, nan)`. In these cases, `math.fma` returns a NaN, and does not raise any exception.

Added in version 3.13.

math.**fmod**(*x*, *y*)

플랫폼 C 라이브러리 함수 `fmod(x, y)`이 정의한 대로, `x / y`의 부동 소수점 나머지를 반환합니다. 파이썬 표현식 `x % y`가 같은 결과를 반환하지 않을 수 있음에 유의하십시오. C 표준의 의도는 어떤 정수 *n*에 대해 `fmod(x, y)`가 `x - n*y`와 정확히 (수학적으로; 무한 정밀도로) 같고, 결과는 *x*와 같은 부호를 가지며 크기(절댓값)는 `abs(y)`보다 작아지도록 하는 것입니다. 파이썬의 `x % y`는 대신 *y*의 부호를 갖는 결과를 반환하며 float 인자에 대해 정확하게 계산할 수 없을 수 있습니다. 예를 들어, `fmod(-1e-100, 1e100)`는 `-1e-100`이지만, 파이썬의 `-1e-100 % 1e100`의 결과는 `1e100-1e-100`이며, 부동 소수점으로 정확하게 표현할 수 없어서, 의외의 `1e100`으로 반올림됩니다. 이러한 이유로, 함수 *fmod()*는 일반적으로 float로 작업할 때 선호되는 반면 파이썬의 `x % y`는 정수로 작업할 때 선호됩니다.

math.**modf**(*x*)

*x*의 소수와 정수 부분을 반환합니다. 두 결과 모두 *x*의 부호를 가지며 float입니다.

*modf()*는 C 대응물과는 다른 호출/반환 패턴을 가지고 있음에 유의하십시오: 두 번째 반환 값을 '출력 매개 변수' 로 반환하는 대신 (파이썬에는 그러한 것이 없습니다), 단일 인자를 받아서 값의 쌍을 반환합니다.

math.**remainder**(*x*, *y*)

*y*에 대한 *x*의 IEEE 754 스타일 나머지를 반환합니다. 유한한 *x*와 0이 아닌 유한한 *y*에 대해, 이것은 차이 `x - n*y`입니다. 여기서 `n`은 몫 `x / y`의 정확한 값에 가장 가까운 정수입니다. `x / y`가 두 개의 인접한 정수 사이의 정확히 중간이면, 가장 가까운 짝수 정수가 `n`으로 사용됩니다. 따라서 나머지 `r = remainder(x, y)`는 항상 `abs(r) <= 0.5 * abs(y)`를 만족합니다.

IEEE 754에 따른 특별한 경우: 특히, `remainder(x, math.inf)`는 모든 유한한 *x*에 대해서는 *x*이고, `remainder(x, 0)`과 `remainder(math.inf, x)`는 모든 NaN이 아닌 *x*에 대해 *ValueError*를 발생시킵니다. 나머지 연산의 결과가 0이면, 해당 0은 *x*와 같은 부호를 갖습니다.

IEEE 754 이진 부동 소수점을 사용하는 플랫폼에서, 이 연산의 결과는 항상 정확하게 표현 가능합니다: 자리 올림 오차는 발생하지 않습니다.

Added in version 3.7.

math.**trunc**(*x*)

Return *x* with the fractional part removed, leaving the integer part. This rounds toward 0: `trunc()` is equivalent to *floor()* for positive *x*, and equivalent to *ceil()* for negative *x*. If *x* is not a float, delegates to `x.__trunc__`, which should return an *Integral* value.

ceil(), *floor()* 및 *modf()* 함수의 경우, 충분히 큰 절댓값을 갖는 모든 부동 소수점 숫자는 정확한 정수입니다. 파이썬 float는 일반적으로 53비트 이하의 정밀도를 가지는데 (플랫폼 C double 형과 같습니다), 이때 `abs(x) >= 2**52`를 만족하는 모든 float *x*는 소수 비트를 갖지 않습니다.

9.2.3 부동 소수점 조작 함수

`math.`**`copysign`**(*x*, *y*)

*x*의 크기(절댓값)와 *y*의 부호를 갖는 float를 반환합니다. 부호 있는 0을 지원하는 플랫폼에서, `copysign(1.0, -0.0)`은 *-1.0*을 반환합니다.

`math.`**`frexp`**(*x*)

*x*의 가수(mantissa)와 지수(exponent)를 `(m, e)` 쌍으로 반환합니다. *m*은 float이고, *e*는 정수이며, 정확히 `x == m * 2**e`가 성립합니다. *x*가 0이면, `(0.0, 0)`을 반환하고, 그렇지 않으면 `0.5 <= abs(m) < 1`입니다. 이것은 float의 내부 표현을 이식성 있는 방식으로 "분리"하는 데 사용됩니다.

*frexp()*는 C 대응물과는 다른 호출/반환 패턴을 가지고 있음에 유의하십시오: 두 번째 반환 값을 '출력 매개 변수'로 반환하는 대신 (파이썬에는 그러한 것이 없습니다), 단일 인자를 받아서 값의 쌍을 반환합니다.

`math.`**`isclose`**(*a*, *b*, **, *rel_tol=1e-09*, *abs_tol=0.0*)

값 *a*와 *b*가 서로 가까이 있으면 `True`를, 그렇지 않으면 `False`를 반환합니다.

두 값이 근접한 것으로 간주하는지는 주어진 절대와 상대 허용 오차에 따라 결정됩니다. 에러가 발생하지 않으면, 결과는 다음과 같습니다: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

*rel_tol*은 상대 허용 오차입니다 – *a*와 *b* 중 더 큰 절댓값에 대해 상대적으로, *a*와 *b* 간에 허용되는 최대 차이입니다. 예를 들어, 허용 오차를 5%로 설정하려면, `rel_tol=0.05`를 전달하십시오. 기본 허용 오차는 `1e-09`이며, 두 값이 약 9자리 십진 숫자 내에서 같다는 것을 보장합니다. *rel_tol*은 음이 아니고 `1.0`보다 작아야 합니다.

abs_tol is the absolute tolerance; it defaults to `0.0` and it must be nonnegative. When comparing `x` to `0.0`, `isclose(x, 0)` is computed as `abs(x) <= rel_tol * abs(x)`, which is `False` for any nonzero `x` and *rel_tol* less than `1.0`. So add an appropriate positive *abs_tol* argument to the call.

IEEE 754 특수 값 `NaN`, `inf` 및 `-inf`는 IEEE 규칙에 따라 처리됩니다. 특히, `NaN`은 `NaN`을 포함하여 다른 어떤 값과도 근접한 것으로 간주하지 않습니다. `inf`와 `-inf`는 오직 자신과만 가까운 것으로 간주합니다.

Added in version 3.5.

> **더 보기**
>
> **PEP 485** – 근사적 동등성을 검사하는 함수

`math.`**`isfinite`**(*x*)

*x*가 무한대나 NaN이 아니면 `True`를, 그렇지 않으면 `False`를 반환합니다. (`0.0`은 유한한 것으로 간주합니다.)

Added in version 3.2.

`math.`**`isinf`**(*x*)

*x*가 양 또는 음의 무한대이면 `True`를, 그렇지 않으면 `False`를 반환합니다.

`math.`**`isnan`**(*x*)

*x*가 NaN(not a number)이면 `True`를, 그렇지 않으면 `False`를 반환합니다.

`math.`**`ldexp`**(*x*, *i*)

`x * (2**i)`를 반환합니다. 이것은 본질적으로 함수 *frexp()*의 역입니다.

math.**nextafter**(*x*, *y*, *steps=1*)

*x*에서 *y* 방향으로 *steps* 스텝 다음의 부동 소수점 값을 반환합니다.

*x*가 *y*와 같으면, *steps*이 0이 아닌 한 *y*를 반환합니다.

예:

- `math.nextafter(x, math.inf)` 는 올라갑니다: 양의 무한대를 향해.
- `math.nextafter(x, -math.inf)` 는 내려갑니다: 음의 무한대를 향해.
- `math.nextafter(x, 0.0)` 는 0을 향합니다.
- `math.nextafter(x, math.copysign(math.inf, x))` 는 0에서 멀어집니다.

*math.ulp()*도 참조하십시오.

Added in version 3.9.

버전 3.12에서 변경: *steps* 인자를 추가했습니다.

math.**ulp**(*x*)

float *x*의 최하위 비트 값을 반환합니다:

- *x*가 NaN(not a number)이면, *x*를 반환합니다.
- *x*가 음수이면, `ulp(-x)` 를 반환합니다.
- *x*가 양의 무한대이면, *x*를 반환합니다.
- *x*가 0과 같으면, 가장 작은 양의 정규화되지 않은(*denormalized*) 표현 가능한 float를 반환합니다 (가장 작은 양의 정규화된 float, *sys.float_info.min*보다 작습니다).
- *x*가 가장 큰 양의 표현 가능한 float와 같으면, *x*보다 작은 첫 번째 float가 `x - ulp(x)` 가 되도록, *x*의 최하위 비트 값을 반환합니다.
- 그렇지 않으면 (*x*가 양의 유한 수이면), *x*보다 큰 첫 번째 float가 `x + ulp(x)` 가 되도록, *x*의 최하위 비트 값을 반환합니다.

ULP는 "Unit in the Last Place(마지막 자리의 단위)"를 나타냅니다.

*math.nextafter()*와 *sys.float_info.epsilon*도 참조하십시오.

Added in version 3.9.

9.2.4 거듭제곱, 지수 및 로그 함수

math.**cbrt**(*x*)

*x*의 세제곱근을 반환합니다.

Added in version 3.11.

math.**exp**(*x*)

*e*의 *x* 거듭제곱을 반환합니다. 여기서 *e* = 2.718281… 는 자연로그의 밑(base)입니다. 일반적으로 `math.e ** x`나 `pow(math.e, x)` 보다 정확합니다.

math.**exp2**(*x*)

*2*의 *x* 거듭제곱을 반환합니다.

Added in version 3.11.

math.**expm1**(*x*)

*e*의 *x* 거듭제곱에서 1을 뺀 값을 반환합니다. 여기서 *e*는 자연로그의 밑(base)입니다. 작은 float *x*의 경우, `exp(x) - 1`의 뺄셈은 상당한 정밀도 소실을 일으킬 수 있습니다; *expm1()* 함수는 이 양을 최대 정밀도로 계산하는 방법을 제공합니다:

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

Added in version 3.2.

math.**log**(*x*[, *base*])

하나의 인자를 제공하면, *x*의 자연로그를 반환합니다 (밑 *e*).

두 개의 인자를 제공하면, 주어진 밑(*base*)으로 *x*의 로그를 반환합니다, `log(x)/log(base)` 로 계산합니다.

math.**log1p**(*x*)

*1+x*의 자연로그를 반환합니다 (밑 *e*). 결과는 0에 가까운 *x*에 대해 정확한 방식으로 계산됩니다.

math.**log2**(*x*)

*x*의 밑이 2인 로그를 반환합니다. 이것은 일반적으로 `log(x, 2)` 보다 정확합니다.

Added in version 3.3.

> **더 보기**
>
> `int.bit_length()` 는 부호와 선행 0을 제외하고 정수를 이진수로 나타내는 데 필요한 비트 수를 반환합니다.

math.**log10**(*x*)

*x*의 밑이 10인 로그를 반환합니다. 이것은 일반적으로 `log(x, 10)` 보다 정확합니다.

math.**pow**(*x*, *y*)

*x*의 *y* 거듭제곱을 반환합니다. 예외적인 경우는 최대한 IEEE 754 표준을 따릅니다. 특히, *x*가 0이거나 NaN일 때도 `pow(1.0, x)` 와 `pow(x, 0.0)` 는 항상 `1.0`을 반환합니다. *x* 와 *y*가 모두 유한하고, *x*가 음수이고, *y*가 정수가 아니면 `pow(x, y)` 는 정의되지 않고 `ValueError`를 발생시킵니다.

내장 `**` 연산자와 달리, `math.pow()` 는 두 인자를 모두 `float` 형으로 변환합니다. 정확한 정수 거듭제곱을 계산하려면 `**` 나 내장 `pow()` 함수를 사용하십시오.

버전 3.11에서 변경: The special cases `pow(0.0, -inf)` and `pow(-0.0, -inf)` were changed to return `inf` instead of raising `ValueError`, for consistency with IEEE 754.

math.**sqrt**(*x*)

*x*의 제곱근을 반환합니다.

9.2.5 합과 곱 함수

math.**dist**(*p*, *q*)

각각 좌표 시퀀스(또는 이터러블)로 제공되는, 두 점 *p*와 *q* 사이의 유클리드 거리를 반환합니다. 두 점의 차원(dimension)은 같아야 합니다.

대략 다음과 동등합니다:

```
sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))
```

Added in version 3.8.

math.**fsum**(*iterable*)

이터러블(iterable)에 있는 값의 정확한(accurate) 부동 소수점 합을 반환합니다. 여러 중간 부분 합을 추적하여 정밀도 손실을 방지합니다.

알고리즘의 정확도는 IEEE-754 산술의 보증과 자리 올림 모드가 짝수로 반올림(half-even)인 일반적인 경우에 의존합니다. 윈도우 이외의 일부 빌드에서, 하부 C 라이브러리는 확장 정밀도 덧셈을 사용하고, 때때로 중간 합을 이중 자리 올림(double-round) 하여 최하위 비트(least significant bit)에서 분리할 수 있습니다.

자세한 논의와 두 가지 대안은, ASPN cookbook recipes for accurate floating point summation을 참조하십시오.

`math.hypot(*coordinates)`

유클리드 크기(norm) `sqrt(sum(x**2 for x in coordinates))`를 반환합니다. 원점에서 coordinates로 지정된 점까지의 벡터의 길이입니다.

2차원 점 `(x, y)`의 경우, 피타고라스 정리를 사용하여 직각 삼각형의 빗변(hypotenuse)을 계산하는 것과 동등합니다, `sqrt(x*x + y*y)`.

버전 3.8에서 변경: n 차원 점에 대한 지원이 추가되었습니다. 이전에는, 2차원인 경우만 지원되었습니다.

버전 3.10에서 변경: Improved the algorithm' s accuracy so that the maximum error is under 1 ulp (unit in the last place). More typically, the result is almost always correctly rounded to within 1/2 ulp.

`math.prod(iterable, *, start=1)`

입력 이터러블(*iterable*)에 있는 모든 요소의 곱을 계산합니다. 곱의 기본 *start* 값은 `1`입니다.

iterable이 비어 있으면, start 값을 반환합니다. 이 함수는 숫자 값과 함께 사용하기 위한 것으로, 숫자가 아닌 형을 거부 할 수 있습니다.

Added in version 3.8.

`math.sumprod(p, q)`

Return the sum of products of values from two iterables *p* and *q*.

Raises `ValueError` if the inputs do not have the same length.

대략 다음과 동등합니다:

```
sum(map(operator.mul, p, q, strict=True))
```

For float and mixed int/float inputs, the intermediate products and sums are computed with extended precision.

Added in version 3.12.

9.2.6 각도 변환

`math.degrees(x)`

각도 *x*를 라디안에서 도(degree)로 변환합니다.

`math.radians(x)`

각도 *x*를 도(degree)에서 라디안으로 변환합니다.

9.2.7 삼각 함수

`math.acos(x)`

*x*의 아크 코사인(arc cosine)을 라디안으로 반환합니다. 결과는 `0`과 `pi` 사이입니다.

`math.asin(x)`

*x*의 아크 사인(arc sine)을 라디안으로 반환합니다. 결과는 `-pi/2`와 `pi/2` 사이입니다.

`math.atan(x)`

*x*의 아크 탄젠트(arc tangent)를 라디안으로 반환합니다. 결과는 `-pi/2`와 `pi/2` 사이입니다.

math.**atan2**(*y*, *x*)

atan(y / x)를 라디안으로 반환합니다. 결과는 -pi와 pi 사이입니다. 평면에 있는 원점에서 점 (x, y)까지의 벡터는 양의 X 축과 이 각도를 이룹니다. *atan2()*의 요점은 두 입력의 부호가 모두 알려져 있기 때문에 각도에 대한 정확한 사분면을 계산할 수 있다는 것입니다. 예를 들어, atan(1)과 atan2(1, 1)은 모두 pi/4이지만, atan2(-1, -1)은 -3*pi/4입니다.

math.**cos**(*x*)

x 라디안의 코사인(cosine)을 반환합니다.

math.**sin**(*x*)

x 라디안의 사인(sine)을 반환합니다.

math.**tan**(*x*)

x 라디안의 탄젠트(tangent)를 반환합니다.

9.2.8 쌍곡선 함수

쌍곡선 함수는 원 대신 쌍곡선을 기반으로 하는 삼각 함수의 동류(analog)입니다.

math.**acosh**(*x*)

*x*의 역 쌍곡 코사인(inverse hyperbolic cosine)을 반환합니다.

math.**asinh**(*x*)

*x*의 역 쌍곡 사인(inverse hyperbolic sine)을 반환합니다.

math.**atanh**(*x*)

*x*의 역 쌍곡 탄젠트(inverse hyperbolic tangent)를 반환합니다.

math.**cosh**(*x*)

*x*의 쌍곡 코사인(hyperbolic cosine)을 반환합니다.

math.**sinh**(*x*)

*x*의 쌍곡 사인(hyperbolic sine)을 반환합니다.

math.**tanh**(*x*)

*x*의 쌍곡 탄젠트(hyperbolic tangent)를 반환합니다.

9.2.9 특수 함수

math.**erf**(*x*)

*x*의 오차 함수(error function)를 반환합니다.

erf() 함수는 누적 표준 정규 분포와 같은 전통적인 통계 함수를 계산하는 데 사용할 수 있습니다:

```
def phi(x):
    '표준 정규 분포의 누적 분포 함수'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

Added in version 3.2.

math.**erfc**(*x*)

*x*의 여오차 함수를 반환합니다. 여오차 함수(complementary error function)는 1.0 - erf(x)로 정의됩니다. 뺄셈으로 인해 유효 숫자의 소실이 발생하는 *x*의 큰 값에 사용됩니다.

Added in version 3.2.

math.**gamma**(*x*)

*x*의 감마 함수(Gamma function)를 반환합니다.

Added in version 3.2.

math.**lgamma**(*x*)

x의 감마 함수의 절댓값의 자연로그를 반환합니다.

Added in version 3.2.

9.2.10 상수

math.**pi**

사용 가능한 정밀도로, 수학 상수 $\pi = 3.141592\cdots$

math.**e**

사용 가능한 정밀도로, 수학 상수 $e = 2.718281\cdots$

math.**tau**

The mathematical constant $\tau = 6.283185\cdots$, to available precision. Tau is a circle constant equal to 2π, the ratio of a circle' s circumference to its radius. To learn more about Tau, check out Vi Hart' s video Pi is (still) Wrong, and start celebrating Tau day by eating twice as much pie!

Added in version 3.6.

math.**inf**

부동 소수점 양의 무한대. (음의 무한대는 `-math.inf`를 사용하십시오.) `float('inf')`의 출력과 동등합니다.

Added in version 3.5.

math.**nan**

A floating-point "not a number" (NaN) value. Equivalent to the output of `float('nan')`. Due to the requirements of the IEEE-754 standard, `math.nan` and `float('nan')` are not considered to equal to any other numeric value, including themselves. To check whether a number is a NaN, use the *isnan()* function to test for NaNs instead of `is` or `==`. Example:

```
>>> import math
>>> math.nan == math.nan
False
>>> float('nan') == float('nan')
False
>>> math.isnan(math.nan)
True
>>> math.isnan(float('nan'))
True
```

Added in version 3.5.

버전 3.11에서 변경: It is now always available.

math 모듈은 대부분 플랫폼 C 수학 라이브러리 함수 주위의 얇은 래퍼로 구성됩니다. 예외적인 경우의 행동은 적절한 경우 C99 표준의 부록 F를 따릅니다. 현재 구현은 `sqrt(-1.0)`이나 `log(0.0)`과 같은 잘못된 연산의 경우 *ValueError*를 발생시키고 (C99 부록 F에서 잘못된 연산이나 0으로 나누기를 신호를 주도록 권장하는 경우), 오버플로 하는 결과(예를 들어, `exp(1000.0)`)의 경우 *OverflowError*를 발생시킵니다. 하나 이상의 입력 인자가 NaN이 아니면, NaN은 위의 함수에서 반환되지 않습니다; 입력 인자가 NaN이면 대부분 함수는 NaN을 반환하지만, (다시 한번 C99 부록 F를 따라) 이 규칙에는 예를 들어 `pow(float('nan'), 0.0)`이나 `hypot(float('nan'), float('inf'))`와 같은 몇 가지 예외가 있습니다.

파이썬은 신호를 주는 NaN(signaling NaN)을 조용한 NaN(quiet NaN)과 구별하기 위해 노력하지 않으며, 신호를 주는 NaN의 동작은 지정되지 않은 상태로 남아 있습니다. 일반적인 동작은 모든 NaN을 조용한 것으로 취급하는 것입니다.

더 보기

모듈 *cmath*
이 함수 중 많은 것들의 복소수 버전.

9.3 cmath — 복소수를 위한 수학 함수

이 모듈은 복소수를 위한 수학 함수에 대한 액세스를 제공합니다. 이 모듈의 함수는 정수, 부동 소수점 수 또는 복소수를 인자로 받아들입니다. 이들은 또한 `__complex__()` 나 `__float__()` 메서드를 가진 임의의 파이썬 객체를 받아들일 것입니다: 이 메서드는 객체를 각각 복소수나 부동 소수점 수로 변환하기 위해 사용되며, 함수는 변환 결과에 적용됩니다.

참고

For functions involving branch cuts, we have the problem of deciding how to define those functions on the cut itself. Following Kahan' s "Branch cuts for complex elementary functions" paper, as well as Annex G of C99 and later C standards, we use the sign of zero to distinguish one side of the branch cut from the other: for a branch cut along (a portion of) the real axis we look at the sign of the imaginary part, while for a branch cut along the imaginary axis we look at the sign of the real part.

For example, the *cmath.sqrt()* function has a branch cut along the negative real axis. An argument of `-2-0j` is treated as though it lies *below* the branch cut, and so gives a result on the negative imaginary axis:

```
>>> cmath.sqrt(-2-0j)
-1.4142135623730951j
```

But an argument of `-2+0j` is treated as though it lies above the branch cut:

```
>>> cmath.sqrt(-2+0j)
1.4142135623730951j
```

극좌표 변환	
phase(z)	z의 위상을 반환합니다
polar(z)	z의 극좌표 표현을 반환합니다
rect(r, phi)	극좌표 r과 *phi*를 가지는 복소수 z를 반환합니다
거듭제곱과 로그 함수	
exp(z)	Return e raised to the power z
log(z[, base])	Return the logarithm of z to the given *base* (e by default)
log10(z)	z의 10을 밑으로하는 로그를 반환합니다
sqrt(z)	z의 제곱근을 반환합니다
삼각 함수	
acos(z)	z의 아크 코사인을 반환합니다
asin(z)	z의 아크 사인을 반환합니다
atan(z)	z의 아크 탄젠트를 반환합니다
cos(z)	z의 코사인을 반환합니다
sin(z)	z의 사인을 반환합니다
tan(z)	z의 탄젠트를 반환합니다
쌍곡선(hyperbolic) 함수	
acosh(z)	z의 역 쌍곡선 코사인을 반환합니다
asinh(z)	z의 역 쌍곡선 사인을 반환합니다
atanh(z)	z의 역 쌍곡선 탄젠트를 반환합니다
cosh(z)	z의 쌍곡선 코사인을 반환합니다
sinh(z)	z의 쌍곡선 사인을 반환합니다
tanh(z)	z의 쌍곡선 탄젠트를 반환합니다

다음 페이지에 계속

표 2 – 이전 페이지에서 계속

분류 함수	
`isfinite(z)`	Check if all components of z are finite
`isinf(z)`	Check if any component of z is infinite
`isnan(z)`	Check if any component of z is a NaN
`isclose(a, b, *, rel_tol, abs_tol)`	a와 b 값이 서로 가까운지 검사합니다
상수	
`pi`	$\pi = 3.141592\cdots$
`e`	$e = 2.718281\cdots$
`tau`	$\tau = 2\pi = 6.283185\cdots$
`inf`	Positive infinity
`infj`	Pure imaginary infinity
`nan`	"Not a number" (NaN)
`nanj`	Pure imaginary NaN

9.3.1 극좌표 변환

파이썬 복소수 `z`는 직교 혹은 데카르트 좌표를 사용하여 내부적으로 저장됩니다. 실수부 `z.real`과 허수부 `z.imag`에 의해 완전히 결정됩니다.

극좌표(*polar coordinates*)는 복소수를 나타내는 다른 방법을 제공합니다. 극좌표에서, 복소수 z는 모듈러스(modulus) r과 위상 각(phase angle) *phi*로 정의됩니다. 모듈러스 r은 z에서 원점까지의 거리이며, 위상 *phi*는 양의 x축에서 원점과 z를 잇는 선분으로의 라디안(radian)으로 측정한 반 시계 방향 각도입니다.

네이티브 직교 좌표와 극좌표 간의 변환에 다음 함수를 사용할 수 있습니다.

`cmath.`**`phase`**`(`*z*`)`

z의 위상(z의 편각(*argument*)이라고도 합니다)을 float로 반환합니다. `phase(z)`는 `math.atan2(z.imag, z.real)`과 동등합니다. 결과는 [-π, π] 범위에 놓이고, 이 작업의 분지 절단은 음의 실수 축에 놓입니다. 결과의 부호가 `z.imag`가 0일 때도 `z.imag`의 부호와 같음을 의미합니다:

```
>>> phase(-1+0j)
3.141592653589793
>>> phase(-1-0j)
-3.141592653589793
```

참고

복소수 z의 모듈러스(절댓값)는 내장 `abs()` 함수를 사용하여 계산할 수 있습니다. 이 연산을 위한 별도의 `cmath` 모듈 함수는 없습니다.

`cmath.`**`polar`**`(`*z*`)`

z 표현을 극좌표로 반환합니다. 쌍 `(r, phi)`를 반환합니다. 여기서 r은 z의 모듈러스이고 *phi*는 z의 위상입니다. `polar(z)`는 `(abs(z), phase(z))`와 동등합니다.

`cmath.`**`rect`**`(`*r*, *phi*`)`

극좌표 r과 *phi*를 가지는 복소수 z를 반환합니다. `complex(r * math.cos(phi), r * math.sin(phi))`와 동등합니다.

9.3.2 거듭제곱과 로그 함수

`cmath.`**`exp`**`(`*z*`)`

e의 z 거듭제곱을 반환합니다. 여기서 e는 자연로그(natural logarithms)의 밑입니다.

`cmath.`**`log`**`(`*z*[, *base*]`)`

주어진 밑 *base*에 대한 z의 로그를 반환합니다. *base*가 지정되지 않으면, z의 자연로그를 반환합니다. 음의 실수 축을 따라 0에서부터 -∞ 까지 가는 하나의 분지 절단이 있습니다.

cmath.**log10**(*z*)

*z*의 밑이 10인 로그를 반환합니다. 이것은 *log()*와 같은 분지 절단을 가집니다.

cmath.**sqrt**(*z*)

*z*의 제곱근을 반환합니다. 이것은 *log()*와 같은 분지 절단을 가집니다.

9.3.3 삼각 함수

cmath.**acos**(*z*)

*z*의 아크 코사인을 반환합니다. 두 개의 분지 절단이 있습니다: 하나는 실수 축을 따라 1에서 오른쪽으로 ∞까지 확장합니다. 다른 하나는 실수 축을 따라 -1에서 왼쪽으로 -∞까지 확장합니다.

cmath.**asin**(*z*)

*z*의 아크 사인을 반환합니다. 이것은 *acos()*와 같은 분지 절단을 가집니다.

cmath.**atan**(*z*)

*z*의 아크 탄젠트를 반환합니다. 두 개의 분지 절단이 있습니다: 하나는 허수 축을 따라 `1j`에서 `∞j`까지 확장합니다. 다른 하나는 허수 축을 따라 `-1j`에서 `-∞j`까지 확장합니다.

cmath.**cos**(*z*)

*z*의 코사인을 반환합니다.

cmath.**sin**(*z*)

*z*의 사인을 반환합니다.

cmath.**tan**(*z*)

*z*의 탄젠트를 반환합니다.

9.3.4 쌍곡선(hyperbolic) 함수

cmath.**acosh**(*z*)

*z*의 역 쌍곡선 코사인을 반환합니다. 하나의 분지 절단이 있습니다, 실수 축을 따라 1에서 왼쪽으로 -∞까지 확장합니다.

cmath.**asinh**(*z*)

*z*의 역 쌍곡선 사인을 반환합니다. 두 개의 분지 절단이 있습니다: 하나는 허수 축을 따라 `1j`에서 `∞j`까지 확장합니다. 다른 하나는 허수 축을 따라 `-1j`에서 `-∞j`까지 확장합니다.

cmath.**atanh**(*z*)

*z*의 역 쌍곡선 탄젠트를 반환합니다. 두 개의 분지 절단이 있습니다: 하나는 실수 축을 따라 `1`에서 `∞`까지 확장합니다. 다른 하나는 실수 축을 따라 `-1`에서 `-∞`까지 확장합니다.

cmath.**cosh**(*z*)

*z*의 쌍곡선 코사인을 반환합니다.

cmath.**sinh**(*z*)

*z*의 쌍곡선 사인을 반환합니다.

cmath.**tanh**(*z*)

*z*의 쌍곡선 탄젠트를 반환합니다.

9.3.5 분류 함수

cmath.**isfinite**(*z*)

*z*의 실수부와 허수부가 모두 유한이면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

Added in version 3.2.

cmath.**isinf**(*z*)

*z*의 실수부나 허수부 중 하나가 무한이면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

cmath.**isnan**(*z*)

*z*의 실수부나 허수부 중 하나가 NaN이면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

cmath.**isclose**(*a*, *b*, **, rel_tol=1e-09, abs_tol=0.0*)

*a*와 *b* 값이 서로 가까우면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

두 값을 가까운 것으로 간주하는지는 주어진 절대와 상대 허용 오차에 따라 결정됩니다. 에러가 발생하지 않으면, 결과는 다음과 같습니다: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`.

*rel_tol*은 상대 허용 오차입니다 – *a*와 *b* 사이의 최대 허용 차이이고, *a*나 *b*의 절댓값 중 더 큰 값에 상대적입니다. 예를 들어, 5%의 허용 오차를 설정하려면, `rel_tol=0.05`를 전달하십시오. 기본 허용 오차는 `1e-09`이며, 이는 두 값이 약 9자리 십진 숫자 내에서 같음을 보장합니다. *rel_tol*은 음이 아니고 `1.0`보다 작아야 합니다.

*abs_tol*은 절대 허용 오차입니다; 기본값은 `0.0`이고 음이 아니어야합니다. `x`를 `0.0`과 비교할 때, `isclose(x, 0)`는 `abs(x) <= rel_tol * abs(x)`로 계산되며, 모든 `x`와 `1.0`보다 작은 rel_tol 에 대해 `False`가 됩니다. 따라서 호출에 적절한 양의 abs_tol 인자를 추가하십시오.

IEEE 754 특수 값 `NaN`, `inf` 및 `-inf`는 IEEE 규칙에 따라 처리됩니다. 특히, `NaN`은 `NaN`을 포함한 다른 모든 값과 가깝다고 간주하지 않습니다. `inf`와 `-inf`는 그들 자신하고만 가깝다고 간주합니다.

Added in version 3.5.

더 보기

PEP 485 – 근사 동등을 검사하는 함수.

9.3.6 상수

cmath.**pi**

수학 상수 π의 float 값.

cmath.**e**

수학 상수 e의 float 값.

cmath.**tau**

수학 상수 τ의 float 값.

Added in version 3.6.

cmath.**inf**

부동 소수점 양의 무한대. `float('inf')`와 동등합니다.

Added in version 3.6.

cmath.**infj**

0 실수부와 양의 무한대 허수부를 갖는 복소수. `complex(0.0, float('inf'))`와 동등합니다.

Added in version 3.6.

cmath.**nan**

부동 소수점 "not a number" (NaN) 값. `float('nan')`과 동등합니다.

Added in version 3.6.

cmath.**nanj**

0 실수부와 NaN 허수부를 갖는 복소수. `complex(0.0, float('nan'))`과 동등합니다.

Added in version 3.6.

함수 선택은 모듈 *math*에서와 유사하지만 동일하지는 않습니다. 두 개의 모듈이 있는 이유는 일부 사용자가 복소수에 관심이 없고, 어쩌면 복소수가 무엇인지 모를 수도 있기 때문입니다. 그들에게는 `math.sqrt(-1)` 이 복소수를 반환하기보다 예외를 발생시키는 것이 좋습니다. 또한, *cmath*에 정의된 함수는, 결과를 실수로 표현할 수 있을 때도 항상 복소수를 반환합니다 (이때 복소수의 허수부는 0입니다).

분지 절단에 대한 참고 사항: 주어진 함수가 연속적이지 않은 점을 지나는 곡선입니다. 이것들은 많은 복소수 기능에서 필요한 기능입니다. 복소수 함수로 계산해야 할 때, 분지 절단에 대해 이해가 필요하다고 가정합니다. 이해를 위해서는 복소 변수에 관한 (너무 기초적이지 않은) 아무 책이나 참고하면 됩니다. 수치 계산의 목적으로 분지 절단을 적절히 선택하는 방법에 대한 정보에 대해서는, 다음과 같은 좋은 참고 문헌이 있습니다:

더 보기

Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothing' s sign bit. Iserles, A., and Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165–211.

9.4 `decimal` — Decimal fixed-point and floating-point arithmetic

소스 코드: Lib/decimal.py

The *decimal* module provides support for fast correctly rounded decimal floating-point arithmetic. It offers several advantages over the *float* datatype:

- Decimal "은 사람을 염두에 두고 설계된 부동 소수점 모델에 기반하고, 필연적으로 최고 원리를 갖습니다 – 컴퓨터는 사람들이 학교에서 배우는 산술과 같은 방식으로 동작하는 산술을 반드시 제공해야 한다." – 십진 산술 명세에서 발췌.
- Decimal numbers can be represented exactly. In contrast, numbers like `1.1` and `2.2` do not have exact representations in binary floating point. End users typically would not expect `1.1 + 2.2` to display as `3.3000000000000003` as it does with binary floating point.
- The exactness carries over into arithmetic. In decimal floating point, `0.1 + 0.1 + 0.1 - 0.3` is exactly equal to zero. In binary floating point, the result is `5.5511151231257827e-017`. While near to zero, the differences prevent reliable equality testing and differences can accumulate. For this reason, decimal is preferred in accounting applications which have strict equality invariants.
- The decimal module incorporates a notion of significant places so that `1.30 + 1.20` is `2.50`. The trailing zero is kept to indicate significance. This is the customary presentation for monetary applications. For multiplication, the "schoolbook" approach uses all the figures in the multiplicands. For instance, `1.3 * 1.2` gives `1.56` while `1.30 * 1.20` gives `1.5600`.
- 하드웨어 기반 이진 부동 소수점과는 달리, decimal 모듈은 사용자가 변경할 수 있는 정밀도(기본값은 28자리)를 가지며, 주어진 문제에 따라 필요한 만큼 커질 수 있습니다:

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- 이진 및 십진 부동 소수점 모두 출판된 표준에 따라 구현됩니다. 내장 float 형이 기능의 적당한 부분만을 드러내지만, decimal 모듈은 표준의 모든 필수 부분을 노출합니다. 필요한 경우, 프로그래머는 자리 올림(rounding) 및 신호(signal) 처리를 완전히 제어할 수 있습니다. 여기에는 정확하지 않은 연산을 차단하기 위한 예외를 사용하여 정확한 산술을 강제하는 옵션이 포함됩니다.

- decimal 모듈은 "편견 없이, (때로 고정 소수점 산술이라고도 불리는) 정확한 자리 올림 없는 십진 산술과 자리 올림 있는 부동 소수점 산술을 모두" 지원하도록 설계되었습니다. – 십진 산술 명세에서 발췌.

모듈 설계의 중심 개념은 세 가지입니다: 십진수, 산술을 위한 컨텍스트, 신호(signal).

A decimal number is immutable. It has a sign, coefficient digits, and an exponent. To preserve significance, the coefficient digits do not truncate trailing zeros. Decimals also include special values such as `Infinity`, `-Infinity`, and `NaN`. The standard also differentiates `-0` from `+0`.

산술 컨텍스트는 정밀도, 자리 올림 규칙, 지수에 대한 제한, 연산 결과를 나타내는 플래그 및 신호가 예외로 처리될지를 결정하는 트랩 활성화기(trap enabler)를 지정하는 환경입니다. 자리 올림 옵션에는 *ROUND_CEILING*, *ROUND_DOWN*, *ROUND_FLOOR*, *ROUND_HALF_DOWN*, *ROUND_HALF_EVEN*, *ROUND_HALF_UP*, *ROUND_UP* 및 *ROUND_05UP* 가 있습니다.

신호는 계산 과정에서 발생하는 예외적인 조건의 그룹입니다. 응용 프로그램의 필요에 따라, 신호가 무시되거나, 정보로 간주하거나, 예외로 처리될 수 있습니다. decimal 모듈의 신호는 *Clamped*, *InvalidOperation*, *DivisionByZero*, *Inexact*, *Rounded*, *Subnormal*, *Overflow*, *Underflow*, *FloatOperation* 입니다.

각 신호에는 플래그와 트랩 활성화기가 있습니다. 신호와 만났을 때, 플래그가 1로 설정되고 트랩 활성화기가 1로 설정된 경우, 예외가 발생합니다. 플래그는 상태가 유지되므로(sticky) 계산을 감시하기 전에 재설정할 필요가 있습니다.

더 보기

- IBM' s General Decimal Arithmetic Specification, The General Decimal Arithmetic Specification.

9.4.1 Quick-start tutorial

decimal을 사용하는 일반적인 시작은 모듈을 임포트하고, *getcontext()* 로 현재 컨텍스트를 보고, 필요하다면 정밀도, 자리 올림 또는 활성화된 트랩에 대해 새 값을 설정하는 것입니다:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7       # Set a new precision
```

Decimal instances can be constructed from integers, strings, floats, or tuples. Construction from an integer or a float performs an exact conversion of the value of that integer or float. Decimal numbers include special values such as `NaN` which stands for "Not a number", positive and negative `Infinity`, and `-0`:

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.14000000000000012434497875801753252744674682617187 5')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.4142135623730950488016887242')
>>> Decimal('NaN')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

FloatOperation 신호를 트랩 하는 경우, 실수로 생성자나 대소비교에서 Decimal 수와 실수(float)를 혼합하면 예외가 발생합니다:

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

Added in version 3.3.

새로운 Decimal의 유효 숫자는 입력된 숫자의 개수에 의해서만 결정됩니다. 컨텍스트 정밀도 및 자리올림은 오직 산술 연산 중에만 작용합니다.

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

C 버전의 내부 제한을 초과하면, Decimal 을 만들 때 *InvalidOperation* 를 일으킵니다:

```
>>> Decimal("1e999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

버전 3.3에서 변경.

Decimals interact well with much of the rest of Python. Here is a small decimal floating-point flying circus:

```
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split()))
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)
Decimal('1.3')
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

그리고 Decimal에는 몇 가지 수학 함수도 있습니다:

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')
```

The *quantize()* method rounds a number to a fixed exponent. This method is useful for monetary applications that often round results to a fixed number of places:

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

위에서 보듯이, *getcontext()* 함수는 현재 컨텍스트에 액세스하고 설정을 변경할 수 있게 합니다. 이 방법은 대부분 응용 프로그램의 요구를 충족시킵니다.

고급 작업을 위해, Context() 생성자를 사용하여 대체 컨텍스트를 만드는 것이 유용할 수 있습니다. 대체 컨텍스트를 활성화하려면, *setcontext()* 함수를 사용하십시오.

표준에 따라, *decimal* 모듈은 당장 사용할 수 있는 두 개의 표준 컨텍스트 *BasicContext* 와 *ExtendedContext* 를 제공합니다. 특히 전자는 많은 트랩이 활성화되어있어 디버깅에 유용합니다:

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

Contexts also have signal flags for monitoring exceptional conditions encountered during computations. The flags remain set until explicitly cleared, so it is best to clear the flags before each set of monitored computations by using the `clear_flags()` method.

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])
```

The *flags* entry shows that the rational approximation to pi was rounded (digits beyond the context precision were thrown away) and that the result is inexact (some of the discarded digits were non-zero).

Individual traps are set using the dictionary in the `traps` attribute of a context:

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

대부분 프로그램은 프로그램 시작 시에 한 번만 현재 컨텍스트를 조정합니다. 그리고, 많은 응용 프로그램에서, 데이터는 루프 내에서 단일형변환으로 `Decimal`로 변환되어, 프로그램 대부분은 다른 파이썬 숫자형과 별로 다르지 않게 데이터를 조작합니다.

9.4.2 Decimal 객체

class decimal.**Decimal**(*value='0'*, *context=None*)

value 를 기반으로 새 `Decimal` 객체를 만듭니다.

value 는 정수, 문자열, 튜플, `float` 또는 다른 `Decimal` 객체일 수 있습니다. *value* 가 주어지지 않으면, `Decimal('0')` 을 반환합니다. *value* 가 문자열이면, 앞뒤의 공백 문자 및 밑줄이 제거된 후 십진수 문자열 문법에 맞아야 합니다:

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

Other Unicode decimal digits are also permitted where `digit` appears above. These include decimal digits from various other alphabets (for example, Arabic-Indic and Devanāgarī digits) along with the fullwidth digits `'\uff10'` through `'\uff19'`. Case is not significant, so, for example, `inf`, `Inf`, `INFINITY`, and `iNfINity` are all acceptable spellings for positive infinity.

If *value* is a `tuple`, it should have three components, a sign (`0` for positive or `1` for negative), a `tuple` of digits, and an integer exponent. For example, `Decimal((0, (1, 4, 1, 4), -3))` returns `Decimal('1.414')`.

If *value* is a `float`, the binary floating-point value is losslessly converted to its exact decimal equivalent. This conversion can often require 53 or more digits of precision. For example, `Decimal(float('1.1'))` converts to `Decimal('1.1000000000000000888178419700125232338905334472656 25')`.

context 정밀도는 저장되는 자릿수에 영향을 주지 않습니다. 저장되는 자릿수는 *value* 의 자릿수만으로 결정됩니다. 예를 들어 `Decimal('3.00000')` 은 컨텍스트 정밀도가 단지 3이라도 5개의 모든 0을 기록합니다.

The purpose of the *context* argument is determining what to do if *value* is a malformed string. If the context traps `InvalidOperation`, an exception is raised; otherwise, the constructor returns a new Decimal with the value of `NaN`.

일단 만들어지면, `Decimal` 객체는 불변입니다.

버전 3.2에서 변경: 생성자에 대한 인자는 이제 `float` 인스턴스가 될 수 있습니다.

버전 3.3에서 변경: `float` 인자는 `FloatOperation` 트랩이 설정되면 예외를 발생시킵니다. 기본적으로 트랩은 꺼져 있습니다.

버전 3.6에서 변경: 코드에서의 정수와 부동 소수점 리터럴과 마찬가지로, 밑줄로 무리 지을 수 있습니다.

Decimal floating-point objects share many properties with the other built-in numeric types such as `float` and `int`. All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed, used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as `float` or `int`).

Decimal 객체에 대한 산술과 정수 및 실수에 대한 산술에는 약간의 차이가 있습니다. Decimal 객체에 나머지 연산자 `%` 가 적용될 때, 결과의 부호는 제수의 부호가 아닌 피제수의 부호가 됩니다:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

정수 나눗셈 연산자 `//` 의 동작 역시 비슷한 차이를 보입니다. 즉, 가장 가까운 정수로 내림하는 대신 실제 몫의 정수 부(0을 향해 자르기)를 돌려줍니다. 그래서 일반적인 항등식 `x == (x // y) * y + x % y` 를 유지합니다:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

`%` 와 `//` 연산자는 명세에 설명된 대로 각각 `remainder` 와 `divide-integer` 연산을 구현합니다.

Decimal 객체는 일반적으로 산술 연산에서 float 나 `fractions.Fraction` 인스턴스와 결합 할 수 없습니다: 예를 들어, `float` 에 a `Decimal`을 더하려고 하면 `TypeError` 를 일으킵니다. 그러나, 파이썬의 비교 연산자를 사용하여 `Decimal` 인스턴스 `x` 와 다른 숫자 `y` 를 비교할 수 있습니다. 이렇게 해서 서로 다른 형의 숫자 간에 동등 비교를 할 때 혼란스러운 결과를 피합니다.

버전 3.2에서 변경: `Decimal` 인스턴스와 다른 숫자 형 사이의 혼합형 비교가 이제 완전히 지원됩니다.

In addition to the standard numeric properties, decimal floating-point objects also have a number of specialized methods:

adjusted()

최상위 숫자만 남을 때까지 계수의 가장 오른쪽 숫자들을 밀어내도록 조정된 지수를 반환합니다. `Decimal('321e+5').adjusted()` 는 7을 반환합니다. 소수점으로부터의 최상위 유효 숫자의 위치를 결정하는 데 사용됩니다.

as_integer_ratio()

주어진 *Decimal* 인스턴스를, 분모가 양수인 기약 분수로 나타내는 정수의 쌍 `(n, d)` 을 돌려줍니다:

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

변환은 정확합니다. 무한대는 OverflowError를, NaN 은 ValueError를 일으킵니다.

Added in version 3.6.

as_tuple()

숫자의 네임드 튜플 표현을 반환합니다: `DecimalTuple(sign, digits, exponent)`.

canonical()

인자의 규범적인 인코딩을 돌려줍니다. 현재 *Decimal* 인스턴스의 인코딩은 항상 규범적이므로, 이 연산은 인자를 변경하지 않고 반환합니다.

compare(*other*, *context=None*)

두 Decimal 인스턴스의 값을 비교합니다. *compare()* 는 Decimal 인스턴스를 반환하고, 피연산자 중 하나가 NaN이면 결과는 NaN입니다:

```
a or b is a NaN  ==> Decimal('NaN')
a < b            ==> Decimal('-1')
a == b           ==> Decimal('0')
a > b            ==> Decimal('1')
```

compare_signal(*other*, *context=None*)

이 연산은, 모든 NaN 이 신호를 준다는 것을 제외하면 *compare()* 메서드와 같습니다. 즉, 피연산자가 모두 신호를 주는 NaN이 아니면, 모든 조용한 NaN 피연산자가 마치 신호를 주는 NaN인 것처럼 처리됩니다.

compare_total(*other*, *context=None*)

두 개의 피연산자를 숫자 값 대신 추상 표현을 사용하여 비교합니다. *compare()* 메서드와 비슷하지만, 결과는 *Decimal* 인스턴스에 대해 전 순서(total ordering)를 부여합니다. 같은 숫자 값을 갖지만 다른 표현의 두 *Decimal* 인스턴스는 이 순서에 의해 다른 것으로 비교됩니다:

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

조용한 NaN과 신호를 주는 NaN도 전 순서에 포함됩니다. 이 함수의 결과는, 두 피연산자가 같은 표현을 가질 때 `Decimal('0')`, 첫 번째 피연산자가 전 순서에서 두 번째 피연산자보다 낮으면 `Decimal('-1')`, 첫 번째 피연산자가 전 순서에서 두 번째 피연산자보다 높으면 `Decimal('1')` 입니다. 전 순서에 대한 세부 사항은 명세를 참조하십시오.

이 연산은 컨텍스트의 영향을 받지 않고, 조용합니다: 어떤 플래그도 변경되지 않고, 어떤 자리 올림도 수행되지 않습니다. 예외적으로, 두 번째 피연산자를 정확하게 변환할 수 없으면 C 버전은 InvalidOperation을 발생시킬 수 있습니다.

compare_total_mag(*other*, *context=None*)

*compare_total()*처럼 두 개의 피연산자를 숫자 값 대신 추상 표현을 사용하여 비교하지만, 각 피연산자의 부호를 무시합니다. `x.compare_total_mag(y)` 는 `x.copy_abs().compare_total(y.copy_abs())` 와 동등합니다.

이 연산은 컨텍스트의 영향을 받지 않고, 조용합니다: 어떤 플래그도 변경되지 않고, 어떤 자리 올림도 수행되지 않습니다. 예외적으로, 두 번째 피연산자를 정확하게 변환할 수 없으면 C 버전은 InvalidOperation을 발생시킬 수 있습니다.

conjugate()

그냥 self를 돌려줍니다. 이 메서드는 Decimal 명세를 준수하기 위한 것뿐입니다.

copy_abs()

인자의 절댓값을 반환합니다. 이 연산은 컨텍스트의 영향을 받지 않고, 조용합니다: 어떤 플래그도 변경되지 않고, 어떤 자리 올림도 수행되지 않습니다.

copy_negate()

인자의 음의 부정을 돌려줍니다. 이 연산은 컨텍스트의 영향을 받지 않고, 조용합니다: 어떤 플래그도 변경되지 않고, 어떤 자리 올림도 수행되지 않습니다.

copy_sign(*other*, *context=None*)

두 번째 피연산자의 부호와 같은 부호로 설정된 첫 번째 피연산자의 복사본을 반환합니다. 예를 들어:

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

이 연산은 컨텍스트의 영향을 받지 않고, 조용합니다: 어떤 플래그도 변경되지 않고, 어떤 자리 올림도 수행되지 않습니다. 예외적으로, 두 번째 피연산자를 정확하게 변환할 수 없으면 C 버전은 InvalidOperation을 발생시킬 수 있습니다.

exp(*context=None*)

주어진 숫자에 대한 (자연) 지수 함수 `e**x`의 값을 반환합니다. 결과는 *ROUND_HALF_EVEN* 자리 올림 모드를 사용하여 올바르게 자리 올림 됩니다.

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

classmethod from_float(*f*, /)

Alternative constructor that only accepts instances of *float* or *int*.

Note `Decimal.from_float(0.1)` is not the same as `Decimal('0.1')`. Since 0.1 is not exactly representable in binary floating point, the value is stored as the nearest representable value which is `0x1.999999999999ap-4`. That equivalent value in decimal is `0.1000000000000000055511151231257827021181583404541015625`.

> **참고**
>
> 파이썬 3.2 이후부터는, *Decimal* 인스턴스를 *float*에서 직접 생성할 수 있습니다.

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

Added in version 3.1.

classmethod from_number(*number*, /)

Alternative constructor that only accepts instances of *float*, *int* or *Decimal*, but not strings or tuples.

```
>>> Decimal.from_number(314)
Decimal('314')
>>> Decimal.from_number(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_number(Decimal('3.14'))
Decimal('3.14')
```

Added in version 3.14.

fma(*other*, *third*, *context=None*)

합성된 곱셈-덧셈(fused multiply-add). 중간값 self*other의 자리 올림 없이 self*other+third를 반환합니다.

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical()

인자가 규범적이면 *True*를 반환하고, 그렇지 않으면 *False*를 반환합니다. 현재 *Decimal* 인스턴스는 항상 규범적이므로 이 연산은 항상 *True*를 반환합니다.

is_finite()

인자가 유한 수이면 *True*를 반환하고, 인자가 무한대나 NaN 이면 *False*를 반환합니다.

is_infinite()

인자가 양이나 음의 무한대면 *True*를 반환하고, 그렇지 않으면 *False*를 반환합니다.

is_nan()

인자가 (조용한 또는 신호를 주는) NaN이면 *True*를 반환하고, 그렇지 않으면 *False*를 반환합니다.

is_normal(*context=None*)

인자가 정상(*normal*) 유한 수이면 *True*를 반환합니다. 인자가 0, 비정상(subnormal), 무한대 또는 NaN 이면 *False*를 반환합니다.

is_qnan()

인자가 조용한 NaN이면 *True*를 반환하고, 그렇지 않으면 *False*를 반환합니다.

is_signed()

인자가 음의 부호를 가지면 *True*를 반환하고, 그렇지 않으면 *False*를 반환합니다. 0과 NaN 모두 부호를 가질 수 있다는 것에 유의하세요.

is_snan()

인자가 신호를 주는 NaN이면 *True*를 반환하고, 그렇지 않으면 *False*를 반환합니다.

is_subnormal(*context=None*)

인자가 비정상(subnormal)이면 *True*를 반환하고, 그렇지 않으면 *False*를 반환합니다.

is_zero()

인자가 (양 또는 음의) 0이면 *True*를 반환하고, 그렇지 않으면 *False*를 반환합니다.

ln(*context=None*)

피연산자의 자연로그(밑 e)를 반환합니다. 결과는 *ROUND_HALF_EVEN* 자리 올림 모드를 사용하여 올바르게 반올림됩니다.

log10(*context=None*)

피연산자의 상용로그를 반환합니다. 결과는 *ROUND_HALF_EVEN* 자리 올림 모드를 사용하여 올바르게 반올림됩니다.

logb(*context=None*)

0이 아닌 수의 경우, 피연산자의 조정된 지수를 `Decimal` 인스턴스로 반환합니다. 피연산자가 0이면 `Decimal('-Infinity')` 가 반환되고 `DivisionByZero` 플래그가 발생합니다. 피연산자가 무한대면 `Decimal('Infinity')` 가 반환됩니다.

logical_and(*other*, *context=None*)

`logical_and()` 는 두 개의 논리적 피연산자(논리적 피연산자를 보세요)를 취하는 논리적 연산입니다. 결과는 두 피연산자의 자릿수별 `and` 입니다.

logical_invert(*context=None*)

`logical_invert()` 는 논리적 연산입니다. 결과는 피연산자의 자릿수별 반전입니다.

logical_or(*other*, *context=None*)

`logical_or()` 는 두 개의 논리적 피연산자(논리적 피연산자를 보세요)를 취하는 논리적 연산입니다. 결과는 두 피연산자의 자릿수별 `or` 입니다.

logical_xor(*other*, *context=None*)

`logical_xor()` 은 두 개의 논리적 피연산자(논리적 피연산자를 보세요)를 취하는 논리적 연산입니다. 결과는 두 피연산자의 자릿수별 배타적 or 입니다.

max(*other*, *context=None*)

Like `max(self, other)` except that the context rounding rule is applied before returning and that `NaN` values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

max_mag(*other*, *context=None*)

`max()` 와 비슷하지만, 피연산자의 절댓값을 사용하여 비교가 이루어집니다.

min(*other*, *context=None*)

Like `min(self, other)` except that the context rounding rule is applied before returning and that `NaN` values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

min_mag(*other*, *context=None*)

`min()` 과 비슷하지만, 피연산자의 절댓값을 사용하여 비교가 이루어집니다.

next_minus(*context=None*)

주어진 피연산자보다 작고, 주어진 컨텍스트(또는 context가 주어지지 않으면 현재 스레드의 컨텍스트)에서 표현 가능한 가장 큰 수를 돌려줍니다.

next_plus(*context=None*)

주어진 피연산자보다 크고, 주어진 컨텍스트(또는 context가 주어지지 않으면 현재 스레드의 컨텍스트)에서 표현 가능한 가장 작은 수를 돌려줍니다.

next_toward(*other*, *context=None*)

두 피연산자가 같지 않으면, 두 번째 피연산자의 방향으로 첫 번째 피연산자에 가장 가까운 숫자를 반환합니다. 두 피연산자가 수치로 같으면, 첫 번째 피연산자의 복사본을 반환하는데, 부호를 두 번째 피연산자의 것으로 설정합니다.

normalize(*context=None*)

Used for producing canonical values of an equivalence class within either the current context or the specified context.

This has the same semantics as the unary plus operation, except that if the final result is finite it is reduced to its simplest form, with all trailing zeros removed and its sign preserved. That is, while the coefficient is non-zero and a multiple of ten the coefficient is divided by ten and the exponent is incremented by 1. Otherwise (the coefficient is zero) the exponent is set to 0. In all cases the sign is unchanged.

For example, `Decimal('32.100')` and `Decimal('0.321000e+2')` both normalize to the equivalent value `Decimal('32.1')`.

Note that rounding is applied *before* reducing to simplest form.

In the latest versions of the specification, this operation is also known as `reduce`.

number_class(*context=None*)

피연산자의 클래스 를 설명하는 문자열을 반환합니다. 반환 값은 다음 10개의 문자열 중 하나입니다.

- `"-Infinity"`, 피연산자가 음의 무한대임을 나타냅니다.
- `"-Normal"`, 피연산자가 음의 정상 수임을 나타냅니다.
- `"-Subnormal"`, 피연산자가 음의 비정상 수임을 나타냅니다.
- `"-Zero"`, 피연산자가 음의 0임을 나타냅니다.
- `"+Zero"`, 피연산자가 양의 0임을 나타냅니다.
- `"+Subnormal"`, 피연산자가 양의 비정상 수임을 나타냅니다.
- `"+Normal"`, 피연산자가 양의 정상 수임을 나타냅니다.
- `"+Infinity"`, 피연산자가 양의 무한대임을 나타냅니다.
- `"NaN"`, 피연산자가 조용한 NaN(Not a Number) 임을 나타냅니다.
- `"sNaN"`, 피연산자가 신호를 주는 NaN 임을 나타냅니다.

quantize(*exp*, *rounding=None*, *context=None*)

자리 올림 후에 첫 번째 피연산자와 같고 두 번째 피연산자의 지수를 갖는 값을 반환합니다.

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

다른 연산과 달리, quantize 연산 후의 계수의 길이가 정밀도보다 크면, *InvalidOperation* 신호를 줍니다. 이는, 에러 조건이 없으면, quantize 된 지수가 항상 오른쪽 피연산자의 지수와 같음을 보장합니다.

또한, 다른 연산과는 달리, 결과가 비정상(subnormal) 이고 부정확한 경우조차도, quantize는 결코 Underflow 신호를 보내지 않습니다.

두 번째 피연산자의 지수가 첫 번째 피연산자의 지수보다 크면 자리 올림이 필요할 수 있습니다. 이 경우, 자리 올림 모드는 (주어지면) `rounding` 인자에 의해 결정됩니다. 그렇지 않으면 주어진 `context` 인자에 의해 결정됩니다; 두 인자 모두 주어지지 않으면, 현재 스레드의 컨텍스트의 자리 올림 모드가 사용됩니다.

An error is returned whenever the resulting exponent is greater than *Emax* or less than *Etiny()*.

radix()

Decimal 클래스가 모든 산술을 수행하는 진수(기수) 인 `Decimal(10)` 을 반환합니다. 명세와의 호환성을 위해 포함됩니다.

remainder_near(*other*, *context=None*)

self 를 *other* 로 나눈 나머지를 반환합니다. 이것은 나머지의 절댓값을 최소화하기 위해 나머지의 부호가 선택된다는 점에서 `self % other` 와 다릅니다. 좀 더 정확히 말하면, 반환 값은 `self - n * other` 인데, 여기서 `n` 은 `self / other` 의 정확한 값에 가장 가까운 정수이고, 두 개의 정수와의 거리가 같으면 짝수가 선택됩니다.

결과가 0이면 그 부호는 *self* 의 부호가 됩니다.

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(*other*, *context=None*)

첫 번째 피연산자의 계수를 두 번째 피연산자로 지정된 양만큼 회전한 결과를 반환합니다. 두 번째 피연산자는 -precision에서 precision 범위의 정수여야 합니다. 두 번째 피연산자의 절댓값은 회전할 자리의 수를 나타냅니다. 두 번째 피연산자가 양수면 왼쪽으로 회전합니다; 그렇지 않으면 오른쪽으로 회전합니다. 필요하면 정밀도에 맞추기 위해 첫 번째 피연산자의 계수에 0이 왼쪽에 채워집니다. 첫 번째 피연산자의 부호와 지수는 변경되지 않습니다.

same_quantum(*other*, *context=None*)

Test whether self and other have the same exponent or whether both are `NaN`.

이 연산은 컨텍스트의 영향을 받지 않고, 조용합니다: 어떤 플래그도 변경되지 않고, 어떤 자리 올림도 수행되지 않습니다. 예외적으로, 두 번째 피연산자를 정확하게 변환할 수 없으면 C 버전은 InvalidOperation을 발생시킬 수 있습니다.

scaleb(*other*, *context=None*)

첫 번째 피연산자의 지수를 두 번째 피연산자만큼 조정한 값을 반환합니다. 달리 표현하면, 첫 번째 피연산자에 `10**other` 를 곱한 값을 반환합니다. 두 번째 피연산자는 정수여야 합니다.

shift(*other*, *context=None*)

첫 번째 피연산자의 계수를 두 번째 피연산자로 지정된 양만큼 이동한 결과를 반환합니다. 두 번째 피연산자는 -precision에서 precision 범위의 정수여야 합니다. 두 번째 피연산자의 절댓값은 이동할 자리의 수를 나타냅니다. 두 번째 피연산자가 양수면 왼쪽으로 이동합니다; 그렇지 않으면 오른쪽으로 이동합니다. 이동으로 인해 계수에 들어오는 숫자는 0입니다. 첫 번째 피연산자의 부호와 지수는 변경되지 않습니다.

sqrt(*context=None*)

인자의 제곱근을 완전한 정밀도로 반환합니다.

to_eng_string(*context=None*)

문자열로 변환합니다. 지수가 필요하면 공학 표기법을 사용합니다.

공학 표기법의 지수는 3의 배수입니다. 이렇게 하면 소수점 왼쪽에 최대 3자리를 남기게 되고, 하나나 두 개의 후행 0을 추가해야 할 수 있습니다.

예를 들어, 이 메서드는 `Decimal('123E+1')` 을 `Decimal('1.23E+3')` 으로 변환합니다.

to_integral(*rounding=None*, *context=None*)

`to_integral_value()` 메서드와 같습니다. `to_integral` 이름은 이전 버전과의 호환성을 위해 유지되었습니다.

to_integral_exact(*rounding=None*, *context=None*)

`Inexact` 나 *`Rounded`* 신호를 주면서 가장 가까운 정수로 자리 올림 합니다. 자리 올림 모드는 (주어지면) `rounding` 매개 변수에 의해, 그렇지 않으면 그렇지 않으면 `context` 에 의해 결정됩니다. 두 매개 변수 모두 지정되지 않으면, 현재 컨텍스트의 자리 올림 모드가 사용됩니다.

to_integral_value(*rounding=None*, *context=None*)

`Inexact` 나 *`Rounded`* 신호를 주지 않고 가장 가까운 정수로 자리 올림 합니다. 주어지면, *rounding* 을 적용합니다; 그렇지 않으면, 제공된 *context* 나 현재 컨텍스트의 자리 올림 방법을 사용합니다.

Decimal numbers can be rounded using the *`round()`* function:

round(number)

round(number, ndigits)

If *ndigits* is not given or `None`, returns the nearest *`int`* to *number*, rounding ties to even, and ignoring the rounding mode of the *`Decimal`* context. Raises *`OverflowError`* if *number* is an infinity or *`ValueError`* if it is a (quiet or signaling) NaN.

If *ndigits* is an *`int`*, the context' s rounding mode is respected and a *`Decimal`* representing *number* rounded to the nearest multiple of `Decimal('1E-ndigits')` is returned; in this case, `round(number, ndigits)` is equivalent to `self.quantize(Decimal('1E-ndigits'))`. Returns `Decimal('NaN')` if *number* is a quiet NaN. Raises *`InvalidOperation`* if *number* is an infinity, a signaling NaN, or if the length of the coefficient after the quantize operation would be greater than the current context' s precision. In other words, for the non-corner cases:

- if *ndigits* is positive, return *number* rounded to *ndigits* decimal places;
- if *ndigits* is zero, return *number* rounded to the nearest integer;
- if *ndigits* is negative, return *number* rounded to the nearest multiple of `10**abs(ndigits)`.

For example:

```
>>> from decimal import Decimal, getcontext, ROUND_DOWN
>>> getcontext().rounding = ROUND_DOWN
>>> round(Decimal('3.75'))      # context rounding ignored
4
>>> round(Decimal('3.5'))       # round-ties-to-even
4
>>> round(Decimal('3.75'), 0)   # uses the context rounding
Decimal('3')
>>> round(Decimal('3.75'), 1)
Decimal('3.7')
>>> round(Decimal('3.75'), -1)
Decimal('0E+1')
```

논리적 피연산자

The `logical_and()`, `logical_invert()`, `logical_or()`, and `logical_xor()` methods expect their arguments to be *logical operands*. A *logical operand* is a `Decimal` instance whose exponent and sign are both zero, and whose digits are all either `0` or `1`.

9.4.3 Context 객체

컨텍스트는 산술 연산을 위한 환경입니다. 정밀도를 제어하고, 자리 올림 규칙을 설정하며, 어떤 신호가 예외로 처리되는지 결정하고, 지수의 범위를 제한합니다.

각 스레드는 자신만의 현재 컨텍스트를 가지는데, `getcontext()` 와 `setcontext()` 함수를 사용하여 액세스하거나 변경합니다:

`decimal.`**`getcontext`**`()`

활성 스레드의 현재 컨텍스트를 돌려줍니다.

`decimal.`**`setcontext`**`(`*c*, /`)`

활성 스레드의 현재 컨텍스트를 *c* 로 설정합니다.

또한 `with` 문과 `localcontext()` 함수를 사용하여 활성 컨텍스트를 일시적으로 변경할 수 있습니다.

`decimal.`**`localcontext`**`(`*ctx=None*, ***kwargs*`)`

Return a context manager that will set the current context for the active thread to a copy of *ctx* on entry to the with-statement and restore the previous context when exiting the with-statement. If no context is specified, a copy of the current context is used. The *kwargs* argument is used to set the attributes of the new context.

예를 들어, 다음 코드는 현재 십진 정밀도를 42자리로 설정하고, 계산을 수행한 다음, 이전 컨텍스트를 자동으로 복원합니다:

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # Perform a high precision calculation
    s = calculate_something()
s = +s  # Round the final result back to the default precision
```

Using keyword arguments, the code would be the following:

```
from decimal import localcontext

with localcontext(prec=42) as ctx:
    s = calculate_something()
s = +s
```

Raises *TypeError* if *kwargs* supplies an attribute that *Context* doesn' t support. Raises either *TypeError* or *ValueError* if *kwargs* supplies an invalid value for an attribute.

버전 3.11 에서 변경: *localcontext()* now supports setting context attributes through the use of keyword arguments.

decimal.**IEEEContext**(*bits*)

Return a context object initialized to the proper values for one of the IEEE interchange formats. The argument must be a multiple of 32 and less than *IEEE_CONTEXT_MAX_BITS*.

Added in version 3.14.

아래에 설명된 *Context* 생성자를 사용하여 새로운 컨텍스트를 만들 수도 있습니다. 또한, 이 모듈은 세 가지 미리 만들어진 컨텍스트를 제공합니다:

decimal.**BasicContext**

이것은 일반 십진 산술 명세에서 정의된 표준 컨텍스트입니다. 정밀도는 9로 설정됩니다. 자리 올림은 *ROUND_HALF_UP*으로 설정됩니다. 모든 플래그가 지워집니다. 모든 트랩은 *Inexact*, *Rounded*, *Subnormal*을 제외하고는 활성화됩니다 (예외로 처리됩니다).

많은 트랩이 활성화되었으므로, 이 컨텍스트는 디버깅에 유용합니다.

decimal.**ExtendedContext**

이것은 일반 십진 산술 명세에서 정의된 표준 컨텍스트입니다. 정밀도는 9로 설정됩니다. 자리 올림은 *ROUND_HALF_EVEN*으로 설정됩니다. 모든 플래그가 지워집니다. 아무 트랩도 활성화되지 않습니다 (그래서 계산 중에 예외가 발생하지 않습니다).

Because the traps are disabled, this context is useful for applications that prefer to have result value of `NaN` or `Infinity` instead of raising exceptions. This allows an application to complete a run in the presence of conditions that would otherwise halt the program.

decimal.**DefaultContext**

이 컨텍스트는 새로운 컨텍스트의 프로토타입으로 *Context* 생성자에 의해 사용됩니다. 필드(가령 정밀도)를 변경하면 *Context* 생성자에 의해 생성된 새로운 컨텍스트에 대한 기본값을 변경하는 효과가 있습니다.

이 컨텍스트는 다중 스레드 환경에서 가장 유용합니다. 스레드가 시작되기 전에 필드 중 하나를 변경하면 시스템 전체의 기본값을 설정하는 효과가 있습니다. 스레드가 시작된 후에 필드를 변경하는 것은. 스레드 동기화를 통해 경쟁 조건을 방지해야 하므로 권장되지 않습니다.

단일 스레드 환경에서는, 이 컨텍스트를 아예 사용하지 않는 것이 좋습니다. 대신, 아래에 설명된 대로 명시적으로 컨텍스트를 만드십시오.

The default values are *Context.prec*=28, *Context.rounding*=*ROUND_HALF_EVEN*, and enabled traps for *Overflow*, *InvalidOperation*, and *DivisionByZero*.

3 개의 제공된 컨텍스트 외에도, 새로운 컨텍스트를 *Context* 생성자를 사용하여 만들 수 있습니다.

class decimal.**Context**(*prec=None, rounding=None, Emin=None, Emax=None, capitals=None, clamp=None, flags=None, traps=None*)

새로운 컨텍스트를 만듭니다. 필드가 지정되지 않았거나 *None* 이면, 기본값은 *DefaultContext* 에서 복사됩니다. *flags* 필드가 지정되지 않았거나 *None* 이면, 모든 플래그가 지워집니다.

prec

An integer in the range [1, *MAX_PREC*] that sets the precision for arithmetic operations in the context.

rounding

One of the constants listed in the section *Rounding Modes*.

traps

flags

Lists of any signals to be set. Generally, new contexts should only set traps and leave the flags clear.

Emin

Emax

Integers specifying the outer limits allowable for exponents. *Emin* must be in the range [`MIN_EMIN`, `0`], *Emax* in the range [`0`, `MAX_EMAX`].

capitals

Either `0` or `1` (the default). If set to `1`, exponents are printed with a capital `E`; otherwise, a lowercase `e` is used: `Decimal('6.02e+23')`.

clamp

Either `0` (the default) or `1`. If set to `1`, the exponent `e` of a `Decimal` instance representable in this context is strictly limited to the range `Emin - prec + 1 <= e <= Emax - prec + 1`. If *clamp* is `0` then a weaker condition holds: the adjusted exponent of the `Decimal` instance is at most `Emax`. When *clamp* is `1`, a large normal number will, where possible, have its exponent reduced and a corresponding number of zeros added to its coefficient, in order to fit the exponent constraints; this preserves the value of the number but loses information about significant trailing zeros. For example:

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

A *clamp* value of `1` allows compatibility with the fixed-width decimal interchange formats specified in IEEE 754.

The `Context` class defines several general purpose methods as well as a large number of methods for doing arithmetic directly in a given context. In addition, for each of the `Decimal` methods described above (with the exception of the `adjusted()` and `as_tuple()` methods) there is a corresponding `Context` method. For example, for a `Context` instance `C` and `Decimal` instance `x`, `C.exp(x)` is equivalent to `x.exp(context=C)`. Each `Context` method accepts a Python integer (an instance of `int`) anywhere that a Decimal instance is accepted.

clear_flags()

Resets all of the flags to `0`.

clear_traps()

Resets all of the traps to `0`.

Added in version 3.3.

copy()

컨텍스트의 복사본을 돌려줍니다.

copy_decimal(*num*, /)

Decimal 인스턴스 num의 복사본을 반환합니다.

create_decimal(*num='0'*, /)

self 를 컨텍스트로 사용해서, *num* 으로 새 Decimal 인스턴스를 만듭니다. `Decimal` 생성자와 달리, 컨텍스트 정밀도, 자리 올림 방법, 플래그 및 트랩이 변환에 적용됩니다.

이는 상수가 보통 응용 프로그램에 필요한 것보다 더 큰 정밀도로 제공되기 때문에 유용합니다. 또 다른 이점은 자리 올림이 현재 정밀도를 초과하는 자릿수로 인한 의도하지 않은 결과를 즉시 제거한다는 것입니다. 다음 예제에서, 자리 올림 되지 않은 입력을 사용한다는 것은 합계에 0을 추가하면 결과가 달라질 수 있음을 의미합니다.:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

이 메서드는 IBM 명세의 to-number 연산을 구현합니다. 인자가 문자열이면, 선행 또는 후행 공백이나 밑줄이 허용되지 않습니다.

create_decimal_from_float(*f*, /)

float *f* 로 새 Decimal 인스턴스를 만들지만, *self* 를 컨텍스트로 사용하여 자리 올림 합니다. *Decimal.from_float()* 클래스 메서드와는 달리, 컨텍스트 정밀도, 자리 올림 방법, 플래그 및 트랩이 변환에 적용됩니다.

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
    ...
decimal.Inexact: None
```

Added in version 3.1.

Etiny()

비정상 결과에 대한 최소 지수 값인 `Emin - prec + 1` 과 같은 값을 반환합니다. 언더 플로우가 발생하면, 지수는 *Etiny* 로 설정됩니다.

Etop()

`Emax - prec + 1` 과 같은 값을 반환합니다.

십진수로 작업하는 일반적인 접근법은 *Decimal* 인스턴스를 생성한 다음 활성 스레드의 현재 컨텍스트 내에서 진행되는 산술 연산을 적용하는 것입니다. 다른 방법은 특정 컨텍스트 내에서 계산하기 위해 컨텍스트 메서드를 사용하는 것입니다. 메서드는 *Decimal* 클래스의 메서드와 비슷하며 여기에서는 간단히 설명합니다.

abs(*x*, /)

x 의 절댓값을 돌려줍니다.

add(*x*, *y*, /)

x 와 *y* 의 합을 돌려줍니다.

canonical(*x*, /)

같은 Decimal 객체 *x* 를 반환합니다.

compare(*x*, *y*, /)

x 와 *y* 를 수치로 비교합니다.

compare_signal(*x*, *y*, /)

두 피연산자의 값을 수치로 비교합니다.

compare_total(*x*, *y*, /)

추상 표현을 사용하여 두 피연산자를 비교합니다.

compare_total_mag(*x*, *y*, /)

부호를 무시하고, 추상 표현을 사용하여 두 피연산자를 비교합니다.

copy_abs(*x*, /)

부호가 0으로 설정되어있는 *x* 의 복사본을 돌려줍니다.

copy_negate (*x*, /)

부호가 반전된 *x* 복사본을 반환합니다.

copy_sign (*x*, *y*, /)

y 에서 *x* 로 부호를 복사합니다.

divide (*x*, *y*, /)

x 를 *y* 로 나눈 값을 반환합니다.

divide_int (*x*, *y*, /)

x 를 *y* 로 나눈 후 정수로 잘라낸 값을 반환합니다.

divmod (*x*, *y*, /)

두 숫자를 나누고 결과의 정수 부분을 반환합니다.

exp (*x*, /)

Returns `e ** x`.

fma (*x*, *y*, *z*, /)

x 에 *y* 를 곱한 후 *z* 를 더한 값을 반환합니다.

is_canonical (*x*, /)

x 가 규범적일 경우 `True`를 반환합니다; 그렇지 않으면 `False`를 반환합니다.

is_finite (*x*, /)

x 가 유한이면 `True`를 반환합니다; 그렇지 않으면 `False`를 반환합니다.

is_infinite (*x*, /)

x 가 무한대면 `True`를 반환합니다; 그렇지 않으면 `False`를 반환합니다.

is_nan (*x*, /)

x 가 qNaN 이나 sNaN 이면 `True`를 반환합니다; 그렇지 않으면 `False`를 반환합니다.

is_normal (*x*, /)

x 가 정상 수면 `True`를 반환합니다; 그렇지 않으면 `False`를 반환합니다.

is_qnan (*x*, /)

x 가 조용한 NaN 이면 `True`를 반환합니다; 그렇지 않으면 `False`를 반환합니다.

is_signed (*x*, /)

x 가 음수면 `True`를 반환합니다; 그렇지 않으면 `False`를 반환합니다.

is_snan (*x*, /)

x 가 신호를 주는 NaN 이면 `True`를 반환합니다; 그렇지 않으면 `False`를 반환합니다.

is_subnormal (*x*, /)

x 가 비정상이면 `True`를 반환합니다; 그렇지 않으면 `False`를 반환합니다.

is_zero (*x*, /)

x 가 0 이면 `True`를 반환합니다; 그렇지 않으면 `False`를 반환합니다.

ln (*x*, /)

x 의 자연로그(밑 e)를 반환합니다.

log10 (*x*, /)

x 의 상용로그를 반환합니다.

logb (*x*, /)

피연산자의 최상위 유효 숫자의 크기의 지수를 반환합니다.

logical_and (*x*, *y*, /)

각 피연산자의 자릿수별로 논리적 연산 *and* 를 적용합니다.

logical_invert(*x*, /)

x 의 모든 자릿수를 반전합니다.

logical_or(*x*, *y*, /)

각 피연산자의 자릿수별로 논리적 연산 *or* 를 적용합니다.

logical_xor(*x*, *y*, /)

각 피연산자의 자릿수별로 논리적 연산 *xor* 를 적용합니다.

max(*x*, *y*, /)

두 값을 수치로 비교해, 최댓값을 돌려줍니다.

max_mag(*x*, *y*, /)

부호를 무시하고 값을 수치로 비교합니다.

min(*x*, *y*, /)

두 값을 수치로 비교해, 최솟값을 돌려줍니다.

min_mag(*x*, *y*, /)

부호를 무시하고 값을 수치로 비교합니다.

minus(*x*, /)

minus는 파이썬에서 단항 접두사 빼기 연산자에 해당합니다.

multiply(*x*, *y*, /)

x 와 *y* 의 곱을 반환합니다.

next_minus(*x*, /)

x 보다 작고 표현 가능한 가장 큰 수를 반환합니다.

next_plus(*x*, /)

x 보다 크고 표현 가능한 가장 작은 수를 반환합니다.

next_toward(*x*, *y*, /)

y 방향으로 *x* 에 가장 가까운 숫자를 반환합니다.

normalize(*x*, /)

x 를 가장 간단한 형태로 환원합니다.

number_class(*x*, /)

x 의 클래스를 가리키는 문자열을 돌려줍니다.

plus(*x*, /)

plus는 파이썬에서 단항 접두사 더하기 연산자에 해당합니다. 이 연산은 컨텍스트 정밀도와 자리 올림을 적용하므로 항등 연산이 아닙니다.

power(*x*, *y*, *modulo=None*)

`x` 의 `y` 거듭제곱을 돌려줍니다. 주어지면 `modulo` 모듈로로 환원합니다.

With two arguments, compute `x**y`. If `x` is negative then `y` must be integral. The result will be inexact unless `y` is integral and the result is finite and can be expressed exactly in 'precision' digits. The rounding mode of the context is used. Results are always correctly rounded in the Python version.

`Decimal(0) ** Decimal(0)` 은 `InvalidOperation` 이 되며, `InvalidOperation` 가 트랩 되지 않으면, `Decimal('NaN')` 이 됩니다.

버전 3.3에서 변경: The C module computes `power()` in terms of the correctly rounded `exp()` and `ln()` functions. The result is well-defined but only "almost always correctly rounded".

세 인자로는 `(x**y) % modulo` 를 계산합니다. 세 인자 형식의 경우, 인자에 다음과 같은 제한이 있습니다:

- 세 인자는 모두 정수여야 합니다.
- `y` 는 음수가 아니어야 합니다.

- `x` 나 `y` 중 적어도 하나는 0이 아니어야 합니다
- `modulo` 는 0이 아니고 최대 'precision' 자릿수를 가져야 합니다

`Context.power(x, y, modulo)` 의 결괏값은 무한 정밀도로 `(x**y) % modulo` 를 계산할 때 얻을 수 있는 값과 같지만, 더 효율적으로 계산됩니다. 결과의 지수는 `x`, `y` 및 `modulo` 의 지수와 관계없이 0입니다. 결과는 항상 정확합니다.

quantize(*x*, *y*, /)

y 의 지수를 가지는 (자리 올림 된) *x* 와 같은 값을 반환합니다.

radix()

Decimal이기 때문에 단지 10을 반환합니다, :)

remainder(*x*, *y*, /)

정수 나눗셈의 나머지를 반환합니다.

결과가 0이 아닐 때, 결과의 부호는 원래의 피제수와 같습니다.

remainder_near(*x*, *y*, /)

`x - y * n` 을 반환하는데, *n* 은 `x / y` 의 정확한 값에 가장 가까운 정수입니다 (결과가 0이면 그 부호는 *x* 의 부호가 됩니다).

rotate(*x*, *y*, /)

x 를 *y* 번 회전한 복사본을 반환합니다.

same_quantum(*x*, *y*, /)

두 피연산자의 지수가 같으면 `True`를 반환합니다.

scaleb(*x*, *y*, /)

첫 번째 피연산자의 지수에 두 번째 값을 더해서 반환합니다.

shift(*x*, *y*, /)

x 를 *y* 번 이동한 복사본을 반환합니다.

sqrt(*x*, /)

음이 아닌 수의 제곱근을 컨텍스트의 정밀도로 반환합니다.

subtract(*x*, *y*, /)

x 와 *y* 의 차를 돌려줍니다.

to_eng_string(*x*, /)

문자열로 변환합니다. 지수가 필요하면 공학 표기법을 사용합니다.

공학 표기법의 지수는 3의 배수입니다. 이렇게 하면 소수점 왼쪽에 최대 3자리를 남기게 되고, 하나나 두 개의 후행 0을 추가해야 할 수 있습니다.

to_integral_exact(*x*, /)

정수로 자리 올림 합니다.

to_sci_string(*x*, /)

과학 표기법을 사용하여 숫자를 문자열로 변환합니다.

9.4.4 상수

이 절의 상수는 C 모듈에서만 의미가 있습니다. 호환성을 위해 순수 파이썬 버전에도 포함되어 있습니다.

	32-비트	64-비트
decimal.**MAX_PREC**	425000000	999999999999999999
decimal.**MAX_EMAX**	425000000	999999999999999999
decimal.**MIN_EMIN**	-425000000	-999999999999999999
decimal.**MIN_ETINY**	-849999999	-1999999999999999997
decimal.**IEEE_CONTEXT_MAX_BITS**	256	512

decimal.**HAVE_THREADS**

값은 `True`입니다. 이제 파이썬에는 항상 스레드가 있기 때문에, 폐지되었습니다.

버전 3.9부터 폐지됨.

decimal.**HAVE_CONTEXTVAR**

The default value is `True`. If Python is `configured using the --without-decimal-contextvar option`, the C version uses a thread-local rather than a coroutine-local context and the value is `False`. This is slightly faster in some nested context scenarios.

Added in version 3.8.3.

9.4.5 자리 올림 모드

decimal.**ROUND_CEILING**

Round towards `Infinity`.

decimal.**ROUND_DOWN**

0을 향해 자리 올림 합니다.

decimal.**ROUND_FLOOR**

Round towards `-Infinity`.

decimal.**ROUND_HALF_DOWN**

가장 가까운 값으로 반올림하고, 동률이면 0에서 가까운 것을 선택합니다.

decimal.**ROUND_HALF_EVEN**

가장 가까운 값으로 반올림하고, 동률이면 짝수를 선택합니다.

decimal.**ROUND_HALF_UP**

가장 가까운 값으로 반올림하고, 동률이면 0에서 먼 것을 선택합니다.

decimal.**ROUND_UP**

0에서 먼 쪽으로 자리 올림 합니다.

decimal.**ROUND_05UP**

0을 향해 자리 올림 했을 때 마지막 숫자가 0이나 5면 0에서 먼 쪽으로 자리 올림 합니다. 그렇지 않으면 0을 향해 자리 올림 합니다.

9.4.6 신호

신호는 계산 중 발생하는 조건을 나타냅니다. 각각은 하나의 컨텍스트 플래그와 하나의 컨텍스트 트랩 활성화기에 대응합니다.

컨텍스트 플래그는 조건이 발생할 때마다 설정됩니다. 계산 후에, 플래그는 정보를 얻기 위한 목적으로 확인될 수 있습니다 (예를 들어, 계산이 정확한지를 판별하기 위해). 플래그를 확인한 후 다음 계산을 시작하기 전에 모든 플래그를 지우십시오.

컨텍스트의 트랩 활성화기가 신호에 대해 설정되면, 조건은 파이썬 예외를 일으킵니다. 예를 들어, *DivisionByZero* 트랩이 설정되면, 이 조건을 만날 때 *DivisionByZero* 예외가 발생합니다.

class decimal.**Clamped**

표현 제약 조건에 맞도록 지수를 변경했습니다.

Typically, clamping occurs when an exponent falls outside the context' s *Emin* and *Emax* limits. If possible, the exponent is reduced to fit by adding zeros to the coefficient.

class decimal.**DecimalException**

다른 신호의 베이스 클래스이고 *ArithmeticError* 의 서브 클래스입니다.

class decimal.**DivisionByZero**

무한대가 아닌 숫자를 0으로 나눴다는 신호를 줍니다.

Can occur with division, modulo division, or when raising a number to a negative power. If this signal is not trapped, returns `Infinity` or `-Infinity` with the sign determined by the inputs to the calculation.

class decimal.**Inexact**

자리 올림이 발생했고 결과가 정확하지 않음을 나타냅니다.

자리 올림 도중 0이 아닌 숫자가 삭제된 경우 신호를 줍니다. 자리 올림 된 결과가 반환됩니다. 신호 플래그나 트랩은 결과가 정확하지 않을 때를 감지하는 데 사용됩니다.

class decimal.**InvalidOperation**

유효하지 않은 연산이 수행되었습니다.

Indicates that an operation was requested that does not make sense. If not trapped, returns `NaN`. Possible causes include:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class decimal.**Overflow**

수치적 오버플로.

Indicates the exponent is larger than *Context.Emax* after rounding has occurred. If not trapped, the result depends on the rounding mode, either pulling inward to the largest representable finite number or rounding outward to `Infinity`. In either case, *Inexact* and *Rounded* are also signaled.

class decimal.**Rounded**

정보가 손실되지는 않았지만 자리 올림이 발생했습니다.

Signaled whenever rounding discards digits; even if those digits are zero (such as rounding `5.00` to `5.0`). If not trapped, returns the result unchanged. This signal is used to detect loss of significant digits.

class decimal.**Subnormal**

Exponent was lower than *Emin* prior to rounding.

연산 결과가 비정상(지수가 너무 작음)일 때 발생합니다. 트랩 되지 않으면, 결과를 그대로 반환합니다.

class decimal.**Underflow**

결과가 0으로 자리 올림 되는 수치적 언더플로.

자리 올림에 의해 비정상 결과가 0으로 밀릴 때 발생합니다. *Inexact*와 *Subnormal* 신호도 줍니다.

class decimal.**FloatOperation**

float와 Decimal을 혼합하는 데 더 엄격한 의미를 사용합니다.

신호가 트랩되지 않으면 (기본값), *Decimal* 생성자, *create_decimal()* 및 모든 비교 연산자에서 float와 Decimal을 혼합 할 수 있습니다. 변환과 비교 모두 정확합니다. 복합 연산의 발생은 컨텍스트 플래그에 *FloatOperation* 을 설정하여 조용히 기록됩니다. *from_float()* 나 *create_decimal_from_float()* 를 사용한 명시적 변환은 플래그를 설정하지 않습니다.

그렇지 않으면 (신호가 트랩되면), 같음 비교와 명시적 변환만 조용히 수행됩니다. 다른 모든 혼합된 연산은 *FloatOperation* 을 발생시킵니다.

다음 표는 신호의 계층 구조를 요약한 것입니다:

```
exceptions.ArithmeticError(exceptions.Exception)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
        FloatOperation(DecimalException, exceptions.TypeError)
```

9.4.7 Floating-point notes

증가시킨 정밀도로 자리 올림 오차 줄이기

The use of decimal floating point eliminates decimal representation error (making it possible to represent `0.1` exactly); however, some operations can still incur round-off error when non-zero digits exceed the fixed precision.

The effects of round-off error can be amplified by the addition or subtraction of nearly offsetting quantities resulting in loss of significance. Knuth provides two instructive examples where rounded floating-point arithmetic with insufficient precision causes the breakdown of the associative and distributive properties of addition:

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> u * (v+w)
Decimal('0.0060000')
```

decimal 모듈은 유효숫자의 손실을 피할 수 있을 만큼 정밀도를 확장함으로써 항등 관계를 복구할 수 있게 합니다:

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

특수 값

The number system for the *decimal* module provides special values including `NaN`, `sNaN`, `-Infinity`, `Infinity`, and two zeros, `+0` and `-0`.

무한대는 다음과 같이 직접 생성될 수 있습니다: `Decimal('Infinity')`. 또한, *DivisionByZero* 신호가 트랩 되지 않을 때 0으로 나눠서 발생할 수 있습니다. 마찬가지로, *Overflow* 신호가 트랩 되지 않을 때, 무한대는 표현 가능한 가장 큰 수의 한계를 넘어서 자리 올림 된 결과가 될 수 있습니다.

무한대는 부호가 있고 (아핀) 산술 연산에 사용될 수 있는데, 매우 크고 불확정적(indeterminate) 인 숫자로 취급됩니다. 예를 들어, 무한대에 상수를 더하면 또 다른 무한대를 줍니다.

Some operations are indeterminate and return `NaN`, or if the *InvalidOperation* signal is trapped, raise an exception. For example, `0/0` returns `NaN` which means "not a number". This variety of `NaN` is quiet and, once created, will flow through other computations always resulting in another `NaN`. This behavior can be useful for a series of computations that occasionally have missing inputs — it allows the calculation to proceed while flagging specific results as invalid.

A variant is `sNaN` which signals rather than remaining quiet after every operation. This is a useful return value when an invalid result needs to interrupt a calculation for special handling.

The behavior of Python's comparison operators can be a little surprising where a `NaN` is involved. A test for equality where one of the operands is a quiet or signaling `NaN` always returns *False* (even when doing `Decimal('NaN')==Decimal('NaN')`), while a test for inequality always returns *True*. An attempt to compare two Decimals using any of the <, <=, > or >= operators will raise the *InvalidOperation* signal if either operand is a `NaN`, and return *False* if this signal is not trapped. Note that the General Decimal Arithmetic specification does not specify the behavior of direct comparisons; these rules for comparisons involving a `NaN` were taken from the IEEE 854 standard (see Table 3 in section 5.7). To ensure strict standards-compliance, use the *compare()* and *compare_signal()* methods instead.

부호 있는 0은 언더플로 하는 계산의 결과일 수 있습니다. 계산을 더 정밀하게 수행한다면 얻게 될 결과의 기호를 유지합니다. 크기가 0이기 때문에, 양과 음의 0은 같다고 취급되며 부호는 정보 용입니다.

In addition to the two signed zeros which are distinct yet equal, there are various representations of zero with differing precisions yet equivalent in value. This takes a bit of getting used to. For an eye accustomed to normalized floating-point representations, it is not immediately obvious that the following calculation returns a value equal to zero:

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

9.4.8 스레드로 작업하기

getcontext() 함수는 스레드마다 다른 *Context* 객체에 접근합니다. 별도의 스레드 컨텍스트를 갖는다는 것은 스레드가 다른 스레드를 방해하지 않고 변경할 수 있음을 의미합니다(가령 `getcontext().prec=10`).

마찬가지로, *setcontext()* 함수는 자동으로 대상을 현재 스레드에 할당합니다.

If *setcontext()* has not been called before *getcontext()*, then *getcontext()* will automatically create a new context for use in the current thread. New context objects have default values set from the *decimal.DefaultContext* object.

The *sys.flags.thread_inherit_context* flag affects the context for new threads. If the flag is false, new threads will start with an empty context. In this case, *getcontext()* will create a new context object when called and use the default values from *DefaultContext*. If the flag is true, new threads will start with a copy of context from the caller of *threading.Thread.start()*.

To control the defaults so that each thread will use the same values throughout the application, directly modify the *DefaultContext* object. This should be done *before* any threads are started so that there won' t be a race condition between threads calling *getcontext()*. For example:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
 . . .
```

9.4.9 조리법

다음은 유틸리티 함수로 사용되고 *Decimal* 클래스로 작업하는 방법을 보여주는 몇 가지 조리법입니다:

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

    places:  required number of places after the decimal point
    curr:    optional currency symbol before the sign (may be blank)
    sep:     optional grouping separator (comma, period, space, or blank)
    dp:      decimal point indicator (comma or period)
             only specify as blank when places is zero
    pos:     optional sign for positive numbers: '+', space or blank
    neg:     optional sign for negative numbers: '-', '(', space or blank
    trailneg:optional trailing minus indicator:  '-', ')', space or blank

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    '<0.02>'

    """
    q = Decimal(10) ** -places      # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = list(map(str, digits))
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    if places:
        build(dp)
    if not digits:
        build('0')
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
    build(curr)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print(pi())
    3.141592653589793238462643383

    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x.  Result type matches input type.

    >>> print(exp(Decimal(1)))
    2.718281828459045235360287471
    >>> print(exp(Decimal(2)))
    7.389056098930650227230427461
    >>> print(exp(2.0))
    7.38905609893
    >>> print(exp(2+0j))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    (7.38905609893+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(cos(Decimal('0.5')))
    0.8775825618903727161162815826
    >>> print(cos(0.5))
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(sin(Decimal('0.5')))
    0.4794255386042030002732879352
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

9.4.10 Decimal FAQ

Q. `decimal.Decimal('1234.5')` 라고 입력하는 것은 귀찮은 일입니다. 대화형 인터프리터를 사용할 때 타자를 최소화할 방법이 있습니까?

A. 일부 사용자는 생성자를 하나의 문자로 축약합니다:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. 소수점 두 자리의 고정 소수점 응용 프로그램에서, 일부 입력에 여러 자리가 있고 자리 올림 해야 합니다. 어떤 것은 여분의 자릿수가 없다고 가정되지만, 유효성 검사가 필요합니다. 어떤 방법을 사용해야 합니까?

A. The *quantize()* method rounds to a fixed number of decimal places. If the *Inexact* trap is set, it is also useful for validation:

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
   ...
Inexact: None
```

Q. 일단 유효한 두 자리 입력이 있으면, 응용 프로그램 전체에서 해당 불변성을 어떻게 유지합니까?

A. Some operations like addition, subtraction, and multiplication by an integer will automatically preserve fixed point. Others operations, like division and non-integer multiplication, will change the number of decimal places and need to be followed-up with a *quantize()* step:

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)     # Must quantize non-integer multiplication
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)      # And quantize division
Decimal('0.03')
```

In developing fixed-point applications, it is convenient to define functions to handle the `quantize()` step:

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
...
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                        # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. There are many ways to express the same value. The numbers `200`, `200.000`, `2E2`, and `.02E+4` all have the same value at various precisions. Is there a way to transform them to a single recognizable canonical value?

A. The `normalize()` method maps all equivalent values to a single representative:

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. When does rounding occur in a computation?

A. It occurs *after* the computation. The philosophy of the decimal specification is that numbers are considered exact and are created independent of the current context. They can even have greater precision than current context. Computations process with those exact inputs and then rounding (or other context operations) is applied to the *result* of the computation:

```
>>> getcontext().prec = 5
>>> pi = Decimal('3.1415926535')    # More than 5 digits
>>> pi                              # All digits are retained
Decimal('3.1415926535')
>>> pi + 0                          # Rounded after an addition
Decimal('3.1416')
>>> pi - Decimal('0.00005')         # Subtract unrounded numbers, then round
Decimal('3.1415')
>>> pi + 0 - Decimal('0.00005').    # Intermediate values are rounded
Decimal('3.1416')
```

Q. 일부 십진수 값은 항상 지수 표기법으로 인쇄됩니다. 지수가 아닌 표현을 얻을 방법이 있습니까?

A. For some values, exponential notation is the only way to express the number of significant places in the coefficient. For example, expressing `5.0E+3` as `5000` keeps the value constant but cannot show the original' s two-place significance.

응용 프로그램이 유효 숫자를 추적하는 데 신경 쓰지 않으면, 지수 및 후행 0을 제거하고 유효숫자를 잃지만, 값이 바뀌지 않도록 하기는 쉽습니다:

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

```
>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

Q. 일반 float를 `Decimal`로 변환하는 방법이 있습니까?

A. Yes, any binary floating-point number can be exactly expressed as a Decimal though an exact conversion may take more precision than intuition would suggest:

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. 복잡한 계산에서, 정밀도가 부족하거나 자리 올림 이상이 발생하여 엉터리 결과를 얻지는 않았는지 확인하려면 어떻게 해야 합니까?

A. decimal 모듈은 결과를 쉽게 테스트할 수 있게 합니다. 가장 좋은 방법은 더 높은 정밀도와 다양한 자리 올림 모드를 사용하여 계산을 다시 실행하는 것입니다. 크게 다른 결과는 정밀도 부족, 자리 올림 모드 문제, 부적절한 입력 또는 수치가 불안정한 알고리즘을 나타냅니다.

컨텍스트 정밀도가 입력이 아닌 연산 결과에 적용된다는 사실을 확인했습니다. 다른 정밀도의 값을 혼합할 때 주의해야 할 것이 있습니까?

A. 그렇습니다. 원칙은 모든 값이 정확한 것으로 간주하므로 해당 값에 대한 산술도 마찬가지라는 것입니다. 결과 만 자리 올림 됩니다. 입력에 대한 이점은 "입력하는 것이 얻는 것" 이라는 것입니다. 단점은 입력값을 자리 올림 하는 것을 잊어버리면 결과가 이상하게 보일 수 있다는 점입니다:

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

해법은 정밀도를 높이거나 단항 플러스 연산을 사용하여 입력의 자리 올림을 강제 수행하는 것입니다:

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # unary plus triggers rounding
Decimal('1.23')
```

다른 방법으로, 입력은 `Context.create_decimal()` 메서드를 사용하여 생성 시에 자리 올림 될 수 있습니다:

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

Q. CPython 구현은 커다란 수에서 빠릅니까?

A. Yes. In the CPython and PyPy3 implementations, the C/CFFI versions of the decimal module integrate the high speed libmpdec library for arbitrary precision correctly rounded decimal floating-point arithmetic[1]. `libmpdec` uses Karatsuba multiplication for medium-sized numbers and the Number Theoretic Transform for very large numbers.

The context must be adapted for exact arbitrary precision arithmetic. `Emin` and `Emax` should always be set to the maximum values, `clamp` should always be 0 (the default). Setting `prec` requires some care.

The easiest approach for trying out bignum arithmetic is to use the maximum value for `prec` as well[2]:

```
>>> setcontext(Context(prec=MAX_PREC, Emax=MAX_EMAX, Emin=MIN_EMIN))
>>> x = Decimal(2) ** 256
>>> x / 128
Decimal(
↪'904625697166532776746648320380374280103671755200316906558262375061821325312')
```

For inexact results, `MAX_PREC` is far too large on 64-bit platforms and the available memory will be insufficient:

[1] Added in version 3.3.

[2] 버전 3.9에서 변경: 이 접근법은 정수가 아닌 거듭제곱을 제외한 모든 정확한 결과에 적용됩니다.

```
>>> Decimal(1) / 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
```

On systems with overallocation (e.g. Linux), a more sophisticated approach is to adjust *prec* to the amount of available RAM. Suppose that you have 8GB of RAM and expect 10 simultaneous operands using a maximum of 500MB each:

```
>>> import sys
>>>
>>> # Maximum number of digits for a single operand using 500MB in 8-byte words
>>> # with 19 digits per word (4-byte and 9 digits for the 32-bit build):
>>> maxdigits = 19 * ((500 * 1024**2) // 8)
>>>
>>> # Check that this works:
>>> c = Context(prec=maxdigits, Emax=MAX_EMAX, Emin=MIN_EMIN)
>>> c.traps[Inexact] = True
>>> setcontext(c)
>>>
>>> # Fill the available precision with nines:
>>> x = Decimal(0).logical_invert() * 9
>>> sys.getsizeof(x)
524288112
>>> x + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  decimal.Inexact: [<class 'decimal.Inexact'>]
```

일반적으로 (그리고 특히 초과 할당이 없는 시스템에서), 더 엄격한 경계를 추정하고 모든 계산이 정확할 것으로 예상되면 *Inexact* 트랩을 설정하는 것이 좋습니다.

9.5 fractions — 유리수

소스 코드: Lib/fractions.py

---

fractions 모듈은 유리수 산술을 지원합니다.

Fraction 인스턴스는 한 쌍의 정수, 다른 유리수 또는 문자열로 만들 수 있습니다.

class fractions.**Fraction**(*numerator=0, denominator=1*)

class fractions.**Fraction**(*number*)

class fractions.**Fraction**(*string*)

The first version requires that *numerator* and *denominator* are instances of *numbers.Rational* and returns a new *Fraction* instance with value `numerator/denominator`. If *denominator* is `0`, it raises a *ZeroDivisionError*.

The second version requires that *number* is an instance of *numbers.Rational* or has the `as_integer_ratio()` method (this includes *float* and *decimal.Decimal*). It returns a *Fraction* instance with exactly the same value. Assumed, that the `as_integer_ratio()` method returns a pair of coprime integers and last one is positive. Note that due to the usual issues with binary point (see tut-fp-issues), the argument to `Fraction(1.1)` is not exactly equal to 11/10, and so `Fraction(1.1)` does *not* return `Fraction(11, 10)` as one might expect. (But see the documentation for the *limit_denominator()* method below.)

The last version of the constructor expects a string. The usual form for this instance is:

```
[sign] numerator ['/' denominator]
```

이때, 선택적 sign은 '+' 나 '-'일 수 있으며 numerator 와 denominator(있다면)는 십진수 문자열입니다 (밑줄은 코드에서 정수 리터럴과 마찬가지로 숫자를 구분하는 데 사용할 수 있습니다). 또한, 유한한 값을 나타내고 *float* 생성자에서 허용하는 모든 문자열은 *Fraction* 생성자에서도 허용됩니다. 모든 형식에서, 입력 문자열에는 선행과/이나 후행 공백이 있을 수도 있습니다. 여기 예제가 있습니다:

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

Fraction 클래스는 추상 베이스 클래스 *numbers.Rational*를 상속하며, 그 클래스의 모든 메서드와 연산을 구현합니다. *Fraction* 인스턴스는 해시 가능하고, 불변으로 취급해야 합니다. 또한, *Fraction*에는 다음과 같은 프로퍼티와 메서드가 있습니다:

버전 3.2에서 변경: *Fraction* 생성자는 이제 *float*와 *decimal.Decimal* 인스턴스를 받아들입니다.

버전 3.9에서 변경: *math.gcd()* 함수가 이제 *numerator*와 *denominator*를 정규화하는 데 사용됩니다. *math.gcd()*는 항상 *int* 형을 반환합니다. 이전에는, GCD 형이 *numerator*와 *denominator*에 의존했습니다.

버전 3.11에서 변경: Underscores are now permitted when creating a *Fraction* instance from a string, following **PEP 515** rules.

버전 3.11에서 변경: *Fraction* implements `__int__` now to satisfy `typing.SupportsInt` instance checks.

버전 3.12에서 변경: Space is allowed around the slash for string inputs: `Fraction('2 / 3')`.

버전 3.12에서 변경: *Fraction* instances now support float-style formatting, with presentation types `"e"`, `"E"`, `"f"`, `"F"`, `"g"`, `"G"` and `"%"`.

버전 3.13에서 변경: Formatting of *Fraction* instances without a presentation type now supports fill, alignment, sign handling, minimum width and grouping.

버전 3.14에서 변경: The *Fraction* constructor now accepts any objects with the `as_integer_ratio()` method.

numerator

기약 분수로 나타낼 때 Fraction의 분자.

denominator

기약 분수로 나타낼 때 Fraction의 분모.

as_integer_ratio()

비율이 원래 Fraction과 같습니다. 비율은 양의 분모를 갖는 가장 작은 항으로 표현됩니다.

Added in version 3.8.

is_integer()

Return `True` if the Fraction is an integer.

Added in version 3.12.

classmethod from_float(*f*)

float 나 *numbers.Integral*의 인스턴스만 받아들이는 대체 생성자. `Fraction.from_float(0.3)`가 `Fraction(3, 10)`와 같은 값이 아니라는 점에 유의하십시오.

> **참고**
>
> 파이썬 3.2 이상에서는, *float*에서 직접 *Fraction* 인스턴스를 생성할 수도 있습니다.

classmethod from_decimal(*dec*)

decimal.Decimal 나 *numbers.Integral*의 인스턴스만 받아들이는 대체 생성자.

> **참고**
>
> 파이썬 3.2 이상에서는, *decimal.Decimal* 인스턴스에서 직접 *Fraction* 인스턴스를 생성할 수도 있습니다.

classmethod from_number(*number*)

Alternative constructor which only accepts instances of *numbers.Integral*, *numbers.Rational*, *float* or *decimal.Decimal*, and objects with the `as_integer_ratio()` method, but not strings.

Added in version 3.14.

limit_denominator(*max_denominator=1000000*)

분모가 최대 max_denominator인 `self`에 가장 가까운 *Fraction*을 찾아서 반환합니다. 이 메서드는 주어진 부동 소수점 수에 대한 유리한 근사를 찾는 데 유용합니다:

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

또는 float로 표현된 유리수를 복구할 때 유용합니다:

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

__floor__()

가장 큰 *int* `<= self`를 반환합니다. 이 메서드는 *math.floor()* 함수를 통해 액세스할 수도 있습니다:

```
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__()

가장 작은 *int* >= self를 반환합니다. 이 메서드는 *math.ceil()* 함수를 통해 액세스할 수도 있습니다.

__round__()

__round__(*ndigits*)

첫 번째 버전은 self에 가장 가까운 *int*를 반환하는데, 절반은 짝수로 자리 올림 합니다. 두 번째 버전은 self를 가장 가까운 Fraction(1, 10**ndigits)의 배수로 자리 올림 하는데 (ndigits가 음수면 논리적으로), 역시 짝수로 자리 올림 합니다. 이 메서드는 *round()* 함수를 통해 액세스할 수도 있습니다.

__format__(*format_spec*, /)

Provides support for formatting of *Fraction* instances via the *str.format()* method, the *format()* built-in function, or Formatted string literals.

If the format_spec format specification string does not end with one of the presentation types 'e', 'E', 'f', 'F', 'g', 'G' or '%' then formatting follows the general rules for fill, alignment, sign handling, minimum width, and grouping as described in the *format specification mini-language*. The "alternate form" flag '#' is supported: if present, it forces the output string to always include an explicit denominator, even when the value being formatted is an exact integer. The zero-fill flag '0' is not supported.

If the format_spec format specification string ends with one of the presentation types 'e', 'E', 'f', 'F', 'g', 'G' or '%' then formatting follows the rules outlined for the *float* type in the 포맷 명세 미니 언어 section.

Here are some examples:

```
>>> from fractions import Fraction
>>> format(Fraction(103993, 33102), '_')
'103_993/33_102'
>>> format(Fraction(1, 7), '.^+10')
'...+1/7...'
>>> format(Fraction(3, 1), '')
'3'
>>> format(Fraction(3, 1), '#')
'3/1'
>>> format(Fraction(1, 7), '.40g')
'0.1428571428571428571428571428571428571429'
>>> format(Fraction('1234567.855'), '_.2f')
'1_234_567.86'
>>> f"{Fraction(355, 113):*>20.6e}"
'********3.141593e+00'
>>> old_price, new_price = 499, 672
>>> "{:.2%} price increase".format(Fraction(new_price, old_price) - 1)
'34.67% price increase'
```

더 보기

모듈 ***numbers***

숫자 계층을 구성하는 추상 베이스 클래스.

9.6 random — 의사 난수 생성

소스 코드: Lib/random.py

이 모듈은 다양한 분포에 대한 의사 난수 생성기를 구현합니다.

정수에 대해서는, 범위에서 균일한 선택이 있습니다. 시퀀스에 대해서는, 무작위 요소의 균일한 선택, 리스트를 제자리(in-place)에서 임의 순열을 생성하는 함수 및 중복 없는(without replacement) 무작위 표본 추출(sampling)을 위한 함수가 있습니다.

실수에 대해서는, 균일(uniform), 정규(normal) (가우시안(Gaussian)), 로그 정규(lognormal), 음의 지수(negative exponential), 감마(gamma) 및 베타(beta) 분포를 계산하는 함수가 있습니다. 각도 분포를 생성하기 위해, 폰 미제스(von Mises) 분포를 사용할 수 있습니다.

거의 모든 모듈 함수는 기본 함수 `random()`에 의존하는데, 이 함수는 반 열린 구간 `0.0 <= X < 1.0`에서 무작위 float를 균일하게 생성합니다. 파이썬은 메르센 트위스터(Mersenne Twister)를 핵심 생성기로 사용합니다. 53비트 정밀도의 float를 생성하며, 주기는 2**19937-1 입니다. C로 작성된 하부 구현은 빠르고 스레드 안전합니다. 메르센 트위스터는 가장 광범위하게 테스트 된 난수 생성기 중 하나입니다. 그러나, 완전히 결정적이므로, 모든 목적에 적합하지는 않으며, 암호화 목적에는 전혀 적합하지 않습니다.

이 모듈에서 제공하는 함수는 실제로는 `random.Random` 클래스의 숨겨진 인스턴스에 대해 연결된 메서드입니다. `Random` 인스턴스를 직접 인스턴스 화하여 상태를 공유하지 않는 생성기를 얻을 수 있습니다.

Class `Random` can also be subclassed if you want to use a different basic generator of your own devising: see the documentation on that class for more details.

`random` 모듈은 운영 체제에서 제공하는 소스에서 난수를 생성하는 시스템 함수 `os.urandom()`을 사용하는 `SystemRandom` 클래스도 제공합니다.

> **경고**
>
> 이 모듈의 의사 난수 생성기를 보안 목적으로 사용해서는 안 됩니다. 보안이나 암호화 용도를 위해서는, `secrets` 모듈을 참조하십시오.

> **더 보기**
>
> M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3–30 1998.
>
> 긴 주기와 비교적 간단한 업데이트 연산을 하는 호환 가능한 대체 난수 생성기를 위한 Complementary-Multiply-with-Carry recipe

> **참고**
>
> The global random number generator and instances of `Random` are thread-safe. However, in the free-threaded build, concurrent calls to the global generator or to the same instance of `Random` may encounter contention and poor performance. Consider using separate instances of `Random` per thread instead.

9.6.1 관리 함수

`random.seed(a=None, version=2)`

난수 생성기를 초기화합니다.

*a*가 생략되거나 `None`이면, 현재 시스템 시간이 사용됩니다. 운영 체제에서 임의성 소스(randomness sources)를 제공하면, 시스템 시간 대신 사용됩니다 (가용성에 대한 자세한 내용은 `os.urandom()` 함수를 참조하십시오).

*a*가 int이면, 직접 사용됩니다.

버전(version) 2(기본값)에서는, `str`, `bytes` 또는 `bytearray` 객체가 `int`로 변환되어 모든 비트가 사용됩니다.

버전(version) 1(이전 버전의 파이썬에서 온 임의의 시퀀스를 재현하기 위해 제공됩니다)에서는, `str`과 `bytes`를 위한 알고리즘은 더 좁은 범위의 시드(seed)를 생성합니다.

버전 3.2에서 변경: 문자열 시드의 모든 비트를 사용하는 버전 2 체계로 이동했습니다.

버전 3.11에서 변경: *seed*는 다음 형 중 하나여야 합니다: `None`, `int`, `float`, `str`, `bytes` 또는 `bytearray`.

`random.getstate()`

생성기의 현재 내부 상태를 포착하는 객체를 반환합니다. 이 객체는 `setstate()`로 전달되어 상태를 복원 할 수 있습니다.

`random.setstate(state)`

*state*는 `getstate()`에 대한 이전 호출에서 얻은 것이어야 하고, `setstate()`는 생성기의 내부 상태를 `getstate()`가 호출될 당시의 상태로 복원합니다.

9.6.2 바이트열 함수

`random.randbytes(n)`

n 무작위 바이트를 생성합니다.

이 메서드를 사용하여 보안 토큰을 생성해서는 안 됩니다. 대신 `secrets.token_bytes()`를 사용하십시오.

Added in version 3.9.

9.6.3 정수 함수

`random.randrange(stop)`

`random.randrange(start, stop[, step])`

Return a randomly selected element from `range(start, stop, step)`.

This is roughly equivalent to `choice(range(start, stop, step))` but supports arbitrarily large ranges and is optimized for common cases.

The positional argument pattern matches the `range()` function.

Keyword arguments should not be used because they can be interpreted in unexpected ways. For example `randrange(start=100)` is interpreted as `randrange(0, 100, 1)`.

버전 3.2에서 변경: `randrange()`는 균일하게 분포된 값을 생성하는 데 있어 더욱 정교합니다. 이전에는 약간 고르지 않은 분포를 생성 할 수 있는 `int(random()*n)`와 같은 스타일을 사용했습니다.

버전 3.12에서 변경: Automatic conversion of non-integer types is no longer supported. Calls such as `randrange(10.0)` and `randrange(Fraction(10, 1))` now raise a `TypeError`.

`random.randint(a, b)`

`a <= N <= b`를 만족하는 임의의 정수 *N*을 반환합니다. `randrange(a, b+1)`의 별칭.

`random.getrandbits(k)`

k 임의의 비트를 갖는 음이 아닌 파이썬 정수를 반환합니다. 이 메서드는 메르센 트위스터(Mersenne Twister) 생성기와 함께 제공되며, 일부 다른 생성기도 API의 선택적 부분으로 제공할 수 있습니다. 사용 가능할 때, `getrandbits()`는 `randrange()`가 임의로 넓은 범위를 처리 할 수 있도록 합니다.

버전 3.9에서 변경: 이 메서드는 이제 *k*에 0을 허용합니다.

9.6.4 시퀀스 함수

random.**choice**(*seq*)

비어 있지 않은 시퀀스 *seq*에서 임의의 요소를 반환합니다. *seq*가 비어 있으면, `IndexError`를 발생시킵니다.

random.**choices**(*population*, *weights=None*, **, *cum_weights=None*, *k=1*)

*population*에서 중복을 허락하면서(with replacement) 선택한 *k* 크기의 요소 리스트를 반환합니다. *population*이 비어 있으면, `IndexError`를 발생시킵니다.

weights 시퀀스가 지정되면, 상대 가중치에 따라 선택됩니다. 대안적으로, *cum_weights* 시퀀스가 제공되면, (아마도 `itertools.accumulate()`를 사용하여 계산된) 누적 가중치(cumulative weights)에 따라 선택이 이루어집니다. 예를 들어, 상대 가중치 `[10, 5, 30, 5]`는 누적 가중치 `[10, 15, 45, 50]`과 동등합니다. 내부적으로, 상대 가중치는 선택하기 전에 누적 가중치로 변환되므로, 누적 가중치를 제공하면 작업이 줄어듭니다.

*weights*나 *cum_weights*를 지정하지 않으면, 같은 확률로 선택됩니다. weights 시퀀스가 제공되면, *population* 시퀀스와 길이가 같아야 합니다. *weights*와 *cum_weights*를 모두 지정하는 것은 `TypeError`입니다.

*weights*나 *cum_weights*는 `random()`이 반환하는 `float` 값과 상호 운용되는 모든 숫자 형을 사용할 수 있습니다 (정수, 부동 소수점(float) 및 유리수(fractions)는 포함하지만, 십진수(decimal)는 제외합니다). 가중치는 음수가 아니고 유한하다고 가정합니다. 모든 가중치가 0이면 `ValueError`가 발생합니다.

주어진 시드에 대해, 균등한 가중치를 갖는 `choices()` 함수는 일반적으로 `choice()`에 대한 반복 호출과는 다른 시퀀스를 생성합니다. `choices()`에서 사용하는 알고리즘은 내부 일관성과 속도를 위해 부동 소수점 산술을 사용합니다. `choice()`에서 사용하는 알고리즘은 자리 올림 오차로 인한 작은 바이어스(bias)를 피하려고 반복 선택을 통한 정수 산술로 기본 설정됩니다.

Added in version 3.6.

버전 3.9에서 변경: 모든 가중치가 0이면 `ValueError`를 발생시킵니다.

random.**shuffle**(*x*)

시퀀스 *x*를 제자리에서 섞습니다.

불변 시퀀스를 섞고 새로운 섞인 리스트를 반환하려면, 대신 `sample(x, k=len(x))`를 사용하십시오.

작은 `len(x)`의 경우에조차, *x*의 총 순열 수는 대부분의 난수 생성기 주기보다 빠르게 커질 수 있습니다. 이것은 긴 시퀀스의 대부분 순열이 절대로 생성될 수 없음을 의미합니다. 예를 들어, 길이가 2080인 시퀀스가 메르센 트위스터 난수 생성기의 주기 안에 들어갈 수 있는 최대입니다.

버전 3.11에서 변경: 선택적 매개 변수 *random*을 제거했습니다.

random.**sample**(*population*, *k*, **, *counts=None*)

population 시퀀스에서 선택한 고유한 요소의 *k* 길이 리스트를 반환합니다. 중복 없는(without replacement) 무작위 표본 추출(sampling)에 사용됩니다.

원래 population을 변경하지 않고, population의 요소를 포함하는 새 리스트를 반환합니다. 결과 리스트는 선택 순서를 따라서, 모든 서브 슬라이스도 유효한 임의의 표본이 됩니다. 이것은 추첨 당첨자(표본)를 대상(grand prize)과 차점자들(서브 슬라이스)로 나눌 수 있도록 합니다.

population의 멤버는 해시 가능하거나 고유할 필요가 없습니다. population이 반복을 포함하면, 각 등장(occurrence)은 표본에서 가능한 선택입니다.

반복되는 요소는 한 번에 하나씩 또는 선택적 키워드 전용 *counts* 매개 변수로 지정할 수 있습니다. 예를 들어 `sample(['red', 'blue'], counts=[4, 2], k=5)`는 `sample(['red', 'red', 'red', 'red', 'blue', 'blue'], k=5)`와 동등합니다.

정수 범위에서 표본을 선택하려면, `range()` 객체를 인자로 사용하십시오. 이는 큰 population에서 표본 추출할 때 특히 빠르고 공간 효율적입니다: `sample(range(10000000), k=60)`.

표본 크기가 population 크기보다 크면 `ValueError`가 발생합니다.

버전 3.9에서 변경: *counts* 매개 변수를 추가했습니다.

버전 3.11에서 변경: The *population* must be a sequence. Automatic conversion of sets to lists is no longer supported.

9.6.5 이산 분포

The following function generates a discrete distribution.

`random.binomialvariate`(*n=1*, *p=0.5*)

Binomial distribution. Return the number of successes for *n* independent trials with the probability of success in each trial being *p*:

Mathematically equivalent to:

```
sum(random() < p for i in range(n))
```

The number of trials *n* should be a non-negative integer. The probability of success *p* should be between `0.0 <= p <= 1.0`. The result is an integer in the range `0 <= X <= n`.

Added in version 3.12.

9.6.6 실수 분포

다음 함수는 특정 실수 분포를 생성합니다. 함수 매개 변수는 일반적인 수학적 관행에 사용되는 분포 방정식에서 해당 변수의 이름을 따서 명명됩니다; 이러한 방정식의 대부분은 모든 통계 교과서에서 찾을 수 있습니다.

`random.random`()

`0.0 <= X < 1.0` 구간에서 다음 임의의 부동 소수점 숫자를 반환합니다

`random.uniform`(*a*, *b*)

`a <= b` 일 때 `a <= N <= b`, `b < a` 일 때 `b <= N <= a`를 만족하는 임의의 부동 소수점 숫자 *N*을 반환합니다.

종단 값 `b`는 표현식 `a + (b-a) * random()`의 부동 소수점 자리 올림에 따라 범위에 포함되거나 포함되지 않을 수 있습니다.

`random.triangular`(*low*, *high*, *mode*)

`low <= N <= high`를 만족하고 이 경계 사이에 지정된 모드(*mode*)를 갖는 임의의 부동 소수점 숫자 *N*을 반환합니다. *low* 및 *high* 경계는 기본적으로 0과 1입니다. *mode* 인자는 기본적으로 경계 사이의 중간 점으로, 대칭 분포를 제공합니다.

`random.betavariate`(*alpha*, *beta*)

베타 분포. 매개 변수의 조건은 `alpha > 0`과 `beta > 0`입니다. 반환된 값의 범위는 0에서 1입니다.

`random.expovariate`(*lambd=1.0*)

지수 분포. *lambd*는 1.0을 원하는 평균으로 나눈 값입니다. 0이 아니어야 합니다. (매개 변수는 "lambda"라고 부르지만, 파이썬에서는 예약어입니다.) 반환된 값의 범위는, *lambd*가 양수이면 0에서 양의 무한대이고, *lambd*가 음수이면 음의 무한대에서 0입니다.

버전 3.12에서 변경: Added the default value for `lambd`.

`random.gammavariate`(*alpha*, *beta*)

Gamma distribution. (*Not* the gamma function!) The shape and scale parameters, *alpha* and *beta*, must have positive values. (Calling conventions vary and some sources define 'beta' as the inverse of the scale).

확률 분포 함수는 다음과 같습니다:

```
x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

random.**gauss**(*mu=0.0*, *sigma=1.0*)

정규 분포, 가우시안 분포라고도 합니다. *mu*는 평균이고, *sigma*는 표준 편차입니다. 이것은 아래에 정의된 `normalvariate()` 함수보다 약간 빠릅니다.

다중 스레딩 참고: 두 스레드가 이 함수를 동시에 호출하면, 같은 반환 값을 받을 수 있습니다. 이것은 세 가지 방법으로 피할 수 있습니다. 1) 각 스레드가 난수 생성기의 다른 인스턴스를 사용하도록 합니다. 2) 모든 호출에 록을 둡니다. 3) 더 느리지만, 스레드 안전한 `normalvariate()` 함수를 대신 사용합니다.

버전 3.11에서 변경: *mu* and *sigma* now have default arguments.

random.**lognormvariate**(*mu*, *sigma*)

로그 정규 분포. 이 분포의 자연로그를 취하면, 평균 *mu*와 표준 편차 *sigma*를 갖는 정규 분포를 얻게 됩니다. *mu*는 아무 값이나 될 수 있으며, *sigma*는 0보다 커야 합니다.

random.**normalvariate**(*mu=0.0*, *sigma=1.0*)

정규 분포. *mu*는 평균이고, *sigma*는 표준 편차입니다.

버전 3.11에서 변경: *mu* and *sigma* now have default arguments.

random.**vonmisesvariate**(*mu*, *kappa*)

*mu*는 0과 2**pi* 사이의 라디안으로 표현된 평균 각도이며, *kappa*는 0 이상이어야 하는 집중도(concentration) 매개 변수입니다. *kappa*가 0이면, 이 분포는 0에서 2**pi*에 걸친 균등한 임의의 각도로 환원됩니다.

random.**paretovariate**(*alpha*)

파레토 분포. *alpha*는 모양(shape) 매개 변수입니다.

random.**weibullvariate**(*alpha*, *beta*)

베이불 분포. *alpha*는 크기(scale) 매개 변수이고 *beta*는 모양(shape) 매개 변수입니다.

9.6.7 대체 생성기

class random.**Random**([*seed*])

`random` 모듈에서 사용하는 기본 의사 난수 생성기를 구현하는 클래스.

버전 3.11에서 변경: 이전에는 *seed*가 해시 가능한 어떤 객체라도 될 수 있었습니다. 이제는 다음 형으로 제한됩니다: `None`, `int`, `float`, `str`, `bytes` 또는 `bytearray`.

Subclasses of `Random` should override the following methods if they wish to make use of a different basic generator:

seed(*a=None*, *version=2*)

Override this method in subclasses to customise the `seed()` behaviour of `Random` instances.

getstate()

Override this method in subclasses to customise the `getstate()` behaviour of `Random` instances.

setstate(*state*)

Override this method in subclasses to customise the `setstate()` behaviour of `Random` instances.

random()

Override this method in subclasses to customise the `random()` behaviour of `Random` instances.

Optionally, a custom generator subclass can also supply the following method:

getrandbits(*k*)

Override this method in subclasses to customise the `getrandbits()` behaviour of `Random` instances.

randbytes(*n*)

Override this method in subclasses to customise the `randbytes()` behaviour of `Random` instances.

class random.**SystemRandom**([*seed*])

운영 체제에서 제공하는 소스에서 난수를 생성하기 위해 `os.urandom()` 함수를 사용하는 클래스. 모든 시스템에서 사용 가능한 것은 아닙니다. 소프트웨어 상태에 의존하지 않으며, 시퀀스는 재현되지 않습니다. 따라서, `seed()` 메서드는 효과가 없으며, 무시됩니다. `getstate()`와 `setstate()` 메서드는 호출되면 `NotImplementedError`를 발생시킵니다.

9.6.8 재현성에 대한 참고 사항

때때로 의사 난수 생성기가 만든 시퀀스를 재현하는 것이 유용 할 수 있습니다. 시드 값을 재사용하면, 여러 스레드가 실행되고 있지 않은 한 실행할 때마다 같은 시퀀스를 재현할 수 있어야 합니다.

random 모듈의 알고리즘과 시딩(seeding) 함수의 대부분은 파이썬 버전에 따라 변경될 수 있지만, 두 가지 측면은 변경되지 않음이 보장됩니다:

- 새로운 시딩 메서드가 추가되면, 이전 버전과 호환되는 시더(seeder)가 제공될 것입니다.
- 호환 시더에 같은 시드가 제공되면 생성기의 `random()` 메서드는 같은 시퀀스를 계속 생성할 것입니다.

9.6.9 예제

기본 예제:

```
>>> random()                          # 무작위 float:  0.0 <= x < 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                # 무작위 float:  2.5 <= x <= 10.0
3.1800146073117523

>>> expovariate(1 / 5)                # 평균 5초의 도착 간격
5.148957571865031

>>> randrange(10)                     # 경계를 포함하는 0에서 9 사이의 정수
7

>>> randrange(0, 101, 2)              # 경계를 포함하는 0에서 100 사이의 짝수 정수
26

>>> choice(['win', 'lose', 'draw'])   # 시퀀스로부터의 단일 무작위 요소
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                     # 리스트를 섞습니다
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4) # 중복 없는 4개의 표본
[40, 10, 50, 30]
```

시뮬레이션:

```
>>> # 여섯 개의 룰렛 휠이 회전합니다 (중복을 허락하는 가중치 있는 샘플링)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # 52장의 카드 덱에서 중복 없이 20장의 카드를 나누고,
>>> # 10 값(10, 잭, 퀸 또는 킹)을
>>> # 가진 카드의 비율을 결정합니다.
>>> deal = sample(['tens', 'low cards'], counts=[16, 36], k=20)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> deal.count('tens') / 20
0.15

>>> # 60%의 확률로 앞(head)이 나오는 편향된(biased) 동전을 7번 던졌을 때 5번 이상
>>> # 앞이 나올 확률을 추정합니다.
>>> sum(binomialvariate(n=7, p=0.6) >= 5 for i in range(10_000)) / 10_000
0.4169

>>> # 다섯 표본의 중간값이 가운데 2개의 사분위(quartile)에 있을 확률
>>> def trial():
...     return 2_500 <= sorted(choices(range(10_000), k=5))[2] < 7_500
...
>>> sum(trial() for i in range(10_000)) / 10_000
0.7958
```

표본의 평균에 대한 신뢰 구간을 추정하기 위해 중복을 허용하는(with replacement) 재표본추출(resampling)을 사용하는 통계적 부트스트래핑(statistical bootstrapping)의 예:

```
# https://www.thoughtco.com/example-of-bootstrapping-3126155
from statistics import fmean as mean
from random import choices

data = [41, 50, 29, 37, 81, 30, 73, 63, 20, 35, 68, 22, 60, 31, 95]
means = sorted(mean(choices(data, k=len(data))) for i in range(100))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[5]:.1f} to {means[94]:.1f}')
```

약물과 위약의 효과 간에 관찰된 차이의 통계적 유의성 또는 p-값을 결정하기 위한 재표본추출 순열 검증(resampling permutation test)의 예:

```
# Dennis Shasha 와 Manda Wilson의 "Statistics is Easy" 에서 따온 예제
from statistics import fmean as mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10_000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} instances with a difference')
print(f'at least as extreme as the observed difference of {observed_diff:.1f}.')
print(f'The one-sided p-value of {count / n:.4f} leads us to reject the null')
print(f'hypothesis that there is no difference between the drug and the placebo.')
```

다중 서버 큐를 위한 도착 시간과 서비스 제공의 시뮬레이션:

```
from heapq import heapify, heapreplace
from random import expovariate, gauss
from statistics import mean, quantiles
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
average_arrival_interval = 5.6
average_service_time = 15.0
stdev_service_time = 3.5
num_servers = 3

waits = []
arrival_time = 0.0
servers = [0.0] * num_servers  # 각 서버를 사용할 수 있는 시간
heapify(servers)
for i in range(1_000_000):
    arrival_time += expovariate(1.0 / average_arrival_interval)
    next_server_available = servers[0]
    wait = max(0.0, next_server_available - arrival_time)
    waits.append(wait)
    service_duration = max(0.0, gauss(average_service_time, stdev_service_time))
    service_completed = arrival_time + wait + service_duration
    heapreplace(servers, service_completed)

print(f'Mean wait: {mean(waits):.1f}   Max wait: {max(waits):.1f}')
print('Quartiles:', [round(q, 1) for q in quantiles(waits)])
```

더 보기

시뮬레이션(simulation), 표본 추출(sampling), 섞기(shuffling) 및 교차 검증(cross-validation)을 포함하는 몇 가지 기본 개념만을 사용한, 통계 분석에 대한 Jake Vanderplas의 비디오 자습서 Statistics for Hackers

Economics Simulation 이 모듈에서 제공하는 많은 도구와 분포(gauss, uniform, sample, betavariate, choice, triangular 및 randrange)의 효과적인 사용을 보여주는 Peter Norvig의 시장(marketplace) 시뮬레이션.

A Concrete Introduction to Probability (using Python) 확률 이론의 기초, 시뮬레이션 작성 방법 및 파이썬을 사용해서 데이터 분석을 수행하는 방법을 다루는 Peter Norvig의 자습서.

9.6.10 조리법

These recipes show how to efficiently make random selections from the combinatoric iterators in the `itertools` module:

```
def random_product(*args, repeat=1):
    "itertools.product(*args, **kwds) 에서 무작위 선택"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(map(random.choice, pools))

def random_permutation(iterable, r=None):
    "itertools.permutations(iterable, r) 에서 무작위 선택"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "itertools.combinations(iterable, r) 에서 무작위 선택"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    "중복을 허락하면서 r 요소를 선택합니다. 결과를 iterable 과 일치하도록 정렬합니다."
    # 결과는 set(itertools.combinations_with_replacement(iterable, r)) 의 원소입니다.
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.choices(range(n), k=r))
    return tuple(pool[i] for i in indices)
```

기본 `random()` 은 $0.0 \le x < 1.0$ 범위에서 2^{-53}의 배수를 반환합니다. 이러한 모든 숫자는 균등 간격으로 분포되어있고 파이썬 float로 정확하게 표현할 수 있습니다. 그러나, 해당 범위의 다른 많은 표현 가능한 부동 소수점은 가능한 선택이 아닙니다. 예를 들어, `0.05954861408025609` 는 2^{-53}의 정수배가 아닙니다.

다음 조리법은 다른 접근 방식을 사용합니다. 범위의 모든 부동 소수점이 가능한 선택입니다. 가수(mantissa)는 $2^{52} \le mantissa < 2^{53}$ 범위에 있는 정수의 균등 분포(uniform distribution)에서 옵니다. 지수(exponent)는 -53 보다 작은 지수가 다음으로 큰 지수의 절반만큼 자주 발생하는 기하 분포(geometric distribution)에서 옵니다.

```
from random import Random
from math import ldexp

class FullRandom(Random):

    def random(self):
        mantissa = 0x10_0000_0000_0000 | self.getrandbits(52)
        exponent = -53
        x = 0
        while not x:
            x = self.getrandbits(32)
            exponent += x.bit_length() - 32
        return ldexp(mantissa, exponent)
```

클래스의 모든 실숫값 분포는 새 메서드를 사용합니다:

```
>>> fr = FullRandom()
>>> fr.random()
0.05954861408025609
>>> fr.expovariate(0.25)
8.87925541791544
```

조리법은 개념적으로 $0.0 \le x < 1.0$ 범위의 2^{-1074}의 모든 배수에서 선택하는 알고리즘과 동등합니다. 이러한 모든 숫자는 균등한 간격이지만, 대부분은 가장 가까운 표현 가능한 파이썬 부동 소수점으로 자리 내림해야 합니다. (값 2^{-1074}은 가장 작은 양의 정규화되지 않은 부동 소수점이며 `math.ulp(0.0)` 과 같습니다.)

더 보기

Generating Pseudo-random Floating-Point Values Allen B. Downey의 논문은 `random()` 이 일반적으로 생성하는 것보다 더 세밀한 부동 소수점을 생성하는 방법을 설명합니다.

9.6.11 Command-line usage

Added in version 3.13.

The `random` module can be executed from the command line.

```
python -m random [-h] [-c CHOICE [CHOICE ...] | -i N | -f N] [input ...]
```

The following options are accepted:

-h, --help

Show the help message and exit.

-c CHOICE [CHOICE ...]

--choice CHOICE [CHOICE ...]

Print a random choice, using `choice()`.

-i <N>

--integer <N>

Print a random integer between 1 and N inclusive, using `randint()`.

-f <N>

--float <N>

Print a random floating-point number between 0 and N inclusive, using `uniform()`.

If no options are given, the output depends on the input:

- String or multiple: same as `--choice`.
- Integer: same as `--integer`.
- Float: same as `--float`.

9.6.12 Command-line example

Here are some examples of the `random` command-line interface:

```
$ # 무작위로 하나를 선택합니다
$ python -m random egg bacon sausage spam "Lobster Thermidor aux crevettes with a␣
↪Mornay sauce"
Lobster Thermidor aux crevettes with a Mornay sauce

$ # 정수 난수
$ python -m random 6
6

$ # 실수 난수
$ python -m random 1.8
1.7080016272295635

$ # 명시적 인자로
$ python  -m random --choice egg bacon sausage spam "Lobster Thermidor aux␣
↪crevettes with a Mornay sauce"
egg

$ python -m random --integer 6
3

$ python -m random --float 1.8
1.5666339105010318

$ python -m random --integer 6
5

$ python -m random --float 6
3.1942323316565915
```

9.7 statistics — 수학 통계 함수

Added in version 3.4.

소스 코드: Lib/statistics.py

이 모듈은 숫자 (*Real* 값) 데이터의 수학적 통계를 계산하는 함수를 제공합니다.

이 모듈은 NumPy, SciPy와 같은 제삼자 라이브러리나 Minitab, SAS 및 Matlab과 같은 통계 전문가를 대상으로 하는 완전한 기능의 통계 패키지와 경쟁하기 위한 것이 아닙니다. 그래프를 그릴 수 있는 과학 계산기의 수준을 목표로 합니다.

달리 명시되지 않는 한, 이 함수는 *int*, *float*, *decimal.Decimal* 및 *fractions.Fraction*을 지원합니다. 다른 형(숫자 계층에 있든 없든)에서의 동작은 현재 지원되지 않습니다. 여러 형의 컬렉션도 정의되지 않으며 구현에 따라 다릅니다. 입력 데이터가 혼합형으로 구성되었으면, *map()*을 사용하여 일관된 결과를 보장 할 수 있습니다, 예를 들어 `map(float, input_data)`.

Some datasets use `NaN` (not a number) values to represent missing data. Since NaNs have unusual comparison semantics, they cause surprising or undefined behaviors in the statistics functions that sort data or that count occurrences. The functions affected are `median()`, `median_low()`, `median_high()`, `median_grouped()`, `mode()`, `multimode()`, and `quantiles()`. The `NaN` values should be stripped before calling these functions:

```
>>> from statistics import median
>>> from math import isnan
>>> from itertools import filterfalse

>>> data = [20.7, float('NaN'),19.2, 18.3, float('NaN'), 14.4]
>>> sorted(data)  # 놀라운 행동을 보입니다
[20.7, nan, 14.4, 18.3, 19.2, nan]
>>> median(data)  # 예상치 못한 결과입니다
16.35

>>> sum(map(isnan, data))     # 누락된 값의 수
2
>>> clean = list(filterfalse(isnan, data))  # NaN 값을 제거합니다
>>> clean
[20.7, 19.2, 18.3, 14.4]
>>> sorted(clean)  # 정렬이 이제 예상대로 작동합니다
[14.4, 18.3, 19.2, 20.7]
>>> median(clean)       # 이제 결과가 잘 정의됩니다
18.75
```

9.7.1 평균과 중심 위치의 측정

이 함수는 모집단(population)이나 표본(sample)에서 평균이나 최빈값을 계산합니다.

mean()	데이터의 산술 평균(arithmetic mean) (“average”).
fmean()	빠른, 부동 소수점 산술 평균, 선택적 가중치 적용.
geometric_mean()	데이터의 기하 평균(geometric mean).
harmonic_mean()	데이터의 조화 평균(harmonic mean).
kde()	Estimate the probability density distribution of the data.
kde_random()	Random sampling from the PDF generated by kde().
median()	데이터의 중앙값(median) (중간값).
median_low()	데이터의 낮은 중앙값(low median).
median_high()	데이터의 높은 중앙값(high median).
median_grouped()	그룹화된 데이터의 중앙값 (50번째 백분위 수)
mode()	이산(discrete) 또는 범주(nominal) 데이터의 단일 최빈값(mode) (가장 흔한 값)
multimode()	이산 또는 범주 데이터의 최빈값(mode) (가장 흔한 값) 리스트.
quantiles()	데이터를 같은 확률을 갖는 구간으로 나눕니다.

9.7.2 분산 측정

이 함수는 모집단이나 표본이 평균값에서 벗어나는 정도를 측정합니다.

`pstdev()`	데이터의 모집단 표준 편차(population standard deviation).
`pvariance()`	데이터의 모집단 분산(population variance).
`stdev()`	데이터의 표본 표준 편차(sample standard deviation).
`variance()`	데이터의 표본 분산(sample variance).

9.7.3 Statistics for relations between two inputs

These functions calculate statistics regarding relations between two inputs.

`covariance()`	두 변수의 표본 공분산.
`correlation()`	Pearson and Spearman' s correlation coefficients.
`linear_regression()`	Slope and intercept for simple linear regression.

9.7.4 함수 세부 사항

참고: 함수에 전달되는 데이터가 정렬될 필요는 없습니다. 하지만, 읽기 쉽도록 대부분 예제는 정렬된 시퀀스를 보여줍니다.

`statistics.`**`mean`**(*data*)

시퀀스나 이터러블일 수 있는 *data*의 표본 산술 평균을 반환합니다.

산술 평균은 데이터의 합을 데이터 포인트 수로 나눈 값입니다. 흔히 "평균" 이라고 합니다만, 많은 수학적 평균 중 하나일 뿐입니다. 데이터의 중심 위치에 대한 측정(measure) 입니다.

*data*가 비어 있으면, `StatisticsError`가 발생합니다.

사용 예:

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

> **참고**
>
> 평균은 특이치의 영향을 많이 받으며, 반드시 데이터 포인트의 전형적인 예는 아닙니다. 비록 효율성은 떨어지지만, 더 강인한, 중심 경향성의 측정은, `median()`을 참조하십시오.
>
> 표본 평균은 실제 모집단 평균의 편향되지 않은(unbiased) 추정치를 제공합니다. 즉, 가능한 모든 표본에 대해 평균을 취하면, `mean(sample)`은 전체 모집단의 실제 평균에 수렴합니다. *data*가 표본이 아닌 전체 모집단을 나타낸다면, `mean(data)`는 실제 모집단 평균 μ를 계산하는 것과 동등합니다.

`statistics.`**`fmean`**(*data*, *weights=None*)

*data*를 float로 변환하고 산술 평균을 계산합니다.

`mean()` 함수보다 빠르게 실행되며 항상 `float`를 반환합니다. *data*는 시퀀스나 이터러블일 수 있습니다. 입력 data가 비어 있으면 `StatisticsError`를 발생시킵니다.

```
>>> fmean([3.5, 4.0, 5.25])
4.25
```

Optional weighting is supported. For example, a professor assigns a grade for a course by weighting quizzes at 20%, homework at 20%, a midterm exam at 30%, and a final exam at 30%:

```
>>> grades = [85, 92, 83, 91]
>>> weights = [0.20, 0.20, 0.30, 0.30]
>>> fmean(grades, weights)
87.6
```

If *weights* is supplied, it must be the same length as the *data* or a `ValueError` will be raised.

Added in version 3.8.

버전 3.11에서 변경: Added support for *weights*.

statistics.**geometric_mean**(*data*)

*data*를 float로 변환하고 기하 평균(geometric mean)을 계산합니다.

기하 평균은 값의 곱을 사용하는 *data*의 중심 경향(central tendency)이나 대표값(typical value)을 나타냅니다(합을 사용하는 산술 평균과 달리).

입력 data가 비어 있거나, 0을 포함하거나, 음수 값을 포함하면 `StatisticsError`를 발생시킵니다. *data*는 시퀀스나 이터러블일 수 있습니다.

정확한 결과를 얻기 위해 특별한 노력을 기울이지는 않습니다. (하지만, 향후 변경될 수 있습니다.)

```
>>> round(geometric_mean([54, 24, 36]), 1)
36.0
```

Added in version 3.8.

statistics.**harmonic_mean**(*data*, *weights=None*)

실숫값 숫자의 시퀀스나 이터러블인 *data*의 조화 평균(harmonic mean)을 반환합니다. *weights*가 생략되거나 `None`이면, 균등 가중치를 가정합니다.

조화 평균은 데이터의 역수의 산술 `mean()`의 역수입니다. 예를 들어, 세 *a*, *b* 및 *c* 값의 조화 평균은 `3/(1/a + 1/b + 1/c)`와 동등합니다. 값 중 하나가 0이면, 결과는 0입니다.

조화 평균은 데이터의 중심 위치의 측정인 평균의 한가지 유형입니다. 예를 들어 속도와 같은 비율(ratio)이나 율(rate)을 평균할 때 종종 적합합니다.

자동차가 40km/hr로 10km를 주행한 다음, 60km/hr로 10km를 주행한다고 가정해 봅시다. 평균 속도는 얼마입니까?

```
>>> harmonic_mean([40, 60])
48.0
```

Suppose a car travels 40 km/hr for 5 km, and when traffic clears, speeds-up to 60 km/hr for the remaining 30 km of the journey. What is the average speed?

```
>>> harmonic_mean([40, 60], weights=[5, 30])
56.0
```

*data*가 비어 있거나, 0보다 작은 값이 있거나, 가중 합이 양수가 아니면 `StatisticsError`가 발생합니다.

현재 알고리즘은 입력에서 0을 만나면 조기 종료됩니다. 이는 후속 입력의 유효성을 검사하지 않았음을 의미합니다. (이 동작은 나중에 변경될 수 있습니다.)

Added in version 3.6.

버전 3.10에서 변경: Added support for *weights*.

`statistics.kde`(*data*, *h*, *kernel='normal'*, *, *cumulative=False*)

Kernel Density Estimation (KDE): Create a continuous probability density function or cumulative distribution function from discrete samples.

The basic idea is to smooth the data using a kernel function. to help draw inferences about a population from a sample.

The degree of smoothing is controlled by the scaling parameter *h* which is called the bandwidth. Smaller values emphasize local features while larger values give smoother results.

The *kernel* determines the relative weights of the sample data points. Generally, the choice of kernel shape does not matter as much as the more influential bandwidth smoothing parameter.

Kernels that give some weight to every sample point include *normal* (*gauss*), *logistic*, and *sigmoid*.

Kernels that only give weight to sample points within the bandwidth include *rectangular* (*uniform*), *triangular*, *parabolic* (*epanechnikov*), *quartic* (*biweight*), *triweight*, and *cosine*.

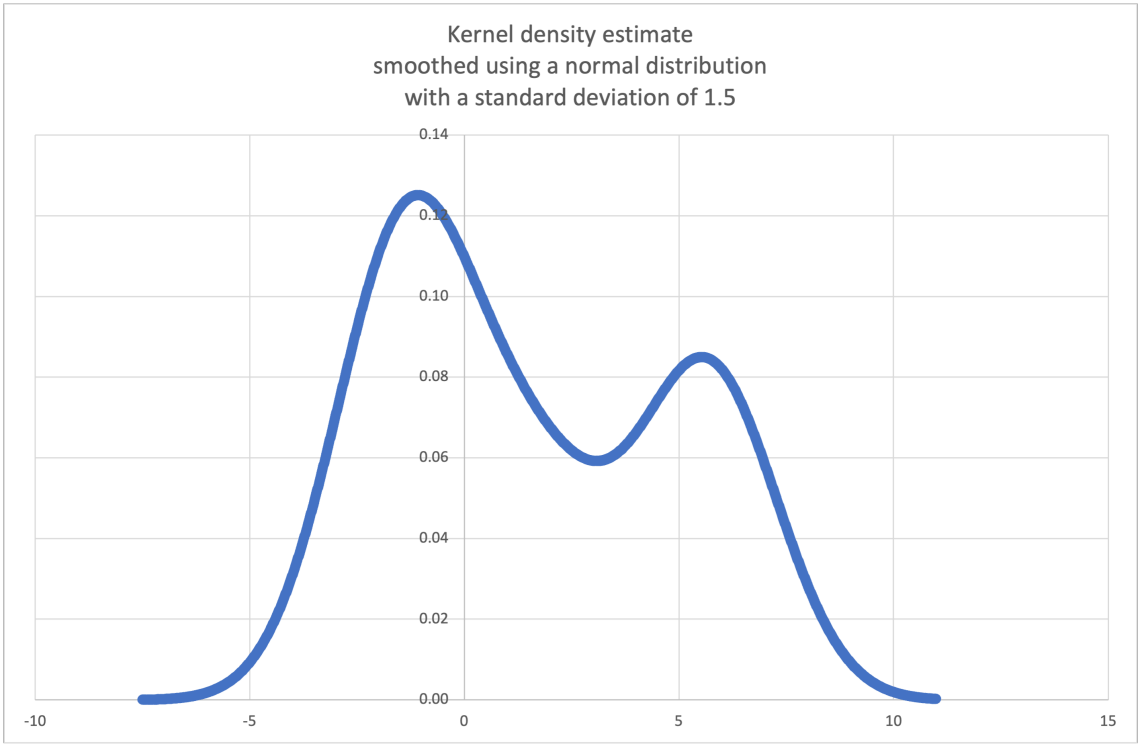
If *cumulative* is true, will return a cumulative distribution function.

data 시퀀스가 비어 있으면 `StatisticsError`를 발생시킵니다.

Wikipedia has an example where we can use `kde()` to generate and plot a probability density function estimated from a small sample:

```
>>> sample = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2]
>>> f_hat = kde(sample, h=1.5)
>>> xarr = [i/100 for i in range(-750, 1100)]
>>> yarr = [f_hat(x) for x in xarr]
```

The points in `xarr` and `yarr` can be used to make a PDF plot:



Added in version 3.13.

`statistics.kde_random`(*data*, *h*, *kernel='normal'*, *, *seed=None*)

Return a function that makes a random selection from the estimated probability density function produced by `kde(data, h, kernel)`.

Providing a *seed* allows reproducible selections. In the future, the values may change slightly as more accurate kernel inverse CDF estimates are implemented. The seed may be an integer, float, str, or bytes.

data 시퀀스가 비어 있으면 `StatisticsError`를 발생시킵니다.

Continuing the example for `kde()`, we can use `kde_random()` to generate new random selections from an estimated probability density function:

```
>>> data = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2]
>>> rand = kde_random(data, h=1.5, seed=8675309)
>>> new_selections = [rand() for i in range(10)]
>>> [round(x, 1) for x in new_selections]
[0.7, 6.2, 1.2, 6.9, 7.0, 1.8, 2.5, -0.5, -1.8, 5.6]
```

Added in version 3.13.

statistics.**median**(*data*)

일반적인 "중간 2개의 평균" 방법을 사용하여, 숫자 data의 중앙값(중간값)을 반환합니다. *data*가 비어 있으면, `StatisticsError`가 발생합니다. *data*는 시퀀스나 이터러블일 수 있습니다.

중앙값은 중심 위치에 대한 강인한 측정이며, 특이치가 있을 때 영향을 덜 받습니다. 데이터 포인트 수가 홀수면, 가운데 데이터 포인트가 반환됩니다:

```
>>> median([1, 3, 5])
3
```

데이터 포인트 수가 짝수면, 중앙값은 두 가운데 값의 평균을 취하여 보간됩니다:

```
>>> median([1, 3, 5, 7])
4.0
```

데이터가 이산(discrete)적이고, 중앙값이 실제 데이터 포인트가 아니라도 상관없을 때 적합합니다.

데이터가 순서는 있지만 (대소 비교 지원) 숫자가 아니면 (덧셈을 지원하지 않음), 대신 `median_low()`나 `median_high()`를 사용하는 것을 고려하십시오.

statistics.**median_low**(*data*)

숫자 데이터의 낮은 중앙값을 반환합니다. *data*가 비어 있으면 `StatisticsError`가 발생합니다. *data*는 시퀀스나 이터러블일 수 있습니다.

낮은 중앙값은 항상 데이터 세트의 멤버입니다. 데이터 포인트 수가 홀수이면 중간값이 반환됩니다. 짝수이면, 두 중간값 중 작은 값이 반환됩니다.

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

데이터가 이산(discrete)적이고 보간된 값이 아닌 실제 데이터 포인트를 중앙값으로 선호할 때 낮은 중앙값을 사용하십시오.

statistics.**median_high**(*data*)

데이터의 높은 중앙값을 반환합니다. *data*가 비어 있으면 `StatisticsError`가 발생합니다. *data*는 시퀀스나 이터러블일 수 있습니다.

높은 중앙값은 항상 데이터 세트의 멤버입니다. 데이터 포인트 수가 홀수이면 중간값이 반환됩니다. 짝수이면, 두 중간값 중 큰 값이 반환됩니다.

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

데이터가 이산(discrete)적이고 보간된 값이 아닌 실제 데이터 포인트를 중앙값으로 선호할 때 높은 중앙값을 사용하십시오.

statistics.**median_grouped**(*data*, *interval=1.0*)

Estimates the median for numeric data that has been grouped or binned around the midpoints of consecutive, fixed-width intervals.

The *data* can be any iterable of numeric data with each value being exactly the midpoint of a bin. At least one value must be present.

The *interval* is the width of each bin.

For example, demographic information may have been summarized into consecutive ten-year age groups with each group being represented by the 5-year midpoints of the intervals:

```
>>> from collections import Counter
>>> demographics = Counter({
...    25: 172,   # 20 에서 30 세
...    35: 484,   # 30 에서 40 세
...    45: 387,   # 40 에서 50 세
...    55:  22,   # 50 에서 60 세
...    65:   6,   # 60 에서 70 세
... })
...
```

The 50th percentile (median) is the 536th person out of the 1071 member cohort. That person is in the 30 to 40 year old age group.

The regular *median()* function would assume that everyone in the tricenarian age group was exactly 35 years old. A more tenable assumption is that the 484 members of that age group are evenly distributed between 30 and 40. For that, we use *median_grouped()*:

```
>>> data = list(demographics.elements())
>>> median(data)
35
>>> round(median_grouped(data, interval=10), 1)
37.5
```

The caller is responsible for making sure the data points are separated by exact multiples of *interval*. This is essential for getting a correct result. The function does not check this precondition.

Inputs may be any numeric type that can be coerced to a float during the interpolation step.

statistics.**mode**(*data*)

이산(discrete)적이거나 범주(nominal)적인 *data*에서 가장 흔한 단일 데이터 포인트를 반환합니다. 최빈값(mode)은 (존재할 때) 가장 흔한 값이며 중심 위치의 측정으로 기능합니다.

같은 빈도의 여러 최빈값이 있으면, *data*에서 처음 발견된 첫 번째 값을 반환합니다. 여러 최빈값 중 가장 작거나 가장 큰 값이 필요하면 대신 `min(multimode(data))` 나 `max(multimode(data))` 를 사용하십시오. 입력 *data*가 비어 있으면, *StatisticsError*가 발생합니다.

`mode`는 이산 데이터를 가정하고 단일 값을 반환합니다. 이것이 학교에서 일반적으로 가르치는 최빈값의 표준적인 처리입니다:

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

최빈값은 범주(nominal)적 (숫자가 아닌) 데이터에도 적용되는 이 패키지에 있는 유일한 통계라는 점에서 특별합니다:

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

Only hashable inputs are supported. To handle type *set*, consider casting to *frozenset*. To handle type *list*, consider casting to *tuple*. For mixed or nested inputs, consider using this slower quadratic algorithm that only depends on equality tests: `max(data, key=data.count)`.

버전 3.8에서 변경: 이제 첫 번째 최빈값을 모드를 반환하여 다봉(multimodal) 데이터 세트를 처리합니다. 이전에는, 둘 이상의 최빈값이 발견되면 *StatisticsError*가 발생했습니다.

`statistics.`**`multimode`**(*data*)

*data*에서 먼저 발견되는 순서대로 가장 자주 등장하는 값의 리스트를 반환합니다. 여러 최빈값이 있으면 둘 이상의 결과를 반환하고, *data*가 비어 있으면 빈 리스트를 반환합니다:

```
>>> multimode('aabbbbccddddeeffffgg')
['b', 'd', 'f']
>>> multimode('')
[]
```

Added in version 3.8.

`statistics.`**`pstdev`**(*data*, *mu=None*)

모집단 표준 편차(모집단 분산의 제곱근)를 반환합니다. 인자와 기타 세부 사항은 *pvariance()*를 참조하십시오.

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

`statistics.`**`pvariance`**(*data*, *mu=None*)

실수 숫자의 비어있지 않은 시퀀스나 이터러블인 *data*의 모집단 분산을 반환합니다. 분산(또는 평균에 대한 이차 모멘트)은 데이터 변동성(퍼진 정도)의 측정입니다. 큰 분산은 데이터가 퍼져 있음을 나타냅니다; 작은 분산은 평균 주변에 군집되어 있음을 나타냅니다.

선택적 두 번째 인자 *mu*가 제공되면, *data*의 모집단 평균이어야 합니다. 평균이 아닌 점을 기준으로 이차 모멘트를 계산하는 데에도 사용할 수 있습니다. 누락되었거나 `None`(기본값)이면, 산술 평균이 자동으로 계산됩니다.

이 함수를 사용하여 전체 모집단의 분산을 계산하십시오. 표본으로 분산을 추정하려면, 일반적으로 *variance()* 함수가 더 좋은 선택입니다.

*data*가 비어 있으면 *StatisticsError*를 발생시킵니다.

예:

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

데이터의 평균을 이미 계산했다면, 재계산을 피하고자 선택적인 두 번째 인자 *mu*로 전달할 수 있습니다:

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

Decimal과 Fraction이 지원됩니다:

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('24.815')

>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

> **참고**
>
> 전체 모집단으로 호출하면, 모집단 분산 σ^2을 줍니다. 대신 표본으로 호출하면, 편향된(biased) 표본 분산 s^2이 됩니다, N 자유도의 분산이라고도 합니다.
>
> 실제 모집단 평균 μ를 어떻게든 알고 있다면, 알려진 모집단 평균을 두 번째 인자로 지정해서, 이 함수를 사용하여 표본의 분산을 계산할 수 있습니다. 데이터 포인트가 모집단의 무작위 표본이면, 결과는 모집단 분산의 편향 없는(unbiased) 추정치가 됩니다.

statistics.**stdev**(*data*, *xbar=None*)

표본 표준 편차(표본 분산의 제곱근)를 반환합니다. 인자와 기타 세부 사항은 `variance()`를 참조하십시오.

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

statistics.**variance**(*data*, *xbar=None*)

적어도 두 개의 실수 숫자를 제공하는 이터러블인 *data*의 표본 분산을 반환합니다. 분산(또는 평균에 대한 이차 모멘트)은 데이터 변동성(퍼진 정도)의 측정입니다. 큰 분산은 데이터가 퍼져 있음을 나타냅니다; 작은 분산은 평균 주변에 군집되어 있음을 나타냅니다.

선택적 두 번째 인자 *xbar*가 제공되면, *data*의 표본 평균이어야 합니다. 누락되었거나 `None`(기본값)이면, 평균은 자동으로 계산됩니다.

데이터가 모집단의 표본이면 이 함수를 사용하십시오. 전체 모집단의 분산을 계산하려면, `pvariance()`를 참조하십시오.

*data*의 두 개 미만의 값을 갖고 있으면 `StatisticsError`를 발생시킵니다.

예:

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

데이터의 표본 평균을 이미 계산했다면, 재계산을 피하고자 선택적인 두 번째 인자 *xbar*로 전달할 수 있습니다:

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

이 함수는 실제 평균을 *xbar*로 전달했는지 확인하지 않습니다. *xbar*에 임의의 값을 사용하면 결과가 유효하지 않거나 불가능한 값일 수 있습니다.

Decimal과 Fraction 값이 지원됩니다:

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('31.01875')

>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

> **참고**
>
> 이것은 베셀 보정(Bessel' s correction)을 적용한 표본 분산 s^2 입니다, N-1 자유도의 분산이라고도 합니다. 데이터 포인트가 대표적(예를 들어, 독립적이고 동일하게 분포된)이라면, 결과는 실제

> 모집단 분산의 편향 없는(unbiased) 추정치가 되어야 합니다.
>
> 실제 모집단 평균 μ를 어떻게든 알고 있다면 *mu* 매개 변수로 `pvariance()` 함수에 전달하여 표본의 분산을 구해야 합니다.

`statistics.`**`quantiles`**(*data*, **, n=4, method='exclusive'*)

*data*를 같은 확률을 갖는 *n* 개의 연속 구간으로 나눕니다. 구간을 분할하는 `n - 1`개의 절단 점(cut point) 리스트를 반환합니다.

사분위는 *n*을 4로 설정하십시오 (기본값). 십분위는 *n*을 10으로 설정하십시오. 백분위는 *n*을 100으로 설정하십시오. 그러면 *data*를 100개의 같은 크기 그룹으로 분할하는 99개의 절단 점이 제공됩니다. *n*이 1 미만이면 `StatisticsError`를 발생시킵니다.

*data*는 표본 데이터를 포함하는 모든 이터러블일 수 있습니다. 의미 있는 결과를 얻으려면, *data*의 데이터 포인트 수가 *n*보다 커야 합니다. 최소한 하나의 데이터 포인트도 없으면 `StatisticsError`를 발생시킵니다.

절단 점은 가장 가까운 두 개의 데이터 포인트에서 선형 보간됩니다. 예를 들어, 절단 점이 두 표본 값 `100`과 `112` 사이의 거리로 1/3 지점에 해당하면, 절단 점은 `104`로 평가됩니다.

균등 분위(quantile) 계산 방법(*method*)은 *data*가 모집단에서 가능한 최젓값과 최곳값을 포함하는지 제외하는지에 따라 달라질 수 있습니다.

기본 *method*는 "exclusive"이며, 표본에서 발견되는 것보다 더 극단적인 값을 가질 수 있는 모집단에서 표본 추출된 데이터에 사용됩니다. *m* 개의 정렬된 데이터 포인트의 *i*-번째 아래로 떨어지는 모집단 부분은 `i / (m + 1)`로 계산됩니다. 9개의 표본 값을 주면, 이 방법은 그들을 정렬한 다음, 다음과 같은 백분위를 할당합니다: 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%.

*method*를 "inclusive"로 설정하는 것은 모집단 데이터를 기술하거나 모집단의 가장 극단적인 값을 포함하는 것으로 알려진 표본에 사용됩니다. *data*의 최솟값은 0번째 백분위 수로 취급되고 최댓값은 100번째 백분위 수로 취급됩니다. *m* 개의 정렬된 데이터 포인트의 *i*-번째 아래로 떨어지는 모집단 부분은 `(i - 1) / (m - 1)`로 계산됩니다. 11개의 표본 값을 주면, 이 방법은 그들을 정렬한 다음, 다음과 같은 백분위를 할당합니다: 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%.

```
# 경험적으로 표본 추출된 데이터에 대한 십 분위 절단 점
>>> data = [105, 129, 87, 86, 111, 111, 89, 81, 108, 92, 110,
...         100, 75, 105, 103, 109, 76, 119, 99, 91, 103, 129,
...         106, 101, 84, 111, 74, 87, 86, 103, 103, 106, 86,
...         111, 75, 87, 102, 121, 111, 88, 89, 101, 106, 95,
...         103, 107, 101, 81, 109, 104]
>>> [round(q, 1) for q in quantiles(data, n=10)]
[81.0, 86.2, 89.0, 99.4, 102.5, 103.6, 106.0, 109.8, 111.0]
```

Added in version 3.8.

버전 3.13에서 변경: No longer raises an exception for an input with only a single data point. This allows quantile estimates to be built up one sample point at a time becoming gradually more refined with each new data point.

`statistics.`**`covariance`**(*x, y, /*)

Return the sample covariance of two inputs *x* and *y*. Covariance is a measure of the joint variability of two inputs.

Both inputs must be of the same length (no less than two), otherwise `StatisticsError` is raised.

예:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> y = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> covariance(x, y)
0.75
>>> z = [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> covariance(x, z)
-7.5
>>> covariance(z, x)
-7.5
```

Added in version 3.10.

statistics.**correlation**(*x*, *y*, /, **, method='linear'*)

Return the Pearson' s correlation coefficient for two inputs. Pearson' s correlation coefficient *r* takes values between -1 and +1. It measures the strength and direction of a linear relationship.

If *method* is "ranked", computes Spearman' s rank correlation coefficient for two inputs. The data is replaced by ranks. Ties are averaged so that equal values receive the same rank. The resulting coefficient measures the strength of a monotonic relationship.

Spearman' s correlation coefficient is appropriate for ordinal data or for continuous data that doesn' t meet the linear proportion requirement for Pearson' s correlation coefficient.

Both inputs must be of the same length (no less than two), and need not to be constant, otherwise `StatisticsError` is raised.

Example with Kepler' s laws of planetary motion:

```
>>> # 수성, 금성, 지구, 화성, 목성, 토성, 천왕성 및 해왕성
>>> orbital_period = [88, 225, 365, 687, 4331, 10_756, 30_687, 60_190]    #␣
↪days
>>> dist_from_sun = [58, 108, 150, 228, 778, 1_400, 2_900, 4_500] # million km

>>> # 완벽한 단조 관계가 존재함을 보입니다
>>> correlation(orbital_period, dist_from_sun, method='ranked')
1.0

>>> # 선형 관계가 불완전함을 관찰합니다
>>> round(correlation(orbital_period, dist_from_sun), 4)
0.9882

>>> # 케플러의 제3법칙을 시연합니다: 궤도 주기의 제곱과
>>> # 태양으로부터의 거리 세제곱 간에 선형 상관관계가
>>> # 존재합니다.
>>> period_squared = [p * p for p in orbital_period]
>>> dist_cubed = [d * d * d for d in dist_from_sun]
>>> round(correlation(period_squared, dist_cubed), 4)
1.0
```

Added in version 3.10.

버전 3.12에서 변경: Added support for Spearman' s rank correlation coefficient.

statistics.**linear_regression**(*x*, *y*, /, **, proportional=False*)

Return the slope and intercept of simple linear regression parameters estimated using ordinary least squares. Simple linear regression describes the relationship between an independent variable *x* and a dependent variable *y* in terms of this linear function:

$$y = slope * x + intercept + noise$$

where `slope` and `intercept` are the regression parameters that are estimated, and `noise` represents the variability of the data that was not explained by the linear regression (it is equal to the difference between predicted and actual values of the dependent variable).

Both inputs must be of the same length (no less than two), and the independent variable *x* cannot be constant; otherwise a `StatisticsError` is raised.

For example, we can use the release dates of the Monty Python films to predict the cumulative number of Monty Python films that would have been produced by 2019 assuming that they had kept the pace.

```
>>> year = [1971, 1975, 1979, 1982, 1983]
>>> films_total = [1, 2, 3, 4, 5]
>>> slope, intercept = linear_regression(year, films_total)
>>> round(slope * 2019 + intercept)
16
```

If *proportional* is true, the independent variable *x* and the dependent variable *y* are assumed to be directly proportional. The data is fit to a line passing through the origin. Since the *intercept* will always be 0.0, the underlying linear function simplifies to:

*y = slope * x + noise*

Continuing the example from `correlation()`, we look to see how well a model based on major planets can predict the orbital distances for dwarf planets:

```
>>> model = linear_regression(period_squared, dist_cubed, proportional=True)
>>> slope = model.slope

>>> # 왜행성:              명왕성,    에리스,     마케마케,    하우메아,    세레스
>>> orbital_periods = [90_560, 204_199, 111_845, 103_410, 1_680]  # 일
>>> predicted_dist = [math.cbrt(slope * (p * p)) for p in orbital_periods]
>>> list(map(round, predicted_dist))
[5912, 10166, 6806, 6459, 414]

>>> [5_906, 10_152, 6_796, 6_450, 414]  # 백만 km 단위의 실제 거리
[5906, 10152, 6796, 6450, 414]
```

Added in version 3.10.

버전 3.11에서 변경: Added support for *proportional*.

9.7.5 예외

하나의 예외가 정의됩니다:

exception statistics.**StatisticsError**

통계 관련 예외를 위한 `ValueError`의 서브 클래스.

9.7.6 `NormalDist` 객체

`NormalDist`는 무작위 변수의 정규 분포를 만들고 조작하기 위한 도구입니다. 데이터 측정의 평균과 표준 편차를 단일 엔티티로 취급하는 클래스입니다.

정규 분포는 중심 극한 정리(Central Limit Theorem)에서 도출되며 통계에서 광범위하게 응용됩니다.

class statistics.**NormalDist**(*mu=0.0, sigma=1.0*)

*mu*가 산술 평균을 나타내고 *sigma*가 표준 편차를 나타내는 새 *NormalDist* 객체를 반환합니다.

*sigma*가 음수이면 `StatisticsError`를 발생시킵니다.

mean

정규 분포의 산술 평균에 대한 읽기 전용 프로퍼티.

median

정규 분포의 중앙값에 대한 읽기 전용 프로퍼티.

mode

정규 분포의 최빈값에 대한 읽기 전용 프로퍼티.

stdev

정규 분포의 표준 편차에 대한 읽기 전용 프로퍼티.

variance

정규 분포의 분산에 대한 읽기 전용 프로퍼티. 표준 편차의 제곱과 같습니다.

classmethod from_samples(*data*)

`fmean()`과 `stdev()`를 사용해서 *data*에서 추정된 *mu*와 *sigma* 매개 변수로 정규 분포 인스턴스를 만듭니다.

*data*는 임의의 이터러블일 수 있으며 `float` 형으로 변환될 수 있는 값으로 구성되어야 합니다. *data*가 두 개 이상의 값을 포함하지 않으면 `StatisticsError`를 발생시키는데, 중심 값을 추정하는 데 적어도 한 점이 필요하고 분산을 추정하는 데 적어도 두 점이 필요하기 때문입니다.

samples(*n*, *, *seed=None*)

주어진 평균과 표준 편차로 *n* 개의 무작위 표본을 생성합니다. `float` 값의 `list`를 반환합니다.

*seed*가 제공되면, 하부 난수 생성기의 새 인스턴스를 만듭니다. 이는 다중 스레딩 문맥에서도, 재현 가능한 결과를 만드는 데 유용합니다.

버전 3.13에서 변경.

Switched to a faster algorithm. To reproduce samples from previous versions, use `random.seed()` and `random.gauss()`.

pdf(*x*)

확률 밀도 함수(pdf)를 사용하여, 무작위 변수 *X*가 주어진 값 *x*에 가까울 상대적 가능도(likelihood)를 계산합니다. 수학적으로, 비율 `P(x <= X < x+dx) / dx`의 *dx*가 0으로 접근할 때의 극한값입니다.

상대적 가능도는 좁은 구간에 표본이 발생할 수 있는 확률을 구간의 너비로 나눈 값으로 계산됩니다 (그래서 "밀도"라고 합니다). 가능도는 다른 점에 상대적이기 때문에, `1.0`보다 클 수 있습니다.

cdf(*x*)

누적 분포 함수 (cdf)를 사용하여, 무작위 변수 *X*가 *x*보다 작거나 같을 확률을 계산합니다. 수학적으로, `P(X <= x)` 라고 씁니다.

inv_cdf(*p*)

분위 함수(quantile function)나 백분위 수(percent-point) 함수라고도 하는 역 누적 분포 함수를 계산합니다. 수학적으로, `x : P(X <= x) = p`라고 씁니다.

변수가 *x*보다 작거나 같을 확률이 주어진 확률 *p*와 같아지도록 하는 무작위 변수 *X*의 값 *x*를 찾습니다.

overlap(*other*)

두 정규 확률 분포 간의 일치를 측정합니다. 두 확률 밀도 함수가 겹치는 영역의 면적을 제공하는 0.0과 1.0 사이의 값을 반환합니다.

quantiles(*n=4*)

정규 분포를 같은 확률을 갖는 *n* 개의 연속 구간으로 나눕니다. 구간을 분할하는 (n - 1) 개의 절단점(cut point) 리스트를 반환합니다.

사분위는 *n*을 4로 설정하십시오 (기본값). 십분위는 *n*을 10으로 설정하십시오. 백분위는 *n*을 100으로 설정하십시오, 그러면 정규 분포를 100개의 같은 크기 그룹으로 분할하는 99개의 절단점이 제공됩니다.

zscore(*x*)

정규 분포의 평균 위나 아래의 표준 편차 수로 *x*를 설명하는 표준 점수(Standard Score)를 계산합니다: `(x - mean) / stdev`.

Added in version 3.9.

`NormalDist`의 인스턴스는 상수에 의한 덧셈, 뺄셈, 곱셈 및 나눗셈을 지원합니다. 이러한 연산은 이동(translation)과 확대(scaling)에 사용됩니다. 예를 들면:

```
>>> temperature_february = NormalDist(5, 2.5)              # 섭씨
>>> temperature_february * (9/5) + 32                      # 화씨
NormalDist(mu=41.0, sigma=4.5)
```

상수를 *NormalDist*의 인스턴스로 나누는 것은 결과가 정규 분포가 되지 않기 때문에 지원되지 않습니다.

정규 분포는 독립 변수의 가산(additive) 효과에서 발생하므로, 두 독립된 정규 분포 무작위 변수를 더하고 빼는 것은 *NormalDist*의 인스턴스로 나타낼 수 있습니다. 예를 들면:

```
>>> birth_weights = NormalDist.from_samples([2.5, 3.1, 2.1, 2.4, 2.7, 3.5])
>>> drug_effects = NormalDist(0.4, 0.15)
>>> combined = birth_weights + drug_effects
>>> round(combined.mean, 1)
3.1
>>> round(combined.stdev, 1)
0.5
```

Added in version 3.8.

9.7.7 예제와 조리법

고전적인 확률 문제

*NormalDist*는 고전적인 확률 문제를 쉽게 해결합니다.

예를 들어, 점수가 평균 1060이고 표준 편차가 195인 정규 분포를 보이는 SAT 시험의 역사적 데이터를 줄 때, 시험 점수가 1100에서 1200 사이인 학생들의 백분율을 결정하십시오. 가장 가까운 정수로 반올림하십시오:

```
>>> sat = NormalDist(1060, 195)
>>> fraction = sat.cdf(1200 + 0.5) - sat.cdf(1100 - 0.5)
>>> round(fraction * 100.0, 1)
18.4
```

SAT 점수의 사분위 수(quartiles)와 십분위 수(deciles)를 찾으십시오:

```
>>> list(map(round, sat.quantiles()))
[928, 1060, 1192]
>>> list(map(round, sat.quantiles(n=10)))
[810, 896, 958, 1011, 1060, 1109, 1162, 1224, 1310]
```

Monte Carlo inputs for simulations

분석적으로 풀기 쉽지 않은 모델의 분포를 추정하기 위해, *NormalDist*는 몬테카를로 시뮬레이션(Monte Carlo simulation)을 위한 입력 표본을 생성 할 수 있습니다:

```
>>> def model(x, y, z):
...     return (3*x + 7*x*y - 5*y) / (11 * z)
...
>>> n = 100_000
>>> X = NormalDist(10, 2.5).samples(n, seed=3652260728)
>>> Y = NormalDist(15, 1.75).samples(n, seed=4582495471)
>>> Z = NormalDist(50, 1.25).samples(n, seed=6582483453)
>>> quantiles(map(model, X, Y, Z))
[1.4591308524824727, 1.8035946855390597, 2.175091447274739]
```

Approximating binomial distributions

표본 크기가 크고 성공적인 시행의 확률이 50%에 가까울 때 정규 분포를 사용하여 이항 분포(Binomial distributions)를 근사할 수 있습니다.

예를 들어, 오픈 소스 회의에는 750명의 참석자와 500명 정원의 방 두 개가 있습니다. 파이썬과 루비에 대한 발표가 있습니다. 이전 회의에서는, 참석자의 65%가 파이썬 발표를 듣는 것을 선호했습니다. 모집단 선호도가 변경되지 않았다고 가정할 때, 파이썬 방이 정원 한도 내에 머무를 확률은 얼마입니까?

```
>>> n = 750             # 표본 크기
>>> p = 0.65            # 파이썬 선호도
>>> q = 1.0 - p         # 루비 선호도
>>> k = 500             # 방 정원

>>> # 누적 정규 분포를 사용한 근사
>>> from math import sqrt
>>> round(NormalDist(mu=n*p, sigma=sqrt(n*p*q)).cdf(k + 0.5), 4)
0.8402

>>> # 누적 이항 분포를 사용한 정확한 해법
>>> from math import comb, fsum
>>> round(fsum(comb(n, r) * p**r * q**(n-r) for r in range(k+1)), 4)
0.8402

>>> # 시뮬레이션을 사용한 근사
>>> from random import seed, binomialvariate
>>> seed(8675309)
>>> mean(binomialvariate(n, p) <= k for i in range(10_000))
0.8406
```

Naive bayesian classifier

정규 분포는 기계 학습 문제에서 흔히 등장합니다.

위키백과에는 나이브 베이즈 분류기(Naive Bayesian Classifier)의 멋진 예가 있습니다. 문제는 키, 몸무게 및 발 크기를 포함하는 정규 분포된 피처(feature)들로부터 사람의 성별을 예측하는 것입니다.

우리는 8명을 측정한 훈련 데이터 집합을 받았습니다. 측정값은 정규 분포로 가정되므로, `NormalDist`로 데이터를 요약합니다:

```
>>> height_male = NormalDist.from_samples([6, 5.92, 5.58, 5.92])
>>> height_female = NormalDist.from_samples([5, 5.5, 5.42, 5.75])
>>> weight_male = NormalDist.from_samples([180, 190, 170, 165])
>>> weight_female = NormalDist.from_samples([100, 150, 130, 150])
>>> foot_size_male = NormalDist.from_samples([12, 11, 12, 10])
>>> foot_size_female = NormalDist.from_samples([6, 8, 7, 9])
```

다음으로, 피처 측정은 알려졌지만, 성별을 모르는 새로운 사람을 만납니다:

```
>>> ht = 6.0        # 키
>>> wt = 130        # 몸무게
>>> fs = 8          # 발 크기
```

남성이나 여성일 50%의 사전 확률(prior probability)로 시작하여, 사전 확률에 주어진 성별이 피처 측정을 줄 우도(likelihood)를 곱해서 사후 확률(posterior)을 계산합니다.:

```
>>> prior_male = 0.5
>>> prior_female = 0.5
>>> posterior_male = (prior_male * height_male.pdf(ht) *
...                   weight_male.pdf(wt) * foot_size_male.pdf(fs))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> posterior_female = (prior_female * height_female.pdf(ht) *
...                     weight_female.pdf(wt) * foot_size_female.pdf(fs))
```

최종 예측은 가장 큰 사후 확률(posterior)이 됩니다. 이것을 최대 사후 확률(maximum a posteriori) 또는 MAP 이라고 합니다.:

```
>>> 'male' if posterior_male > posterior_female else 'female'
'female'
```

CHAPTER 10

함수형 프로그래밍 모듈

이 장에서 설명하는 모듈은 함수형 프로그래밍 스타일을 지원하는 함수 및 클래스와 콜러블에 대한 일반 연산을 제공합니다.

이 장에서는 다음 모듈에 관해 설명합니다:

10.1 `itertools` — Functions creating iterators for efficient looping

이 모듈은 APL, Haskell 및 SML의 구성물들에서 영감을 얻은 여러 이터레이터 빌딩 블록을 구현합니다. 각각을 파이썬에 적합한 형태로 개선했습니다.

이 모듈은 자체적으로 혹은 조합하여 유용한 빠르고 메모리 효율적인 도구의 핵심 집합을 표준화합니다. 함께 모여, 순수 파이썬에서 간결하고 효율적으로 특수화된 도구를 구성할 수 있도록 하는 "이터레이터 대수(iterator algebra)" 를 형성합니다.

예를 들어, SML은 테이블 화 도구를 제공합니다: 시퀀스 `f(0), f(1), ...`를 생성하는 `tabulate(f)`. `map()`과 `count()`를 결합하여 `map(f, count())`를 형성해서 파이썬에서도 같은 효과를 얻을 수 있습니다.

무한 이터레이터:

이터레이터	인자	결과	예
`count()`	[start[, step]]	start, start+step, start+2*step, …	`count(10) → 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') → A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … 끝없이 또는 최대 n 번	`repeat(10, 3) → 10 10 10`

가장 짧은 입력 시퀀스에서 종료되는 이터레이터:

이터레이터	인자	결과	예
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, ⋯	accumulate([1,2,3,4,5]) → 1 3 6 10 15
batched()	p, n	(p0, p1, ⋯, p_n-1), ⋯	batched('ABCDEFG', n=2) → AB CD EF G
chain()	p, q, ⋯	p0, p1, ⋯ plast, q0, q1, ⋯	chain('ABC', 'DEF') → A B C D E F
chain.from_iterabl	iterable	p0, p1, ⋯ plast, q0, q1, ⋯	chain.from_iterable(['ABC', 'DEF']) → A B C D E F
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ⋯	compress('ABCDEF', [1,0,1,0,1,1]) → A C E F
dropwhile()	predicate, seq	seq[n], seq[n+1], starting when predicate fails	dropwhile(lambda x: x<5, [1,4,6,3,8]) → 6 3 8
filterfalse(	predicate, seq	elements of seq where predicate(elem) fails	filterfalse(lambda x: x<5, [1,4,6,3,8]) → 6 8
groupby()	iterable[, key]	key(v)의 값으로 그룹화된 서브 이터레이터들	groupby(['A','B','DEF'], len) → (1, A B) (3, DEF)
islice()	seq, [start,] stop [, step]	seq[start:stop:step]의 요소들	islice('ABCDEFG', 2, None) → C D E F G
pairwise()	iterable	(p[0], p[1]), (p[1], p[2])	pairwise('ABCDEFG') → AB BC CD DE EF FG
starmap()	func, seq	func(*seq[0]), func(*seq[1]), ⋯	starmap(pow, [(2,5), (3,2), (10,3)]) → 32 9 1000
takewhile()	predicate, seq	seq[0], seq[1], until predicate fails	takewhile(lambda x: x<5, [1,4,6,3,8]) → 1 4
tee()	it, n	it1, it2, ⋯ itn 하나의 이터레이터를 n개의 이터레이터로 나눕니다	tee('ABC', 2) → A B C, A B C
zip_longest(	p, q, ⋯	(p[0], q[0]), (p[1], q[1]), ⋯	zip_longest('ABCD', 'xy', fillvalue='-') → Ax By C- D-

조합형 이터레이터:

이터레이터	인자	결과
product()	p, q, ⋯ [repeat=1]	데카르트 곱(cartesian product), 중첩된 for 루프와 동등합니다
permutations()	p[, r]	r-길이 튜플들, 모든 가능한 순서, 반복되는 요소 없음
combinations()	p, r	r-길이 튜플들, 정렬된 순서, 반복되는 요소 없음
combinations_with_replaceme	p, r	r-길이 튜플들, 정렬된 순서, 반복되는 요소 있음

예	결과
product('ABCD', repeat=2)	AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
permutations('ABCD', 2)	AB AC AD BA BC BD CA CB CD DA DB DC
combinations('ABCD', 2)	AB AC AD BC BD CD
combinations_with_replacement('ABCD', 2)	AA AB AC AD BB BC BD CC CD DD

10.1.1 Itertool Functions

The following functions all construct and return iterators. Some provide streams of infinite length, so they should only be accessed by functions or loops that truncate the stream.

itertools.**accumulate**(*iterable*[, *function*, *, *initial=None*])

Make an iterator that returns accumulated sums or accumulated results from other binary functions.

The *function* defaults to addition. The *function* should accept two arguments, an accumulated total and a value from the *iterable*.

If an *initial* value is provided, the accumulation will start with that value and the output will have one more element than the input iterable.

대략 다음과 동등합니다:

```
def accumulate(iterable, function=operator.add, *, initial=None):
    'Return running totals'
    # accumulate([1,2,3,4,5]) → 1 3 6 10 15
    # accumulate([1,2,3,4,5], initial=100) → 100 101 103 106 110 115
    # accumulate([1,2,3,4,5], operator.mul) → 1 2 6 24 120

    iterator = iter(iterable)
    total = initial
    if initial is None:
        try:
            total = next(iterator)
        except StopIteration:
            return

    yield total
    for element in iterator:
        total = function(total, element)
        yield total
```

To compute a running minimum, set *function* to `min()`. For a running maximum, set *function* to `max()`. Or for a running product, set *function* to `operator.mul()`. To build an amortization table, accumulate the interest and apply payments:

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, max))              # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]
>>> list(accumulate(data, operator.mul))     # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]

# Amortize a 5% loan of 1000 with 10 annual payments of 90
>>> update = lambda balance, payment: round(balance * 1.05) - payment
>>> list(accumulate(repeat(90, 10), update, initial=1_000))
[1000, 960, 918, 874, 828, 779, 728, 674, 618, 559, 497]
```

최종 누적값만 반환하는 유사한 함수에 대해서는 `functools.reduce()`를 참조하십시오.

Added in version 3.2.

버전 3.3에서 변경: Added the optional *function* parameter.

버전 3.8에서 변경: 선택적 *initial* 매개 변수를 추가했습니다.

itertools.**batched**(*iterable*, *n*, *, *strict=False*)

Batch data from the *iterable* into tuples of length *n*. The last batch may be shorter than *n*.

If *strict* is true, will raise a `ValueError` if the final batch is shorter than *n*.

Loops over the input iterable and accumulates data into tuples up to size *n*. The input is consumed lazily, just enough to fill a batch. The result is yielded as soon as the batch is full or when the input iterable is exhausted:

```
>>> flattened_data = ['roses', 'red', 'violets', 'blue', 'sugar', 'sweet']
>>> unflattened = list(batched(flattened_data, 2))
>>> unflattened
[('roses', 'red'), ('violets', 'blue'), ('sugar', 'sweet')]
```

대략 다음과 동등합니다:

```
def batched(iterable, n, *, strict=False):
    # batched('ABCDEFG', 2) → AB CD EF G
    if n < 1:
        raise ValueError('n must be at least one')
    iterator = iter(iterable)
    while batch := tuple(islice(iterator, n)):
        if strict and len(batch) != n:
            raise ValueError('batched(): incomplete batch')
        yield batch
```

Added in version 3.12.

버전 3.13에서 변경: Added the *strict* option.

itertools.**chain**(**iterables*)

Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. This combines multiple data sources into a single iterator. Roughly equivalent to:

```
def chain(*iterables):
    # chain('ABC', 'DEF') → A B C D E F
    for iterable in iterables:
        yield from iterable
```

classmethod chain.**from_iterable**(*iterable*)

`chain()`의 대체 생성자. 게으르게 평가되는 단일 이터러블 인자에서 연쇄 입력을 가져옵니다. 대략 다음과 동등합니다:

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) → A B C D E F
    for iterable in iterables:
        yield from iterable
```

itertools.**combinations**(*iterable*, *r*)

입력 *iterable*에서 요소의 길이 *r* 서브 시퀀스들을 반환합니다.

The output is a subsequence of `product()` keeping only entries that are subsequences of the *iterable*. The length of the output is given by `math.comb()` which computes `n! / r! / (n - r)!` when `0 ≤ r ≤ n` or zero when `r > n`.

The combination tuples are emitted in lexicographic order according to the order of the input *iterable*. If the input *iterable* is sorted, the output tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. If the input elements are unique, there will be no repeated values within each combination.

대략 다음과 동등합니다:

```
def combinations(iterable, r):
    # combinations('ABCD', 2) → AB AC AD BC BD CD
    # combinations(range(4), 3) → 012 013 023 123
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))

    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

itertools.**combinations_with_replacement**(*iterable*, *r*)

입력 *iterable*에서 요소의 길이 *r* 서브 시퀀스들을 반환하는데, 개별 요소를 두 번 이상 반복할 수 있습니다.

The output is a subsequence of `product()` that keeps only entries that are subsequences (with possible repeated elements) of the *iterable*. The number of subsequence returned is `(n + r - 1)! / r! / (n - 1)!` when `n > 0`.

The combination tuples are emitted in lexicographic order according to the order of the input *iterable*. if the input *iterable* is sorted, the output tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. If the input elements are unique, the generated combinations will also be unique.

대략 다음과 동등합니다:

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) ▷ AA AB AC BB BC CC

    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r

    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

Added in version 3.1.

itertools.**compress**(*data*, *selectors*)

Make an iterator that returns elements from *data* where the corresponding element in *selectors* is true. Stops when either the *data* or *selectors* iterables have been exhausted. Roughly equivalent to:

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) □ A C E F
    return (datum for datum, selector in zip(data, selectors) if selector)
```

Added in version 3.1.

itertools.**count**(*start=0*, *step=1*)

Make an iterator that returns evenly spaced values beginning with *start*. Can be used with `map()` to generate consecutive data points or with `zip()` to add sequence numbers. Roughly equivalent to:

```
def count(start=0, step=1):
    # count(10) □ 10 11 12 13 14 ...
    # count(2.5, 0.5) □ 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

When counting with floating-point numbers, better accuracy can sometimes be achieved by substituting multiplicative code such as: `(start + step * i for i in count())`.

버전 3.1에서 변경: *step* 인자를 추가하고 정수가 아닌 인자를 허용했습니다.

itertools.**cycle**(*iterable*)

Make an iterator returning elements from the *iterable* and saving a copy of each. When the iterable is exhausted, return elements from the saved copy. Repeats indefinitely. Roughly equivalent to:

```
def cycle(iterable):
    # cycle('ABCD') □ A B C D A B C D A B C D ...

    saved = []
    for element in iterable:
        yield element
        saved.append(element)

    while saved:
        for element in saved:
            yield element
```

This itertool may require significant auxiliary storage (depending on the length of the iterable).

itertools.**dropwhile**(*predicate*, *iterable*)

Make an iterator that drops elements from the *iterable* while the *predicate* is true and afterwards returns every element. Roughly equivalent to:

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,3,8]) □ 6 3 8

    iterator = iter(iterable)
    for x in iterator:
        if not predicate(x):
            yield x
            break

    for x in iterator:
        yield x
```

Note this does not produce *any* output until the predicate first becomes false, so this itertool may have a lengthy start-up time.

itertools.**filterfalse**(*predicate*, *iterable*)

Make an iterator that filters elements from the *iterable* returning only those for which the *predicate* returns a false value. If *predicate* is None, returns the items that are false. Roughly equivalent to:

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x<5, [1,4,6,3,8]) □ 6 8

    if predicate is None:
        predicate = bool

    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.**groupby**(*iterable*, *key=None*)

*iterable*에서 연속적인 키와 그룹을 반환하는 이터레이터를 만듭니다. *key*는 각 요소의 키값을 계산하는 함수입니다. 지정되지 않거나 None이면, *key*의 기본값은 항등함수(identity function)이고 요소를 변경하지 않고 반환합니다. 일반적으로, iterable은 같은 키 함수로 이미 정렬되어 있어야 합니다.

*groupby()*의 작동은 유닉스의 uniq 필터와 유사합니다. 키 함수의 값이 변경될 때마다 중단(break)이나 새 그룹을 생성합니다 (이것이 일반적으로 같은 키 함수를 사용하여 데이터를 정렬해야 하는 이유입니다). 이 동작은 입력 순서와 관계없이 공통 요소를 집계하는 SQL의 GROUP BY와 다릅니다.

반환되는 그룹 자체는 *groupby()*와 하부 이터러블(iterable)을 공유하는 이터레이터입니다. 소스가 공유되므로, *groupby()* 객체가 진행하면, 이전 그룹은 이 더는 보이지 않게 됩니다. 따라서, 나중에 데이터가 필요하면, 리스트로 저장해야 합니다:

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

*groupby()*는 대략 다음과 동등합니다:

```
def groupby(iterable, key=None):
    # [k for k, g in groupby('AAAABBBCCDAABBB')] □ A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] □ AAAA BBB CC D

    keyfunc = (lambda x: x) if key is None else key
    iterator = iter(iterable)
    exhausted = False

    def _grouper(target_key):
        nonlocal curr_value, curr_key, exhausted
        yield curr_value
        for curr_value in iterator:
            curr_key = keyfunc(curr_value)
            if curr_key != target_key:
                return
            yield curr_value
        exhausted = True

    try:
        curr_value = next(iterator)
    except StopIteration:
        return
    curr_key = keyfunc(curr_value)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    while not exhausted:
        target_key = curr_key
        curr_group = _grouper(target_key)
        yield curr_key, curr_group
        if curr_key == target_key:
            for _ in curr_group:
                pass
```

itertools.**islice**(*iterable*, *stop*)

itertools.**islice**(*iterable*, *start*, *stop*[, *step*])

Make an iterator that returns selected elements from the iterable. Works like sequence slicing but does not support negative values for *start*, *stop*, or *step*.

If *start* is zero or `None`, iteration starts at zero. Otherwise, elements from the iterable are skipped until *start* is reached.

If *stop* is `None`, iteration continues until the input is exhausted, if at all. Otherwise, it stops at the specified position.

If *step* is `None`, the step defaults to one. Elements are returned consecutively unless *step* is set higher than one which results in items being skipped.

대략 다음과 동등합니다:

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) □ A B
    # islice('ABCDEFG', 2, 4) □ C D
    # islice('ABCDEFG', 2, None) □ C D E F G
    # islice('ABCDEFG', 0, None, 2) □ A C E G

    s = slice(*args)
    start = 0 if s.start is None else s.start
    stop = s.stop
    step = 1 if s.step is None else s.step
    if start < 0 or (stop is not None and stop < 0) or step <= 0:
        raise ValueError

    indices = count() if stop is None else range(max(start, stop))
    next_i = start
    for i, element in zip(indices, iterable):
        if i == next_i:
            yield element
            next_i += step
```

If the input is an iterator, then fully consuming the *islice* advances the input iterator by `max(start, stop)` steps regardless of the *step* value.

itertools.**pairwise**(*iterable*)

Return successive overlapping pairs taken from the input *iterable*.

The number of 2-tuples in the output iterator will be one fewer than the number of inputs. It will be empty if the input iterable has fewer than two values.

대략 다음과 동등합니다:

```
def pairwise(iterable):
    # pairwise('ABCDEFG') □ AB BC CD DE EF FG
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    iterator = iter(iterable)
    a = next(iterator, None)

    for b in iterator:
        yield a, b
        a = b
```

Added in version 3.10.

itertools.**permutations**(*iterable*, *r=None*)

Return successive *r* length permutations of elements from the *iterable*.

*r*이 지정되지 않았거나 `None`이면, *r*의 기본값은 *iterable*의 길이이며 가능한 모든 최대 길이 순열이 생성됩니다.

The output is a subsequence of `product()` where entries with repeated elements have been filtered out. The length of the output is given by `math.perm()` which computes `n! / (n - r)!` when `0 ≤ r ≤ n` or zero when `r > n`.

The permutation tuples are emitted in lexicographic order according to the order of the input *iterable*. If the input *iterable* is sorted, the output tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. If the input elements are unique, there will be no repeated values within a permutation.

대략 다음과 동등합니다:

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) ▷ AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) ▷ 012 021 102 120 201 210

    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return

    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
    yield tuple(pool[i] for i in indices[:r])

    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

itertools.**product**(**iterables*, *repeat=1*)

Cartesian product of the input iterables.

대략 제너레이터 표현식에서의 중첩된 for-루프와 동등합니다. 예를 들어, `product(A, B)`는 `((x,y) for x in A for y in B)`와 같은 것을 반환합니다.

중첩된 루프는 매 이터레이션마다 가장 오른쪽 요소가 진행되는 주행 거리계처럼 순환합니다. 이 패턴은 사전식 순서를 만들어서 입력의 이터러블들이 정렬되어 있다면, 곱(product) 튜플이 정렬된 순서로 방출됩니다.

이터러블의 자신과의 곱을 계산하려면, 선택적 *repeat* 키워드 인자를 사용하여 반복 횟수를 지정하십시오. 예를 들어, `product(A, repeat=4)` 는 `product(A, A, A, A)` 와 같은 것을 뜻합니다.

이 함수는 실제 구현이 메모리에 중간 결과를 쌓지 않는다는 점을 제외하고 다음 코드와 대략 동등합니다:

```
def product(*iterables, repeat=1):
    # product('ABCD', 'xy') □ Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) □ 000 001 010 011 100 101 110 111

    if repeat < 0:
        raise ValueError('repeat argument cannot be negative')
    pools = [tuple(pool) for pool in iterables] * repeat

    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]

    for prod in result:
        yield tuple(prod)
```

product() 가 실행되기 전에, 입력 이터러블을 완전히 소비하여, 곱을 생성하기 위해 값의 풀(pool)을 메모리에 유지합니다. 따라서, 유한 입력에만 유용합니다.

`itertools.`**`repeat`**(*object*[, *times*])

Make an iterator that returns *object* over and over again. Runs indefinitely unless the *times* argument is specified.

대략 다음과 동등합니다:

```
def repeat(object, times=None):
    # repeat(10, 3) □ 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

A common use for *repeat* is to supply a stream of constant values to *map* or *zip*:

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`itertools.`**`starmap`**(*function*, *iterable*)

Make an iterator that computes the *function* using arguments obtained from the *iterable*. Used instead of *map()* when argument parameters have already been "pre-zipped" into tuples.

The difference between *map()* and *starmap()* parallels the distinction between `function(a,b)` and `function(*c)`. Roughly equivalent to:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) □ 32 9 1000
    for args in iterable:
        yield function(*args)
```

`itertools.`**`takewhile`**(*predicate*, *iterable*)

Make an iterator that returns elements from the *iterable* as long as the *predicate* is true. Roughly equivalent to:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,3,8]) → 1 4
    for x in iterable:
        if not predicate(x):
            break
        yield x
```

Note, the element that first fails the predicate condition is consumed from the input iterator and there is no way to access it. This could be an issue if an application wants to further consume the input iterator after *takewhile* has been run to exhaustion. To work around this problem, consider using more-itertools before_and_after() instead.

itertools.**tee**(*iterable*, *n=2*)

단일 iterable에서 *n* 개의 독립 이터레이터를 반환합니다.

대략 다음과 동등합니다:

```
def tee(iterable, n=2):
    if n < 0:
        raise ValueError
    if n == 0:
        return ()
    iterator = _tee(iterable)
    result = [iterator]
    for _ in range(n - 1):
        result.append(_tee(iterator))
    return tuple(result)

class _tee:

    def __init__(self, iterable):
        it = iter(iterable)
        if isinstance(it, _tee):
            self.iterator = it.iterator
            self.link = it.link
        else:
            self.iterator = it
            self.link = [None, None]

    def __iter__(self):
        return self

    def __next__(self):
        link = self.link
        if link[1] is None:
            link[0] = next(self.iterator)
            link[1] = [None, None]
        value, self.link = link
        return value
```

When the input *iterable* is already a tee iterator object, all members of the return tuple are constructed as if they had been produced by the upstream `tee()` call. This "flattening step" allows nested `tee()` calls to share the same underlying data chain and to have a single update step rather than a chain of calls.

The flattening property makes tee iterators efficiently peekable:

```
def lookahead(tee_iterator):
     "Return the next value without moving the input forward"
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    [forked_iterator] = tee(tee_iterator, 1)
    return next(forked_iterator)
```

```
>>> iterator = iter('abcdef')
>>> [iterator] = tee(iterator, 1)    # Make the input peekable
>>> next(iterator)                   # Move the iterator forward
'a'
>>> lookahead(iterator)              # Check next value
'b'
>>> next(iterator)                   # Continue moving forward
'b'
```

`tee` iterators are not threadsafe. A `RuntimeError` may be raised when simultaneously using iterators returned by the same `tee()` call, even if the original *iterable* is threadsafe.

이 이터레이터 도구에는 상당한 보조 기억 장치가 필요할 수 있습니다 (일시적으로 저장해야 하는 데이터양에 따라 다릅니다). 일반적으로, 다른 이터레이터가 시작하기 전에 하나의 이터레이터가 대부분이나 모든 데이터를 사용하면, `tee()` 대신 `list()`를 사용하는 것이 더 빠릅니다.

`itertools.`**`zip_longest`**(**iterables*, *fillvalue=None*)

Make an iterator that aggregates elements from each of the *iterables*.

If the iterables are of uneven length, missing values are filled-in with *fillvalue*. If not specified, *fillvalue* defaults to `None`.

Iteration continues until the longest iterable is exhausted.

대략 다음과 동등합니다:

```
def zip_longest(*iterables, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='-') □ Ax By C- D-

    iterators = list(map(iter, iterables))
    num_active = len(iterators)
    if not num_active:
        return

    while True:
        values = []
        for i, iterator in enumerate(iterators):
            try:
                value = next(iterator)
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
                iterators[i] = repeat(fillvalue)
                value = fillvalue
            values.append(value)
        yield tuple(values)
```

If one of the iterables is potentially infinite, then the `zip_longest()` function should be wrapped with something that limits the number of calls (for example `islice()` or `takewhile()`).

10.1.2 Itertools 조리법

이 섹션에서는 기존 itertools를 빌딩 블록으로 사용하여 확장 도구 집합을 만드는 방법을 보여줍니다.

The primary purpose of the itertools recipes is educational. The recipes show various ways of thinking about individual tools — for example, that `chain.from_iterable` is related to the concept of flattening. The recipes also give ideas about ways that the tools can be combined — for example, how `starmap()` and `repeat()` can work together. The recipes also show patterns for using itertools with the *operator* and *collections* modules as well as with the built-in itertools such as `map()`, `filter()`, `reversed()`, and `enumerate()`.

A secondary purpose of the recipes is to serve as an incubator. The `accumulate()`, `compress()`, and `pairwise()` itertools started out as recipes. Currently, the `sliding_window()`, `iter_index()`, and `sieve()` recipes are being tested to see whether they prove their worth.

Substantially all of these recipes and many, many others can be installed from the more-itertools project found on the Python Package Index:

```
python -m pip install more-itertools
```

Many of the recipes offer the same high performance as the underlying toolset. Superior memory performance is kept by processing elements one at a time rather than bringing the whole iterable into memory all at once. Code volume is kept small by linking the tools together in a functional style. High speed is retained by preferring "vectorized" building blocks over the use of for-loops and *generators* which incur interpreter overhead.

```
from collections import Counter, deque
from contextlib import suppress
from functools import reduce
from math import comb, prod, sumprod, isqrt
from operator import itemgetter, getitem, mul, neg

def take(n, iterable):
    "Return first n items of the iterable as a list."
    return list(islice(iterable, n))

def prepend(value, iterable):
    "Prepend a single value in front of an iterable."
    # prepend(1, [2, 3, 4]) □ 1 2 3 4
    return chain([value], iterable)

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def repeatfunc(function, times=None, *args):
    "Repeat calls to a function with specified arguments."
    if times is None:
        return starmap(function, repeat(args))
    return starmap(function, repeat(args, times))

def flatten(list_of_lists):
    "Flatten one level of nesting."
    return chain.from_iterable(list_of_lists)

def ncycles(iterable, n):
    "Returns the sequence elements n times."
    return chain.from_iterable(repeat(tuple(iterable), n))

def loops(n):
    "Loop n times. Like range(n) but without creating integers."
    # for _ in loops(100): ...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    return repeat(None, n)

def tail(n, iterable):
    "Return an iterator over the last n items."
    # tail(3, 'ABCDEFG') □ E F G
    return iter(deque(iterable, maxlen=n))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # Use functions that consume iterators at C speed.
    if n is None:
        deque(iterator, maxlen=0)
    else:
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value."
    return next(islice(iterable, n, None), default)

def quantify(iterable, predicate=bool):
    "Given a predicate that returns True or False, count the True results."
    return sum(map(predicate, iterable))

def first_true(iterable, default=False, predicate=None):
    "Returns the first true value or the *default* if there is no true value."
    # first_true([a,b,c], x) □ a or b or c or x
    # first_true([a,b], x, f) □ a if f(a) else b if f(b) else x
    return next(filter(predicate, iterable), default)

def all_equal(iterable, key=None):
    "Returns True if all the elements are equal to each other."
    # all_equal('4□□□□', key=int) □ True
    return len(take(2, groupby(iterable, key))) <= 1

def unique_justseen(iterable, key=None):
    "Yield unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') □ A B C D A B
    # unique_justseen('ABBcCAD', str.casefold) □ A B c A D
    if key is None:
        return map(itemgetter(0), groupby(iterable))
    return map(next, map(itemgetter(1), groupby(iterable, key)))

def unique_everseen(iterable, key=None):
    "Yield unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') □ A B C D
    # unique_everseen('ABBcCAD', str.casefold) □ A B c D
    seen = set()
    if key is None:
        for element in filterfalse(seen.__contains__, iterable):
            seen.add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen.add(k)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                yield element

def unique(iterable, key=None, reverse=False):
   "Yield unique elements in sorted order. Supports unhashable inputs."
   # unique([[1, 2], [3, 4], [1, 2]]) □ [1, 2] [3, 4]
   sequenced = sorted(iterable, key=key, reverse=reverse)
   return unique_justseen(sequenced, key=key)

def sliding_window(iterable, n):
    "Collect data into overlapping fixed-length chunks or blocks."
    # sliding_window('ABCDEFG', 4) □ ABCD BCDE CDEF DEFG
    iterator = iter(iterable)
    window = deque(islice(iterator, n - 1), maxlen=n)
    for x in iterator:
        window.append(x)
        yield tuple(window)

def grouper(iterable, n, *, incomplete='fill', fillvalue=None):
    "Collect data into non-overlapping fixed-length chunks or blocks."
    # grouper('ABCDEFG', 3, fillvalue='x') □ ABC DEF Gxx
    # grouper('ABCDEFG', 3, incomplete='strict') □ ABC DEF ValueError
    # grouper('ABCDEFG', 3, incomplete='ignore') □ ABC DEF
    iterators = [iter(iterable)] * n
    match incomplete:
        case 'fill':
            return zip_longest(*iterators, fillvalue=fillvalue)
        case 'strict':
            return zip(*iterators, strict=True)
        case 'ignore':
            return zip(*iterators)
        case _:
            raise ValueError('Expected fill, strict, or ignore')

def roundrobin(*iterables):
    "Visit input iterables in a cycle until each is exhausted."
    # roundrobin('ABC', 'D', 'EF') □ A D E B F C
    # Algorithm credited to George Sakkis
    iterators = map(iter, iterables)
    for num_active in range(len(iterables), 0, -1):
        iterators = cycle(islice(iterators, num_active))
        yield from map(next, iterators)

def subslices(seq):
    "Return all contiguous non-empty subslices of a sequence."
    # subslices('ABCD') □ A AB ABC ABCD B BC BCD C CD D
    slices = starmap(slice, combinations(range(len(seq) + 1), 2))
    return map(getitem, repeat(seq), slices)

def iter_index(iterable, value, start=0, stop=None):
    "Return indices where a value occurs in a sequence or iterable."
    # iter_index('AABCADEAF', 'A') □ 0 1 4 7
    seq_index = getattr(iterable, 'index', None)
    if seq_index is None:
        iterator = islice(iterable, start, stop)
        for i, element in enumerate(iterator, start):
            if element is value or element == value:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                yield i
    else:
        stop = len(iterable) if stop is None else stop
        i = start
        with suppress(ValueError):
            while True:
                yield (i := seq_index(value, i, stop))
                i += 1

def iter_except(function, exception, first=None):
    "Convert a call-until-exception interface to an iterator interface."
    # iter_except(d.popitem, KeyError) □ non-blocking dictionary iterator
    with suppress(exception):
        if first is not None:
            yield first()
        while True:
            yield function()
```

The following recipes have a more mathematical flavor:

```
def multinomial(*counts):
    "Number of distinct arrangements of a multiset."
    # Counter('abracadabra').values() □ 5 2 2 1 1
    # multinomial(5, 2, 2, 1, 1) □ 83160
    return prod(map(comb, accumulate(counts), counts))

def powerset(iterable):
    "Subsequences of the iterable from shortest to longest."
    # powerset([1,2,3]) □ () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def sum_of_squares(iterable):
    "Add up the squares of the input values."
    # sum_of_squares([10, 20, 30]) □ 1400
    return sumprod(*tee(iterable))

def reshape(matrix, columns):
    "Reshape a 2-D matrix to have a given number of columns."
    # reshape([(0, 1), (2, 3), (4, 5)], 3) □  (0, 1, 2), (3, 4, 5)
    return batched(chain.from_iterable(matrix), columns, strict=True)

def transpose(matrix):
    "Swap the rows and columns of a 2-D matrix."
    # transpose([(1, 2, 3), (11, 22, 33)]) □ (1, 11) (2, 22) (3, 33)
    return zip(*matrix, strict=True)

def matmul(m1, m2):
    "Multiply two matrices."
    # matmul([(7, 5), (3, 5)], [(2, 5), (7, 9)]) □ (49, 80), (41, 60)
    n = len(m2[0])
    return batched(starmap(sumprod, product(m1, transpose(m2))), n)

def convolve(signal, kernel):
    """Discrete linear convolution of two iterables.
    Equivalent to polynomial multiplication.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    Convolutions are mathematically commutative; however, the inputs are
    evaluated differently.  The signal is consumed lazily and can be
    infinite. The kernel is fully consumed before the calculations begin.

    Article:  https://betterexplained.com/articles/intuitive-convolution/
    Video:    https://www.youtube.com/watch?v=KuXjwB4LzSA
    """
    # convolve([1, -1, -20], [1, -3]) → 1 -4 -17 60
    # convolve(data, [0.25, 0.25, 0.25, 0.25]) → Moving average (blur)
    # convolve(data, [1/2, 0, -1/2]) → 1st derivative estimate
    # convolve(data, [1, -2, 1]) → 2nd derivative estimate
    kernel = tuple(kernel)[::-1]
    n = len(kernel)
    padded_signal = chain(repeat(0, n-1), signal, repeat(0, n-1))
    windowed_signal = sliding_window(padded_signal, n)
    return map(sumprod, repeat(kernel), windowed_signal)

def polynomial_from_roots(roots):
    """Compute a polynomial's coefficients from its roots.

       (x - 5) (x + 4) (x - 3)  expands to:   x³ -4x² -17x + 60
    """
    # polynomial_from_roots([5, -4, 3]) → [1, -4, -17, 60]
    factors = zip(repeat(1), map(neg, roots))
    return list(reduce(convolve, factors, [1]))

def polynomial_eval(coefficients, x):
    """Evaluate a polynomial at a specific value.

    Computes with better numeric stability than Horner's method.
    """
    # Evaluate x³ -4x² -17x + 60 at x = 5
    # polynomial_eval([1, -4, -17, 60], x=5) → 0
    n = len(coefficients)
    if not n:
        return type(x)(0)
    powers = map(pow, repeat(x), reversed(range(n)))
    return sumprod(coefficients, powers)

def polynomial_derivative(coefficients):
    """Compute the first derivative of a polynomial.

       f(x)  =  x³ -4x² -17x + 60
       f'(x) = 3x² -8x  -17
    """
    # polynomial_derivative([1, -4, -17, 60]) → [3, -8, -17]
    n = len(coefficients)
    powers = reversed(range(1, n))
    return list(map(mul, coefficients, powers))

def sieve(n):
    "Primes less than n."
    # sieve(30) → 2 3 5 7 11 13 17 19 23 29
    if n > 2:
        yield 2
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    data = bytearray((0, 1)) * (n // 2)
    for p in iter_index(data, 1, start=3, stop=isqrt(n) + 1):
        data[p*p : n : p+p] = bytes(len(range(p*p, n, p+p)))
    yield from iter_index(data, 1, start=3)

def factor(n):
    "Prime factors of n."
    # factor(99) → 3 3 11
    # factor(1_000_000_000_000_007) → 47 59 360620266859
    # factor(1_000_000_000_000_403) → 1000000000000403
    for prime in sieve(isqrt(n) + 1):
        while not n % prime:
            yield prime
            n //= prime
            if n == 1:
                return
    if n > 1:
        yield n

def is_prime(n):
    "Return True if n is prime."
    # is_prime(1_000_000_000_000_403) → True
    return n > 1 and next(factor(n)) == n

def totient(n):
    "Count of natural numbers up to n that are coprime to n."
    # https://mathworld.wolfram.com/TotientFunction.html
    # totient(12) → 4 because len([1, 5, 7, 11]) == 4
    for prime in set(factor(n)):
        n -= n // prime
    return n
```

10.2 functools — 고차 함수와 콜러블 객체에 대한 연산

소스 코드: Lib/functools.py

---

functools 모듈은 고차 함수를 위한 것입니다: 다른 함수에 작용하거나 다른 함수를 반환하는 함수. 일반적으로, 모든 콜러블 객체는 이 모듈의 목적상 함수로 취급될 수 있습니다.

functools 모듈은 다음 함수를 정의합니다:

@functools.**cache**(*user_function*)

단순하고 가벼운 무제한 함수 캐시. 때때로 "memoize"라고도 합니다.

`lru_cache(maxsize=None)` 와 같은 것을 반환하여, 함수 인자의 딕셔너리 조회 주위로 얇은 래퍼를 만듭니다. 이전 값을 제거할 필요가 없어서, 크기 제한이 있는 *lru_cache()* 보다 작고 빠릅니다.

예를 들면:

```
@cache
def factorial(n):
    return n * factorial(n-1) if n else 1

>>> factorial(10)       # 이전에 캐시 된 결과가 없습니다, 11번의 재귀 호출을 만듭니다
3628800
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> factorial(5)        # 캐시 된 값 결과를 조회만 합니다
120
>>> factorial(12)       # 두 개의 새로운 재귀 호출을 만듭니다, 다른 10개는 캐시 된
↪것입니다
479001600
```

The cache is threadsafe so that the wrapped function can be used in multiple threads. This means that the underlying data structure will remain coherent during concurrent updates.

It is possible for the wrapped function to be called more than once if another thread makes an additional call before the initial call has been completed and cached.

Added in version 3.9.

`@functools.cached_property`(*func*)

클래스의 메서드를 값이 한 번 계산된 다음 인스턴스 수명 동안 일반 어트리뷰트로 캐시 되는 프로퍼티로 변환합니다. `property()`와 유사하고, 캐싱이 추가되었습니다. 비싸게 계산되고 그 외에는 사실상 불변인 인스턴스의 프로퍼티에 유용합니다.

예:

```
class DataSet:

    def __init__(self, sequence_of_numbers):
        self._data = tuple(sequence_of_numbers)

    @cached_property
    def stdev(self):
        return statistics.stdev(self._data)
```

`cached_property()`의 메커니즘은 `property()`와 다소 다릅니다. 일반 프로퍼티는 setter가 정의되지 않은 경우 어트리뷰트 쓰기를 차단합니다. 이와는 달리, *cached_property*는 쓰기를 허용합니다.

cached_property 데코레이터는 조회 시에만, 같은 이름의 어트리뷰트가 존재하지 않을 때만 실행됩니다. 실행되면, *cached_property*는 같은 이름의 어트리뷰트에 기록합니다. 후속 어트리뷰트 읽기와 쓰기는 *cached_property* 메서드보다 우선하며 일반 어트리뷰트처럼 작동합니다.

캐시 된 값은 어트리뷰트를 삭제하여 지울 수 있습니다. 이렇게 하면 *cached_property* 메서드가 다시 실행됩니다.

The *cached_property* does not prevent a possible race condition in multi-threaded usage. The getter function could run more than once on the same instance, with the latest run setting the cached value. If the cached property is idempotent or otherwise not harmful to run more than once on an instance, this is fine. If synchronization is needed, implement the necessary locking inside the decorated getter function or around the cached property access.

이 데코레이터는 **PEP 412** 키 공유 딕셔너리의 작동을 방해함에 유의하십시오. 이는 인스턴스 딕셔너리가 평소보다 더 많은 공간을 차지할 수 있음을 의미합니다.

또한, 이 데코레이터는 각 인스턴스의 `__dict__` 어트리뷰트가 가변 매핑일 것을 요구합니다. 이는 메타 클래스(형 인스턴스의 `__dict__` 어트리뷰트가 클래스 이름 공간에 대한 읽기 전용 프락시이기 때문에)와 `__dict__`를 정의된 슬롯 중 하나로 포함하지 않고 `__slots__`를 지정하는 것(이러한 클래스는 `__dict__` 어트리뷰트를 전혀 제공하지 않기 때문에)과 같은 일부 형에서 작동하지 않음을 의미합니다.

가변 매핑을 사용할 수 없거나 공간 효율적인 키 공유가 필요하면, `lru_cache()` 위에 `property()`를 쌓아서 `cached_property()`와 유사한 효과를 얻을 수도 있습니다. 이것이 `cached_property()`와 어떻게 다른지에 관한 자세한 내용은 faq-cache-method-calls을 참조하세요.

Added in version 3.8.

버전 3.12에서 변경: Prior to Python 3.12, `cached_property` included an undocumented lock to ensure that in multi-threaded usage the getter function was guaranteed to run only once per instance. However, the

lock was per-property, not per-instance, which could result in unacceptably high lock contention. In Python 3.12+ this locking is removed.

functools.**cmp_to_key**(*func*)

구식 비교 함수를 키 함수로 변환합니다. (*sorted()*, *min()*, *max()*, *heapq.nlargest()*, *heapq.nsmallest()*, *itertools.groupby()*와 같은) 키 함수를 받아들이는 도구와 함께 사용됩니다. 이 함수는 주로 비교 함수 사용을 지원하는 파이썬 2에서 변환되는 프로그램의 전이 도구로 사용됩니다.

비교 함수는 두 개의 인자를 받아들이고, 그들을 비교하여, 작으면 음수, 같으면 0, 크면 양수를 반환하는 콜러블입니다. 키 함수는 하나의 인자를 받아들이고 정렬 키로 사용할 다른 값을 반환하는 콜러블입니다.

예:

```
sorted(iterable, key=cmp_to_key(locale.strcoll))  # 로케일을 고려한 정렬 순서
```

정렬 예제와 간략한 정렬 자습서는 sortinghowto를 참조하십시오.

Added in version 3.2.

@functools.**lru_cache**(*user_function*)

@functools.**lru_cache**(*maxsize=128, typed=False*)

가장 최근의 *maxsize* 호출까지 저장하는 기억하는(memoizing) 콜러블 함수를 감싸는 데코레이터. 비싸거나 I/O 병목 함수가 같은 인자로 주기적으로 호출될 때 시간을 절약할 수 있습니다.

The cache is threadsafe so that the wrapped function can be used in multiple threads. This means that the underlying data structure will remain coherent during concurrent updates.

It is possible for the wrapped function to be called more than once if another thread makes an additional call before the initial call has been completed and cached.

결과를 캐시 하는 데 딕셔너리가 사용되기 때문에, 함수에 대한 위치와 키워드 인자는 해시 가능해야 합니다.

서도 다른 인자 패턴은 별도의 캐시 항목을 갖는 별개의 호출로 간주할 수 있습니다. 예를 들어, `f(a=1, b=2)`와 `f(b=2, a=1)`은 키워드 인자 순서가 다르며 두 개의 개별 캐시 항목을 가질 수 있습니다.

*user_function*이 지정되면, 콜러블이어야 합니다. 이는 *lru_cache* 데코레이터를 사용자 함수에 직접 적용 할 수 있도록 하며, *maxsize*를 기본값 128로 유지합니다:

```
@lru_cache
def count_vowels(sentence):
    return sum(sentence.count(vowel) for vowel in 'AEIOUaeiou')
```

*maxsize*가 `None`으로 설정되면, LRU 기능이 비활성화되고 캐시가 제한 없이 커질 수 있습니다.

If *typed* is set to true, function arguments of different types will be cached separately. If *typed* is false, the implementation will usually regard them as equivalent calls and only cache a single result. (Some types such as *str* and *int* may be cached separately even when *typed* is false.)

Note, type specificity applies only to the function' s immediate arguments rather than their contents. The scalar arguments, `Decimal(42)` and `Fraction(42)` are be treated as distinct calls with distinct results. In contrast, the tuple arguments `('answer', Decimal(42))` and `('answer', Fraction(42))` are treated as equivalent.

래핑 된 함수는 *maxsize*와 *typed*의 값을 표시하는 새 *dict*를 반환하는 `cache_parameters()` 함수로 인스트루먼트 됩니다. 이것은 정보 제공만을 위한 것입니다. 값을 변경해도 효과가 없습니다. To help measure the effectiveness of the cache and tune the *maxsize* parameter, the wrapped function is instrumented with a `cache_info()` function that returns a *named tuple* showing *hits*, *misses*, *maxsize* and *currsize*. The decorator also provides a `cache_clear()` function for clearing or invalidating the cache.

원래의 하부 함수는 `__wrapped__` 어트리뷰트를 통해 액세스 할 수 있습니다. 이것은 인트로스펙션, 캐시 우회 또는 다른 캐시로 함수를 다시 래핑하는 데 유용합니다.

The cache keeps references to the arguments and return values until they age out of the cache or until the cache is cleared.

If a method is cached, the `self` instance argument is included in the cache. See faq-cache-method-calls

LRU (least recently used) 캐시는 가장 최근 호출이 향후 호출에 대한 최상의 예측일 때 가장 잘 작동합니다 (예를 들어, 뉴스 서버에서 가장 인기 있는 기사는 매일 바뀌는 경향이 있습니다). 캐시의 크기 제한은 웹 서버와 같은 오래 실행되는 프로세스에서 제한 없이 캐시가 커지지 않도록 합니다.

일반적으로, LRU 캐시는 이전에 계산된 값을 재사용하려고 할 때만 사용해야 합니다. 따라서, 부작용이 있는 함수, 각 호출에서 고유한 가변 객체를 만들어야 하는 함수 (가령 제너레이터와 비동기 함수), time() 이나 random() 과 같은 비순수(impure) 함수를 캐시 하는 것은 의미가 없습니다.

정적 웹 콘텐츠를 위한 LRU 캐시의 예:

```
@lru_cache(maxsize=32)
def get_pep(num):
    'PEP(Python Enhancement Proposal) 텍스트를 가져옵니다'
    resource = f'https://peps.python.org/pep-{num:04d}'
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)
...     print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

동적 프로그래밍(dynamic programming) 기법을 구현하기 위해 캐시를 사용하여 피보나치 수를 효율적으로 계산하는 예:

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

Added in version 3.2.

버전 3.3에서 변경: *typed* 옵션을 추가했습니다.

버전 3.8에서 변경: *user_function* 옵션을 추가했습니다.

버전 3.9에서 변경: `cache_parameters()` 함수를 추가했습니다

`@functools.total_ordering`

하나 이상의 풍부한 비교(rich comparison) 순서 메서드를 정의하는 클래스를 주면, 이 클래스 데코레이터가 나머지를 제공합니다. 가능한 모든 풍부한 비교 연산을 지정하는 데 드는 노력이 단순화됩니다:

The class must define one of `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`. In addition, the class should supply an `__eq__()` method.

예를 들면:

```
@total_ordering
class Student:
    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

> **참고**
>
> 이 데코레이터를 사용하면 올바르게 동작하는 전 순서(totally ordered) 형을 쉽게 만들 수 있지만, 파생된 비교 메서드에서 실행 속도가 느려지고 스택 트레이스가 더 복잡해지는 대가를 지불합니다. 성능 벤치마킹이 이것이 특정 응용 프로그램의 병목임을 가리키면, 6가지의 풍부한 비교 메서드를 모두 구현하여 속도를 쉽게 높일 수 있습니다.

> **참고**
>
> This decorator makes no attempt to override methods that have been declared in the class *or its superclasses*. Meaning that if a superclass defines a comparison operator, *total_ordering* will not implement it again, even if the original method is abstract.

Added in version 3.2.

버전 3.4에서 변경: 인식할 수 없는 형에 대해 하부 비교 함수에서 `NotImplemented`를 반환하는 것이 이제 지원됩니다.

`functools.`**`Placeholder`**

A singleton object used as a sentinel to reserve a place for positional arguments when calling *partial()* and *partialmethod()*.

Added in version 3.14.

`functools.`**`partial`**(*func*, /, **args*, ***keywords*)

호출될 때 위치 인자 *args*와 키워드 인자 *keywords*로 호출된 *func*처럼 동작하는 새 *partial 객체*를 반환합니다. 더 많은 인자가 호출에 제공되면, *args*에 추가됩니다. 추가 키워드 인자가 제공되면, *keywords*를 확장하고 대체합니다. 대략 다음과 동등합니다:

```
def partial(func, /, *args, **keywords):
    def newfunc(*more_args, **more_keywords):
        return func(*args, *more_args, **(keywords | more_keywords))
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

The `partial()` function is used for partial function application which "freezes" some portion of a function's arguments and/or keywords resulting in a new object with a simplified signature. For example, *partial()*

can be used to create a callable that behaves like the `int()` function where the *base* argument defaults to `2`:

```
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

If `Placeholder` sentinels are present in *args*, they will be filled first when `partial()` is called. This makes it possible to pre-fill any positional argument with a call to `partial()`; without `Placeholder`, only the chosen number of leading positional arguments can be pre-filled.

If any `Placeholder` sentinels are present, all must be filled at call time:

```
>>> say_to_world = partial(print, Placeholder, Placeholder, "world!")
>>> say_to_world('Hello', 'dear')
Hello dear world!
```

Calling `say_to_world('Hello')` raises a `TypeError`, because only one positional argument is provided, but there are two placeholders that must be filled in.

If `partial()` is applied to an existing `partial()` object, `Placeholder` sentinels of the input object are filled in with new positional arguments. A placeholder can be retained by inserting a new `Placeholder` sentinel to the place held by a previous `Placeholder`:

```
>>> from functools import partial, Placeholder as _
>>> remove = partial(str.replace, _, _, '')
>>> message = 'Hello, dear dear world!'
>>> remove(message, ' dear')
'Hello, world!'
>>> remove_dear = partial(remove, _, ' dear')
>>> remove_dear(message)
'Hello, world!'
>>> remove_first_dear = partial(remove_dear, _, 1)
>>> remove_first_dear(message)
'Hello, dear world!'
```

`Placeholder` cannot be passed to `partial()` as a keyword argument.

버전 3.14에서 변경: Added support for `Placeholder` in positional arguments.

class functools.**partialmethod**(*func, /, *args, **keywords*)

직접 호출하기보다는 메서드 정의로 사용되도록 설계된 것을 제외하고는 `partial`과 같이 동작하는 새 `partialmethod` 디스크립터를 반환합니다.

*func*는 디스크립터나 콜러블이어야 합니다 (일반 함수처럼 둘 모두인 객체는 디스크립터로 처리됩니다).

When *func* is a descriptor (such as a normal Python function, `classmethod()`, `staticmethod()`, `abstractmethod()` or another instance of `partialmethod`), calls to `__get__` are delegated to the underlying descriptor, and an appropriate *partial object* returned as the result.

*func*가 디스크립터가 아닌 콜러블이면, 적절한 연결된 메서드가 동적으로 만들어집니다. 이것은 메서드로 사용될 때 일반 파이썬 함수처럼 작동합니다: `partialmethod` 생성자에 제공된 *args*와 *keywords*보다도 전에 *self* 인자가 첫 번째 위치 인자로 삽입됩니다.

예:

```
>>> class Cell:
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...         return self._alive
...     def set_state(self, state):
...         self._alive = bool(state)
...     set_alive = partialmethod(set_state, True)
...     set_dead = partialmethod(set_state, False)
...
>>> c = Cell()
>>> c.alive
False
>>> c.set_alive()
>>> c.alive
True
```

Added in version 3.4.

functools.**reduce**(*function*, *iterable*, /[, *initial*])

두 인자의 *function*을 왼쪽에서 오른쪽으로 *iterable*의 항목에 누적적으로 적용해서, 이터러블을 단일 값으로 줄입니다. 예를 들어, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])`는 `((((1+2)+3)+4)+5)`를 계산합니다. 왼쪽 인자 *x*는 누적값이고 오른쪽 인자 *y*는 *iterable*에서 온 갱신 값입니다. 선택적 *initial*가 있으면, 계산에서 이터러블의 항목 앞에 배치되고, 이터러블이 비어있을 때 기본값의 역할을 합니다. *initial*가 제공되지 않고 *iterable*에 하나의 항목만 포함되면, 첫 번째 항목이 반환됩니다.

대략 다음과 동등합니다:

```
initial_missing = object()

def reduce(function, iterable, /, initial=initial_missing):
    it = iter(iterable)
    if initial is initial_missing:
        value = next(it)
    else:
        value = initial
    for element in it:
        value = function(value, element)
    return value
```

모든 중간값을 산출하는 이터레이터는 *itertools.accumulate()*를 참조하십시오.

버전 3.14에서 변경: *initial* is now supported as a keyword argument.

@functools.**singledispatch**

함수를 싱글 디스패치 제네릭 함수로 변환합니다.

제네릭 함수를 정의하려면, `@singledispatch` 데코레이터로 데코레이트 하십시오. `@singledispatch`를 사용해서 함수를 정의할 때, 디스패치는 첫 번째 인자의 형으로 일어남에 유의하십시오:

```
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)
```

To add overloaded implementations to the function, use the `register()` attribute of the generic function, which can be used as a decorator. For functions annotated with types, the decorator will infer the type of the first argument automatically:

```
>>> @fun.register
... def _(arg: int, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> @fun.register
... def _(arg: list, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

typing.Union can also be used:

```
>>> @fun.register
... def _(arg: int | float, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> from typing import Union
>>> @fun.register
... def _(arg: Union[list, set], verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
...
```

형 어노테이션을 사용하지 않는 코드의 경우, 적절한 형 인자를 데코레이터 자체에 명시적으로 전달할 수 있습니다:

```
>>> @fun.register(complex)
... def _(arg, verbose=False):
...     if verbose:
...         print("Better than complicated.", end=" ")
...     print(arg.real, arg.imag)
...
```

For code that dispatches on a collections type (e.g., `list`), but wants to typehint the items of the collection (e.g., `list[int]`), the dispatch type should be passed explicitly to the decorator itself with the typehint going into the function definition:

```
>>> @fun.register(list)
... def _(arg: list[int], verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

> **참고**
>
> At runtime the function will dispatch on an instance of a list regardless of the type contained within the list i.e. `[1,2,3]` will be dispatched the same as `["foo", "bar", "baz"]`. The annotation provided in this example is for static type checkers only and has no runtime impact.

To enable registering *lambdas* and pre-existing functions, the `register()` attribute can also be used in a functional form:

```
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)
```

The `register()` attribute returns the undecorated function. This enables decorator stacking, `pickling`, and the creation of unit tests for each variant independently:

```
>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...     print(arg / 2)
...
>>> fun_num is fun
False
```

호출되면, 제네릭 함수는 첫 번째 인자의 형에 따라 디스패치 합니다:

```
>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
>>> fun(['spam', 'spam', 'eggs', 'spam'], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
3 spam
>>> fun(None)
Nothing.
>>> fun(1.23)
0.615
```

특정 형에 대해 등록된 구현이 없으면, 더 일반적인 구현을 찾는 데 메서드 결정 순서가 사용됩니다. `@singledispatch`로 데코레이트 된 원래 함수는 베이스 `object` 형으로 등록되어서, 더 나은 구현이 발견되지 않으면 사용됩니다.

구현이 추상 베이스 클래스에 등록되면, 베이스 클래스의 가상 서브 클래스는 그 구현으로 디스패치 됩니다:

```
>>> from collections.abc import Mapping
>>> @fun.register
... def _(arg: Mapping, verbose=False):
...     if verbose:
...         print("Keys & Values")
...     for key, value in arg.items():
...         print(key, "=>", value)
...
>>> fun({"a": "b"})
a => b
```

제네릭 함수가 주어진 형에 대해 어떤 구현을 선택하는지 확인하려면 `dispatch()` 어트리뷰트를 사용하십시오:

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)    # 참고: 기본 구현
<function fun at 0x103fe0000>
```

등록된 모든 구현에 액세스하려면, 읽기 전용 `registry` 어트리뷰트를 사용하십시오:

```
>>> fun.registry.keys()
dict_keys([<class 'NoneType'>, <class 'int'>, <class 'object'>,
          <class 'decimal.Decimal'>, <class 'list'>,
          <class 'float'>])
>>> fun.registry[float]
<function fun_num at 0x1035a2840>
>>> fun.registry[object]
<function fun at 0x103fe0000>
```

Added in version 3.4.

버전 3.7에서 변경: The `register()` attribute now supports using type annotations.

버전 3.11에서 변경: The `register()` attribute now supports `typing.Union` as a type annotation.

class functools.**singledispatchmethod**(*func*)

메서드를 싱글 디스패치 제네릭 함수로 변환합니다.

제네릭 메서드를 정의하려면, `@singledispatchmethod` 데코레이터로 장식하십시오. `@singledispatchmethod`를 사용해서 함수를 정의할 때, 디스패치는 첫 번째 *self*나 *cls*가 아닌 인자의 형에 따라 일어남에 유의하십시오:

```
class Negator:
    @singledispatchmethod
    def neg(self, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    def _(self, arg: int):
        return -arg

    @neg.register
    def _(self, arg: bool):
        return not arg
```

`@singledispatchmethod`는 `@classmethod`와 같은 다른 데코레이터와의 중첩을 지원합니다. `dispatcher.register`를 허용하기 위해, `singledispatchmethod`는 가장 바깥 데코레이터이어야 함에 유의하십시오. 다음은 `neg` 메서드가 클래스의 인스턴스가 아니라 클래스에 연결되는 `Negator` 클래스입니다:

```
class Negator:
    @singledispatchmethod
    @classmethod
    def neg(cls, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    @classmethod
    def _(cls, arg: int):
        return -arg

    @neg.register
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    @classmethod
    def _(cls, arg: bool):
        return not arg
```

다른 유사한 데코레이터에도 같은 패턴을 사용할 수 있습니다: `@staticmethod`, `@abstractmethod` 및 기타.

Added in version 3.8.

functools.**update_wrapper**(*wrapper*, *wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

wrapped 함수처럼 보이도록 *wrapper* 함수를 갱신합니다. 선택적 인자는 원래 함수의 어떤 어트리뷰트가 wrapper 함수의 일치하는 어트리뷰트에 직접 대입되고 wrapper 함수의 어떤 어트리뷰트가 원래 함수의 해당 어트리뷰트로 갱신되는지 지정하는 튜플입니다. 이 인자들의 기본값은 모듈 수준 상수 `WRAPPER_ASSIGNMENTS`(wrapper 함수의 `__module__`, `__name__`, `__qualname__`, `__annotations__`, `__type_params__` 및 `__doc__` 독스트링에 대입합니다)와 `WRAPPER_UPDATES`(wrapper 함수의 `__dict__`, 즉 인스턴스 딕셔너리를 갱신합니다)입니다.

내부 검사와 기타 목적(예를 들어 `lru_cache()`와 같은 캐싱 데코레이터 우회)을 위해 원래 함수에 액세스 할 수 있도록, 이 함수는 래핑 되는 함수를 가리키는 `__wrapped__` 어트리뷰트를 wrapper에 자동으로 추가합니다.

이 함수의 주요 용도는 데코레이트 된 함수를 래핑하고 wrapper를 반환하는 데코레이터 함수에서 사용하는 것입니다. wrapper 함수가 갱신되지 않으면, 반환된 함수의 메타 데이터는 원래 함수 정의가 아닌 wrapper 정의를 반영하게 되어 일반적으로 도움이 되지 않습니다.

`update_wrapper()`는 함수 이외의 콜러블과 함께 사용할 수 있습니다. 래핑 되는 객체에서 누락된 *assigned*나 *updated*로 이름 지정된 어트리뷰트는 무시됩니다(즉, 이 함수는 wrapper 함수에서 그 어트리뷰트를 설정하려고 시도하지 않습니다). wrapper 함수 자체에 *updated*에 이름 지정된 어트리뷰트가 없으면 여전히 `AttributeError`가 발생합니다.

버전 3.2에서 변경: The `__wrapped__` attribute is now automatically added. The `__annotations__` attribute is now copied by default. Missing attributes no longer trigger an `AttributeError`.

버전 3.4에서 변경: `__wrapped__` 어트리뷰트는 이제 해당 함수가 `__wrapped__` 어트리뷰트를 정의한 경우에도 항상 래핑 된 함수를 참조합니다. (bpo-17482를 참조하십시오)

버전 3.12에서 변경: The `__type_params__` attribute is now copied by default.

@functools.**wraps**(*wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

래퍼 함수를 정의할 때 함수 데코레이터로 `update_wrapper()`를 호출하기 위한 편의 함수입니다. `partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)`와 동등합니다. 예를 들면:

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print('Called example function')
...
>>> example()
Calling decorated function
Called example function
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

Without the use of this decorator factory, the name of the example function would have been `'wrapper'`, and the docstring of the original `example()` would have been lost.

10.2.1 partial 객체

partial 객체는 *partial()*이 만든 콜러블 객체입니다. 세 가지 읽기 전용 어트리뷰트가 있습니다:

partial.**func**

콜러블 객체나 함수. *partial* 객체에 대한 호출은 새로운 인자와 키워드와 함께 *func*로 전달됩니다.

partial.**args**

partial 객체 호출에 제공되는 위치 인자 앞에 추가될 가장 왼쪽 위치 인자들입니다.

partial.**keywords**

partial 객체가 호출될 때 제공될 키워드 인자들입니다.

partial objects are like function objects in that they are callable, weak referenceable, and can have attributes. There are some important differences. For instance, the *__name__* and *__doc__* attributes are not created automatically.

10.3 operator — 함수로서의 표준 연산자

소스 코드: Lib/operator.py

---

operator 모듈은 파이썬의 내장 연산자에 해당하는 효율적인 함수 집합을 내보냅니다. 예를 들어, `operator.add(x, y)`는 `x+y` 표현식과 동등합니다. 많은 함수 이름은 특수 메서드에 사용되는 이름인데, 이중 밑줄이 없습니다. 이전 버전과의 호환성을 위해, 이들 중 많은 것은 이중 밑줄이 있는 변형을 가집니다. 이중 밑줄이 없는 변형이 명확성을 위해 선호됩니다.

함수는 객체 비교, 논리 연산, 수학 연산 및 시퀀스 연산을 수행하는 범주로 분류됩니다.

객체 비교 함수는 모든 객체에 유용하며, 이들이 지원하는 풍부한 비교(rich comparison) 연산자의 이름을 따릅니다:

operator.**lt**(*a*, *b*)

operator.**le**(*a*, *b*)

operator.**eq**(*a*, *b*)

operator.**ne**(*a*, *b*)

operator.**ge**(*a*, *b*)

operator.**gt**(*a*, *b*)

operator.**__lt__**(*a*, *b*)

operator.**__le__**(*a*, *b*)

operator.**__eq__**(*a*, *b*)

operator.**__ne__**(*a*, *b*)

operator.**__ge__**(*a*, *b*)

operator.**__gt__**(*a*, *b*)

*a*와 *b* 사이에 "풍부한 비교(rich comparisons)"를 수행합니다. 구체적으로, `lt(a, b)`는 `a < b`와 동등하고, `le(a, b)`는 `a <= b`와 동등하고, `eq(a, b)`는 `a == b`와 동등하고, `ne(a, b)`는 `a != b`와 동등하고, `gt(a, b)`는 `a > b`와 동등하고, `ge(a, b)`는 `a >= b`와 동등합니다. 이러한 함수는 불리언 값으로 해석 할 수도 있고, 그렇지 않을 수도 있는 임의의 값을 반환 할 수 있음에 유의하십시오. 풍부한 비교에 대한 자세한 정보는 comparisons를 참조하십시오.

논리 연산도 일반적으로 모든 객체에 적용 할 수 있으며, 진릿값 검사, 아이덴티티 검사 및 불리언 연산을 지원합니다:

operator.**not_**(*obj*)

operator.**__not__**(*obj*)

not *obj*의 결과를 반환합니다. (객체 인스턴스에는 `__not__()` 메서드가 없음에 유의하십시오; 인터프리터의 코어만이 이 연산을 정의합니다. 결과는 `__bool__()` 과 `__len__()` 메서드의 영향을 받습니다.)

operator.**truth**(*obj*)

*obj*가 참이면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다. 이것은 `bool` 생성자를 사용하는 것과 동등합니다.

operator.**is_**(*a*, *b*)

`a is b`를 반환합니다. 객체 아이덴티티를 검사합니다.

operator.**is_not**(*a*, *b*)

`a is not b`를 반환합니다. 객체 아이덴티티를 검사합니다.

operator.**is_none**(*a*)

Return `a is None`. Tests object identity.

Added in version 3.14.

operator.**is_not_none**(*a*)

Return `a is not None`. Tests object identity.

Added in version 3.14.

수학적 및 비트별 연산이 가장 많습니다:

operator.**abs**(*obj*)

operator.**__abs__**(*obj*)

*obj*의 절댓값을 반환합니다.

operator.**add**(*a*, *b*)

operator.**__add__**(*a*, *b*)

*a*와 *b* 숫자에 대해, `a + b`를 반환합니다.

operator.**and_**(*a*, *b*)

operator.**__and__**(*a*, *b*)

*a*와 *b*의 비트별 논리곱(and)을 반환합니다.

operator.**floordiv**(*a*, *b*)

operator.**__floordiv__**(*a*, *b*)

`a // b`를 반환합니다.

operator.**index**(*a*)

operator.**__index__**(*a*)

정수로 변환된 *a*를 반환합니다. `a.__index__()` 와 동등합니다.

버전 3.10에서 변경: The result always has exact type `int`. Previously, the result could have been an instance of a subclass of `int`.

operator.**inv**(*obj*)

operator.**invert**(*obj*)

operator.**__inv__**(*obj*)

operator.**__invert__**(*obj*)

숫자 *obj*의 비트별 반전을 반환합니다. 이것은 `~obj`와 동등합니다.

operator.**lshift**(*a*, *b*)

operator.**__lshift__**(*a*, *b*)

*a*를 *b*만큼 왼쪽으로 시프트 한 값을 반환합니다.

operator.**mod**(*a*, *b*)

operator.**__mod__**(*a*, *b*)

`a % b`를 반환합니다.

operator.**mul**(*a*, *b*)

operator.**__mul__**(*a*, *b*)

*a*와 *b* 숫자에 대해, `a * b`를 반환합니다.

operator.**matmul**(*a*, *b*)

operator.**__matmul__**(*a*, *b*)

`a @ b`를 반환합니다.

Added in version 3.5.

operator.**neg**(*obj*)

operator.**__neg__**(*obj*)

*obj*의 부정(`-obj`)을 반환합니다.

operator.**or_**(*a*, *b*)

operator.**__or__**(*a*, *b*)

*a*와 *b*의 비트별 논리합(or)을 반환합니다.

operator.**pos**(*obj*)

operator.**__pos__**(*obj*)

양의 *obj*(`+obj`)를 반환합니다.

operator.**pow**(*a*, *b*)

operator.**__pow__**(*a*, *b*)

*a*와 *b* 숫자에 대해, `a ** b`를 반환합니다.

operator.**rshift**(*a*, *b*)

operator.**__rshift__**(*a*, *b*)

*a*를 *b*만큼 오른쪽으로 시프트 한 값을 반환합니다.

operator.**sub**(*a*, *b*)

operator.**__sub__**(*a*, *b*)

`a - b`를 반환합니다.

operator.**truediv**(*a*, *b*)

operator.**__truediv__**(*a*, *b*)

`a / b`를 반환합니다. 여기서 2/3는 0이 아니라 .66입니다. 이것은 "실수(true)" 나누기라고 도합니다.

operator.**xor**(*a*, *b*)

operator.**__xor__**(*a*, *b*)

*a*와 *b*의 비트별 배타적 논리합을 반환합니다.

시퀀스에 적용되는 연산(일부는 매핑에도 적용됩니다)은 다음과 같습니다:

operator.**concat**(*a*, *b*)

operator.**__concat__**(*a*, *b*)

*a*와 *b* 시퀀스에 대해 `a + b`를 반환합니다.

operator.**contains**(*a*, *b*)

operator.**__contains__**(*a*, *b*)

`b in a` 검사의 결과를 반환합니다. 피연산자가 뒤집혀 있음에 유의하십시오.

operator.**countOf**(*a*, *b*)

*a*에서 *b*가 발생하는 횟수를 반환합니다.

operator.**delitem**(*a*, *b*)

operator.**__delitem__**(*a*, *b*)

*a*의 값을 인덱스 *b*에서 제거합니다.

operator.**getitem**(*a*, *b*)

operator.**__getitem__**(*a*, *b*)

인덱스 *b*에 있는 *a*의 값을 반환합니다.

operator.**indexOf**(*a*, *b*)

*a*에서 *b*가 처음으로 발견되는 인덱스를 반환합니다.

operator.**setitem**(*a*, *b*, *c*)

operator.**__setitem__**(*a*, *b*, *c*)

인덱스 *b*의 *a*의 값을 *c*로 설정합니다.

operator.**length_hint**(*obj*, *default=0*)

obj 객체의 추정된 길이를 반환합니다. 먼저 실제 길이를 반환하려고 시도한 다음, `object.__length_hint__()`를 사용하여 추정치를 반환하려고 하고, 마지막으로 default 값을 반환합니다.

Added in version 3.4.

The following operation works with callables:

operator.**call**(*obj*, */*, **args*, ***kwargs*)

operator.**__call__**(*obj*, */*, **args*, ***kwargs*)

`obj(*args, **kwargs)`를 반환합니다.

Added in version 3.11.

operator 모듈은 일반화된 어트리뷰트와 항목 조회를 위한 도구도 정의합니다. 이것은 *map()*, *sorted()*, *itertools.groupby()* 또는 함수 인자를 기대하는 다른 함수의 인자로 사용될 고속 필드 추출기를 만드는 데 유용합니다.

operator.**attrgetter**(*attr*)

operator.**attrgetter**(**attrs*)

피연산자에서 *attr*을 꺼내는 콜러블 객체를 반환합니다. 둘 이상의 어트리뷰트가 요청되면, 어트리뷰트의 튜플을 반환합니다. 어트리뷰트 이름은 점을 포함할 수도 있습니다. 예를 들어:

- `f = attrgetter('name')` 다음에, `f(b)` 호출은 `b.name`을 반환합니다.
- `f = attrgetter('name', 'date')` 다음에, `f(b)` 호출은 `(b.name, b.date)`를 반환합니다.
- `f = attrgetter('name.first', 'name.last')` 다음에, `f(b)` 호출은 `(b.name.first, b.name.last)`를 반환합니다.

다음과 동등합니다:

```
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

operator.**itemgetter**(*item*)

operator.**itemgetter**(**items*)

피연산자의 `__getitem__()` 메서드를 사용하여 피연산자에서 *item*을 꺼내는 콜러블 객체를 반환합니다. 여러 항목이 지정되면, 조회 값의 튜플을 반환합니다. 예를 들어:

- `f = itemgetter(2)` 다음에, `f(r)` 호출은 `r[2]`를 반환합니다.
- `g = itemgetter(2, 5, 3)` 다음에, `g(r)` 호출은 `(r[2], r[5], r[3])`을 반환합니다.

다음과 동등합니다:

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

항목은 피연산자의 `__getitem__()` 메서드에서 허용되는 모든 형이 될 수 있습니다. 딕셔너리는 모든 해시 가능 값을 허용합니다. 리스트, 튜플 및 문자열은 인덱스나 슬라이스를 허용합니다:

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1, 3, 5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2, None))('ABCDEFG')
'CDEFG'
>>> soldier = dict(rank='captain', name='dotterbart')
>>> itemgetter('rank')(soldier)
'captain'
```

튜플 레코드에서 특정 필드를 꺼내기 위해 *itemgetter()*를 사용하는 예:

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.**methodcaller**(*name, /, *args, **kwargs*)

피연산자에서 *name* 메서드를 호출하는 콜러블 객체를 반환합니다. 추가 인자 및/또는 키워드 인자가 주어지면, 해당 인자도 메서드에 제공됩니다. 예를 들어:

- `f = methodcaller('name')` 다음에, `f(b)` 호출은 `b.name()`을 반환합니다.
- `f = methodcaller('name', 'foo', bar=1)` 다음에, `f(b)` 호출은 `b.name('foo', bar=1)`을 반환합니다.

다음과 동등합니다:

```
def methodcaller(name, /, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

10.3.1 연산자를 함수에 매핑하기

이 표는 추상 연산이 파이썬 문법의 연산자 기호와 *operator* 모듈의 함수로 어떻게 대응되는지를 보여줍니다.

연산	문법	함수
더하기 (Addition)	`a + b`	`add(a, b)`
이어붙이기 (Concatenation)	`seq1 + seq2`	`concat(seq1, seq2)`
포함 검사 (Containment Test)	`obj in seq`	`contains(seq, obj)`
나누기 (Division)	`a / b`	`truediv(a, b)`
나누기 (Division)	`a // b`	`floordiv(a, b)`
비트별 논리곱 (Bitwise And)	`a & b`	`and_(a, b)`
비트별 배타적 논리합 (Bitwise Exclusive Or)	`a ^ b`	`xor(a, b)`
비트별 반전 (Bitwise Inversion)	`~ a`	`invert(a)`
비트별 논리합 (Bitwise Or)	`a | b`	`or_(a, b)`
거듭제곱 (Exponentiation)	`a ** b`	`pow(a, b)`
아이덴티티 (Identity)	`a is b`	`is_(a, b)`
아이덴티티 (Identity)	`a is not b`	`is_not(a, b)`
아이덴티티 (Identity)	`a is None`	`is_none(a)`
아이덴티티 (Identity)	`a is not None`	`is_not_none(a)`
인덱싱된 대입 (Indexed Assignment)	`obj[k] = v`	`setitem(obj, k, v)`
인덱싱된 삭제 (Indexed Deletion)	`del obj[k]`	`delitem(obj, k)`
인덱싱 (Indexing)	`obj[k]`	`getitem(obj, k)`
왼쪽으로 시프트 (Left Shift)	`a << b`	`lshift(a, b)`
모듈로 (Modulo)	`a % b`	`mod(a, b)`
곱하기 (Multiplication)	`a * b`	`mul(a, b)`
행렬 곱하기 (Matrix Multiplication)	`a @ b`	`matmul(a, b)`
부정 (산술) (Negation (Arithmetic))	`- a`	`neg(a)`
부정 (논리) (Negation (Logical))	`not a`	`not_(a)`
양 (Positive)	`+ a`	`pos(a)`
오른쪽으로 시프트 (Right Shift)	`a >> b`	`rshift(a, b)`
슬라이스 대입 (Slice Assignment)	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
슬라이스 삭제 (Slice Deletion)	`del seq[i:j]`	`delitem(seq, slice(i, j))`
슬라이싱 (Slicing)	`seq[i:j]`	`getitem(seq, slice(i, j))`
문자열 포매팅 (String Formatting)	`s % obj`	`mod(s, obj)`
빼기 (Subtraction)	`a - b`	`sub(a, b)`
진릿값 검사 (Truth Test)	`obj`	`truth(obj)`
대소비교 (Ordering)	`a < b`	`lt(a, b)`
대소비교 (Ordering)	`a <= b`	`le(a, b)`
동등성 (Equality)	`a == b`	`eq(a, b)`
다름 (Difference)	`a != b`	`ne(a, b)`
대소비교 (Ordering)	`a >= b`	`ge(a, b)`
대소비교 (Ordering)	`a > b`	`gt(a, b)`

10.3.2 제자리 연산자

많은 연산에는 "제자리 (in-place)" 버전이 있습니다. 아래에 나열된 것들은 일반적인 문법보다 제자리 연산자에 대한 더 기본적인 액세스를 제공하는 함수입니다; 예를 들어, 문장 `x += y`는 `x = operator.iadd(x, y)`와 동등합니다. 또 다른 식으로는, `z = operator.iadd(x, y)`가 복합문 `z = x; z += y`와 동등하다고 말하는 것입니다.

이 예제들에서, 제자리 메서드가 호출될 때, 계산과 대입이 두 개의 분리된 단계에서 수행된다는 점에 유의하십시오. 아래 나열된 제자리 함수는 제자리 메서드를 호출하는 첫 번째 단계만 수행합니다. 두 번째 단계인 대입은 처리되지 않습니다.

문자열, 숫자 및 튜플과 같은 불변 대상의 경우, 갱신된 값이 계산되지만, 입력 변수에 다시 할당되지 않습니다:

```
>>> a = 'hello'
>>> iadd(a, ' world')
'hello world'
>>> a
'hello'
```

리스트와 딕셔너리 같은 가변 대상의 경우, 제자리 메서드가 갱신을 수행하므로, 이후 대입이 필요하지 않습니다:

```
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

operator.**iadd**(*a*, *b*)

operator.**__iadd__**(*a*, *b*)

`a = iadd(a, b)`는 `a += b`와 동등합니다.

operator.**iand**(*a*, *b*)

operator.**__iand__**(*a*, *b*)

`a = iand(a, b)`는 `a &= b`와 동등합니다.

operator.**iconcat**(*a*, *b*)

operator.**__iconcat__**(*a*, *b*)

*a*와 *b* 시퀀스에 대해, `a = iconcat(a, b)`는 `a += b`와 동등합니다.

operator.**ifloordiv**(*a*, *b*)

operator.**__ifloordiv__**(*a*, *b*)

`a = ifloordiv(a, b)`는 `a //= b`와 동등합니다.

operator.**ilshift**(*a*, *b*)

operator.**__ilshift__**(*a*, *b*)

`a = ilshift(a, b)`는 `a <<= b`와 동등합니다.

operator.**imod**(*a*, *b*)

operator.**__imod__**(*a*, *b*)

`a = imod(a, b)`는 `a %= b`와 동등합니다.

operator.**imul**(*a*, *b*)

operator.**__imul__**(*a*, *b*)

`a = imul(a, b)`는 `a *= b`와 동등합니다.

operator.**imatmul**(*a*, *b*)

operator.**__imatmul__**(*a*, *b*)

`a = imatmul(a, b)`는 `a @= b`와 동등합니다.

Added in version 3.5.

operator.**ior**(*a*, *b*)

operator.**__ior__**(*a*, *b*)

a = ior(a, b)는 a |= b와 동등합니다.

operator.**ipow**(*a*, *b*)

operator.**__ipow__**(*a*, *b*)

a = ipow(a, b)는 a **= b와 동등합니다.

operator.**irshift**(*a*, *b*)

operator.**__irshift__**(*a*, *b*)

a = irshift(a, b)는 a >>= b와 동등합니다.

operator.**isub**(*a*, *b*)

operator.**__isub__**(*a*, *b*)

a = isub(a, b)는 a -= b와 동등합니다.

operator.**itruediv**(*a*, *b*)

operator.**__itruediv__**(*a*, *b*)

a = itruediv(a, b)는 a /= b와 동등합니다.

operator.**ixor**(*a*, *b*)

operator.**__ixor__**(*a*, *b*)

a = ixor(a, b)는 a ^= b와 동등합니다.

CHAPTER 11

파일과 디렉터리 액세스

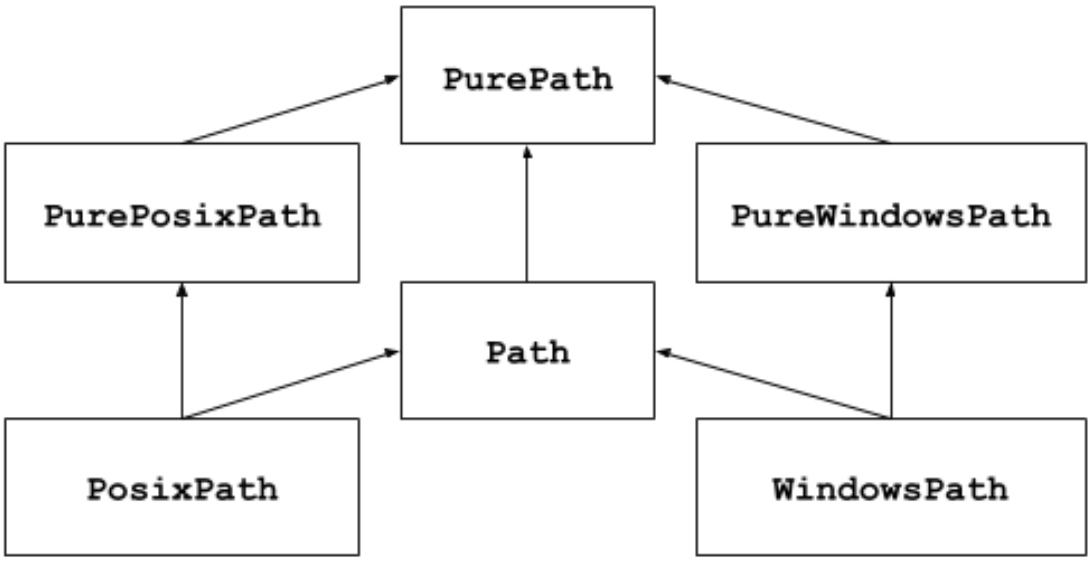
이 장에서 설명하는 모듈은 디스크 파일과 디렉터리를 다룹니다. 예를 들어, 파일의 속성을 읽고, 이식성 있는 방식으로 경로를 조작하고, 임시 파일을 만드는 모듈이 있습니다. 이 장의 전체 모듈 목록은 다음과 같습니다:

11.1 `pathlib` — Object-oriented filesystem paths

Added in version 3.4.

Source code: Lib/pathlib/

이 모듈은 다른 운영 체제에 적합한 의미 체계를 가진 파일 시스템 경로를 나타내는 클래스를 제공합니다. 경로 클래스는 I/O 없이 순수한 계산 연산을 제공하는 순수한 경로와 순수한 경로를 상속하지만, I/O 연산도 제공하는 구상 경로로 구분됩니다.



이전에 이 모듈을 사용한 적이 없거나 어떤 클래스가 작업에 적합한지 확신이 없다면, `Path`가 가장 적합할 가능성이 높습니다. 코드가 실행되는 플랫폼의 구상 경로를 인스턴스화 합니다.

순수한 경로는 특별한 경우에 유용합니다; 예를 들면:

1. 유닉스 기계에서 윈도우 경로를 조작하려고 할 때 (또는 그 반대). 유닉스에서 실행할 때는 `WindowsPath`를 인스턴스화 할 수 없지만, `PureWindowsPath`는 인스턴스화 할 수 있습니다.
2. 코드가 실제로 OS에 액세스하지 않고 경로만 조작한다는 확신이 필요할 때. 이 경우, 순수 클래스 중 하나를 인스턴스화 하면 OS 액세스 연산이 없어서 유용 할 수 있습니다.

더 보기

PEP 428: pathlib 모듈 – 객체 지향 파일 시스템 경로.

더 보기

문자열에 대한 저수준 경로 조작을 위해, `os.path` 모듈을 사용할 수도 있습니다.

11.1.1 기본 사용

메인 클래스 임포트 하기:

```
>>> from pathlib import Path
```

서브 디렉터리 나열하기:

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

이 디렉터리 트리에 있는 파이썬 소스 파일 나열하기:

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

디렉터리 트리 내에서 탐색하기:

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

경로 속성 조회하기:

```
>>> q.exists()
True
>>> q.is_dir()
False
```

파일 열기:

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

11.1.2 Exceptions

exception pathlib.**UnsupportedOperation**

An exception inheriting `NotImplementedError` that is raised when an unsupported operation is called on a path object.

Added in version 3.13.

11.1.3 순수한 경로

순수한 경로 객체는 실제로 파일 시스템에 액세스하지 않는 경로 처리 연산을 제공합니다. 이 클래스에 액세스하는 방법에는 세 가지가 있으며, 플레이버(*flavours*)라고도 부릅니다:

class pathlib.**PurePath**(**pathsegments*)

시스템의 경로 플레이버를 나타내는 일반 클래스 (인스턴스화 하면 `PurePosixPath`나 `PureWindowsPath`를 만듭니다):

```
>>> PurePath('setup.py')      # Running on a Unix machine
PurePosixPath('setup.py')
```

Each element of *pathsegments* can be either a string representing a path segment, or an object implementing the `os.PathLike` interface where the `__fspath__()` method returns a string, such as another path object:

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

*pathsegments*가 비어 있으면, 현재 디렉터리를 가정합니다:

```
>>> PurePath()
PurePosixPath('.')
```

If a segment is an absolute path, all previous segments are ignored (like `os.path.join()`):

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

On Windows, the drive is not reset when a rooted relative path segment (e.g., `r'\foo'`) is encountered:

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

Spurious slashes and single dots are collapsed, but double dots (`'..'`) and leading double slashes (`'//'`) are not, since this would change the meaning of a path for various reasons (e.g. symbolic links, UNC paths):

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('//foo/bar')
PurePosixPath('//foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

(나이브한 접근법은 `PurePosixPath('foo/../bar')`를 `PurePosixPath('bar')`와 동등하게 만드는데, `foo`가 다른 디렉터리에 대한 심볼릭 링크일 때는 잘못됩니다)

순수한 경로 객체는 `os.PathLike` 인터페이스를 구현하여, 이 인터페이스가 허용되는 모든 위치에서 사용할 수 있습니다.

버전 3.6에서 변경: `os.PathLike` 인터페이스에 대한 지원이 추가되었습니다.

class pathlib.**PurePosixPath**(**pathsegments*)

`PurePath`의 서브 클래스, 이 경로 플레이버는 윈도우 이외의 파일 시스템 경로를 나타냅니다:

```
>>> PurePosixPath('/etc/hosts')
PurePosixPath('/etc/hosts')
```

*pathsegments*는 `PurePath`와 유사하게 지정됩니다.

class pathlib.**PureWindowsPath**(**pathsegments*)

A subclass of `PurePath`, this path flavour represents Windows filesystem paths, including UNC paths:

```
>>> PureWindowsPath('c:/', 'Users', 'Ximénez')
PureWindowsPath('c:/Users/Ximénez')
>>> PureWindowsPath('//server/share/file')
PureWindowsPath('//server/share/file')
```

*pathsegments*는 `PurePath`와 유사하게 지정됩니다.

실행 중인 시스템과 관계없이, 이러한 모든 클래스를 인스턴스화 할 수 있는데, 시스템 호출을 수행하는 연산을 제공하지 않기 때문입니다.

일반 속성

Paths are immutable and *hashable*. Paths of a same flavour are comparable and orderable. These properties respect the flavour' s case-folding semantics:

```
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

다른 플레이버의 경로는 다르다고 비교되며 대소 비교할 수 없습니다:

```
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'PureWindowsPath' and
↪'PurePosixPath'
```

연산자

The slash operator helps create child paths, like *os.path.join()*. If the argument is an absolute path, the previous path is ignored. On Windows, the drive is not reset when the argument is a rooted relative path (e.g., `r'\foo'`):

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
>>> p / '/an_absolute_path'
PurePosixPath('/an_absolute_path')
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

경로 객체는 *os.PathLike*을 구현하는 객체가 허용되는 모든 곳에서 사용할 수 있습니다:

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

경로의 문자열 표현은 원시 파일 시스템 경로 자체(네이티브 형식으로, 예를 들어 윈도우에서 역 슬래시)로, 파일 경로를 문자열로 받아들이는 모든 함수에 전달할 수 있습니다:

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
>>> str(p)
'c:\\Program Files'
```

마찬가지로, 경로에 대해 *bytes*를 호출하면 *os.fsencode()*로 인코딩된 바이트열 객체로 원시 파일 시스템 경로를 제공합니다:

```
>>> bytes(p)
b'/etc'
```

> **참고**
>
> *bytes* 호출은 유닉스에서만 권장됩니다. 윈도우에서, 유니코드 형식이 파일 시스템 경로의 규범적 (canonical) 표현입니다.

개별 부분에 액세스하기

경로의 개별 "부분"(구성 요소)에 액세스하려면, 다음 프로퍼티를 사용하십시오:

`PurePath.`**`parts`**

경로의 다양한 구성 요소로의 액세스를 제공하는 튜플:

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')

>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

(드라이브와 로컬 루트가 단일 부분으로 다시 그룹화되는 방식에 유의하십시오)

메서드와 프로퍼티

순수한 경로는 다음과 같은 메서드와 프로퍼티를 제공합니다:

`PurePath.`**`parser`**

The implementation of the *os.path* module used for low-level path parsing and joining: either `posixpath` or `ntpath`.

Added in version 3.13.

`PurePath.`**`drive`**

드라이브 문자나 이름을 나타내는 문자열, 있다면:

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

UNC 공유도 드라이브로 간주합니다:

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\host\\share'
```

`PurePath.`**`root`**

(로컬이나 글로벌) 루트를 나타내는 문자열, 있다면:

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

UNC 공유에는 항상 루트가 있습니다:

```
>>> PureWindowsPath('//host/share').root
'\\'
```

If the path starts with more than two successive slashes, *PurePosixPath* collapses them:

```
>>> PurePosixPath('//etc').root
'//'
>>> PurePosixPath('///etc').root
'/'
>>> PurePosixPath('////etc').root
'/'
```

> **참고**
>
> This behavior conforms to *The Open Group Base Specifications Issue 6*, paragraph 4.11 Pathname Resolution:
>
> "*A pathname that begins with two successive slashes may be interpreted in an implementation-defined manner, although more than two leading slashes shall be treated as a single slash.*"

`PurePath.`**`anchor`**

드라이브와 루트의 이어 붙이기:

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\\host\\share\\'
```

`PurePath.`**`parents`**

경로의 논리적 조상에 대한 액세스를 제공하는 불변 시퀀스:

```
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
>>> p.parents[1]
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

버전 3.10에서 변경: The parents sequence now supports *slices* and negative index values.

`PurePath.`**`parent`**

경로의 논리적 부모:

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```

앵커나 빈 경로를 넘어갈 수 없습니다:

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

> **참고**
>
> 이것은 순수한 어휘(lexical) 연산이라서, 다음과 같이 동작합니다:
>
> ```
> >>> p = PurePosixPath('foo/..')
> >>> p.parent
> PurePosixPath('foo')
> ```
>
> If you want to walk an arbitrary filesystem path upwards, it is recommended to first call `Path.resolve()` so as to resolve symlinks and eliminate `"..."` components.

`PurePath.name`

드라이브와 루트를 제외하고, 마지막 경로 구성 요소를 나타내는 문자열, 있다면:

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

UNC 드라이브 이름은 고려되지 않습니다:

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

`PurePath.suffix`

The last dot-separated portion of the final component, if any:

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

This is commonly called the file extension.

버전 3.14에서 변경: A single dot (".") is considered a valid suffix.

`PurePath.suffixes`

A list of the path' s suffixes, often called file extensions:

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

버전 3.14에서 변경: A single dot (".") is considered a valid suffix.

`PurePath.stem`

suffix가 없는, 마지막 경로 구성 요소:

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

PurePath.**as_posix**()

슬래시(/)가 있는 경로의 문자열 표현을 반환합니다:

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

PurePath.**is_absolute**()

경로가 절대적인지 아닌지를 반환합니다. 루트와 (플레이버가 허락하면) 드라이브가 모두 있으면 경로를 절대적이라고 간주합니다:

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False

>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
>>> PureWindowsPath('//some/share').is_absolute()
True
```

PurePath.**is_relative_to**(*other*)

이 경로가 *other* 경로에 상대적인지를 반환합니다.

```
>>> p = PurePath('/etc/passwd')
>>> p.is_relative_to('/etc')
True
>>> p.is_relative_to('/usr')
False
```

This method is string-based; it neither accesses the filesystem nor treats "..." segments specially. The following code is equivalent:

```
>>> u = PurePath('/usr')
>>> u == p or u in p.parents
False
```

Added in version 3.9.

Deprecated since version 3.12, removed in version 3.14: Passing additional arguments is deprecated; if supplied, they are joined with *other*.

PurePath.**is_reserved**()

*PureWindowsPath*에서는, 경로를 윈도우에서 예약된 것으로 간주하면 `True`를, 그렇지 않으면 `False`를 반환합니다. *PurePosixPath*에서는, 항상 `False`가 반환됩니다.

버전 3.13에서 변경: Windows path names that contain a colon, or end with a dot or a space, are considered reserved. UNC paths may be reserved.

Deprecated since version 3.13, will be removed in version 3.15: This method is deprecated; use `os.path.isreserved()` to detect reserved paths on Windows.

`PurePath.joinpath`(**pathsegments*)

Calling this method is equivalent to combining the path with each of the given *pathsegments* in turn:

```
>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath(PurePosixPath('passwd'))
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

`PurePath.full_match`(*pattern, *, case_sensitive=None*)

Match this path against the provided glob-style pattern. Return `True` if matching is successful, `False` otherwise. For example:

```
>>> PurePath('a/b.py').full_match('a/*.py')
True
>>> PurePath('a/b.py').full_match('*.py')
False
>>> PurePath('/a/b/c.py').full_match('/a/**')
True
>>> PurePath('/a/b/c.py').full_match('**/*.py')
True
```

더 보기

Pattern language documentation.

다른 메서드와 마찬가지로, 대소 문자를 구분할지는 플랫폼 기본값을 따릅니다:

```
>>> PurePosixPath('b.py').full_match('*.PY')
False
>>> PureWindowsPath('b.py').full_match('*.PY')
True
```

Set *case_sensitive* to `True` or `False` to override this behaviour.

Added in version 3.13.

`PurePath.match`(*pattern, *, case_sensitive=None*)

Match this path against the provided non-recursive glob-style pattern. Return `True` if matching is successful, `False` otherwise.

This method is similar to `full_match()`, but empty patterns aren' t allowed (`ValueError` is raised), the recursive wildcard "`**`" isn' t supported (it acts like non-recursive "`*`"), and if a relative pattern is provided, then matching is done from the right:

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

버전 3.12에서 변경: The *pattern* parameter accepts a *path-like object*.

버전 3.12에서 변경: The *case_sensitive* parameter was added.

`PurePath.relative_to`(*other*, *walk_up=False*)

Compute a version of this path relative to the path represented by *other*. If it' s impossible, `ValueError` is raised:

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
>>> p.relative_to('/etc')
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 941, in relative_to
    raise ValueError(error_message.format(str(self), str(formatted)))
ValueError: '/etc/passwd' is not in the subpath of '/usr' OR one path is␣
↪relative and the other is absolute.
```

When *walk_up* is false (the default), the path must start with *other*. When the argument is true, `..` entries may be added to form the relative path. In all other cases, such as the paths referencing different drives, `ValueError` is raised.:

```
>>> p.relative_to('/usr', walk_up=True)
PurePosixPath('../etc/passwd')
>>> p.relative_to('foo', walk_up=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 941, in relative_to
    raise ValueError(error_message.format(str(self), str(formatted)))
ValueError: '/etc/passwd' is not on the same drive as 'foo' OR one path is␣
↪relative and the other is absolute.
```

> **경고**
>
> This function is part of `PurePath` and works with strings. It does not check or access the underlying file structure. This can impact the *walk_up* option as it assumes that no symlinks are present in the path; call `resolve()` first if necessary to resolve symlinks.

버전 3.12에서 변경: The *walk_up* parameter was added (old behavior is the same as `walk_up=False`).

Deprecated since version 3.12, removed in version 3.14: Passing additional positional arguments is deprecated; if supplied, they are joined with *other*.

`PurePath.with_name`(*name*)

`name`이 변경된 새 경로를 반환합니다. 원래 경로에 이름(name)이 없으면 ValueError가 발생합니다:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 751, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

PurePath.**with_stem**(*stem*)

*stem*이 변경된 새 경로를 반환합니다. 원래 경로에 이름(name)이 없으면, ValueError가 발생합니다:

```
>>> p = PureWindowsPath('c:/Downloads/draft.txt')
>>> p.with_stem('final')
PureWindowsPath('c:/Downloads/final.txt')
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_stem('lib')
PureWindowsPath('c:/Downloads/lib.gz')
>>> p = PureWindowsPath('c:/')
>>> p.with_stem('')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 861, in with_stem
    return self.with_name(stem + self.suffix)
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 851, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

Added in version 3.9.

PurePath.**with_suffix**(*suffix*)

*suffix*가 변경된 새 경로를 반환합니다. 원래 경로에 접미사(suffix)가 없으면, 새 *suffix*가 대신 추가됩니다. *suffix*가 빈 문자열이면, 원래 접미사가 제거됩니다:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
>>> p = PureWindowsPath('README.txt')
>>> p.with_suffix('')
PureWindowsPath('README')
```

버전 3.14에서 변경: A single dot (".") is considered a valid suffix. In previous versions, *ValueError* is raised if a single dot is supplied.

PurePath.**with_segments**(**pathsegments*)

Create a new path object of the same type by combining the given *pathsegments*. This method is called whenever a derivative path is created, such as from *parent* and *relative_to()*. Subclasses may override this method to pass information to derivative paths, for example:

```
from pathlib import PurePosixPath

class MyPath(PurePosixPath):
    def __init__(self, *pathsegments, session_id):
        super().__init__(*pathsegments)
        self.session_id = session_id

    def with_segments(self, *pathsegments):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        return type(self)(*pathsegments, session_id=self.session_id)

etc = MyPath('/etc', session_id=42)
hosts = etc / 'hosts'
print(hosts.session_id)  # 42
```

Added in version 3.12.

11.1.4 구상 경로

구상 경로는 순수한 경로 클래스의 서브 클래스입니다. 후자가 제공하는 연산 외에도, 경로 객체에 대해 시스템 호출을 수행하는 메서드도 제공합니다. 구상 경로를 인스턴스화 하는 세 가지 방법이 있습니다:

class pathlib.**Path**(**pathsegments*)

*PurePath*의 서브 클래스, 이 클래스는 시스템의 경로 플레이버의 구상 경로를 나타냅니다 (인스턴스화 하면 *PosixPath*나 *WindowsPath*를 만듭니다):

```
>>> Path('setup.py')
PosixPath('setup.py')
```

*pathsegments*는 *PurePath*와 유사하게 지정됩니다.

class pathlib.**PosixPath**(**pathsegments*)

*Path*와 *PurePosixPath*의 서브 클래스, 이 클래스는 윈도우 이외의 구상 파일 시스템 경로를 나타냅니다:

```
>>> PosixPath('/etc/hosts')
PosixPath('/etc/hosts')
```

*pathsegments*는 *PurePath*와 유사하게 지정됩니다.

버전 3.13 에서 변경: Raises *UnsupportedOperation* on Windows. In previous versions, *NotImplementedError* was raised instead.

class pathlib.**WindowsPath**(**pathsegments*)

*Path*와 *PureWindowsPath*의 서브 클래스, 이 클래스는 구상 윈도우 파일 시스템 경로를 나타냅니다:

```
>>> WindowsPath('c:/', 'Users', 'Ximénez')
WindowsPath('c:/Users/Ximénez')
```

*pathsegments*는 *PurePath*와 유사하게 지정됩니다.

버전 3.13에서 변경: Raises *UnsupportedOperation* on non-Windows platforms. In previous versions, *NotImplementedError* was raised instead.

여러분의 시스템에 해당하는 클래스 플레이버만 인스턴스화 할 수 있습니다 (호환되지 않는 경로 플레이버에 대한 시스템 호출을 허용하면 응용 프로그램에서 버그나 실패가 발생할 수 있습니다):

```
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    % (cls.__name__,))
UnsupportedOperation: cannot instantiate 'WindowsPath' on your system
```

Some concrete path methods can raise an `OSError` if a system call fails (for example because the path doesn't exist).

Parsing and generating URIs

Concrete path objects can be created from, and represented as, 'file' URIs conforming to **RFC 8089**.

> **참고**
>
> File URIs are not portable across machines with different *filesystem encodings*.

classmethod `Path.`**`from_uri`**`(`*uri*`)`

Return a new path object from parsing a 'file' URI. For example:

```
>>> p = Path.from_uri('file:///etc/hosts')
PosixPath('/etc/hosts')
```

On Windows, DOS device and UNC paths may be parsed from URIs:

```
>>> p = Path.from_uri('file:///c:/windows')
WindowsPath('c:/windows')
>>> p = Path.from_uri('file://server/share')
WindowsPath('//server/share')
```

Several variant forms are supported:

```
>>> p = Path.from_uri('file:////server/share')
WindowsPath('//server/share')
>>> p = Path.from_uri('file://///server/share')
WindowsPath('//server/share')
>>> p = Path.from_uri('file:c:/windows')
WindowsPath('c:/windows')
>>> p = Path.from_uri('file:/c|/windows')
WindowsPath('c:/windows')
```

`ValueError` is raised if the URI does not start with `file:`, or the parsed path isn't absolute.

Added in version 3.13.

버전 3.14에서 변경: The URL authority is discarded if it matches the local hostname. Otherwise, if the authority isn't empty or `localhost`, then on Windows a UNC path is returned (as before), and on other platforms a `ValueError` is raised.

`Path.`**`as_uri`**`()`

Represent the path as a 'file' URI. `ValueError` is raised if the path isn't absolute.

```
>>> p = PosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = WindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

Deprecated since version 3.14, will be removed in version 3.19: Calling this method from `PurePath` rather than `Path` is possible but deprecated. The method's use of `os.fsencode()` makes it strictly impure.

Expanding and resolving paths

classmethod Path.**home**()

Return a new path object representing the user' s home directory (as returned by *os.path.expanduser()* with ~ construct). If the home directory can' t be resolved, *RuntimeError* is raised.

```
>>> Path.home()
PosixPath('/home/antoine')
```

Added in version 3.5.

Path.**expanduser**()

Return a new path with expanded ~ and ~user constructs, as returned by *os.path.expanduser()*. If a home directory can' t be resolved, *RuntimeError* is raised.

```
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

Added in version 3.5.

classmethod Path.**cwd**()

현재 디렉터리를 나타내는 새 경로 객체를 반환합니다. *os.getcwd()*가 반환하는 것과 유사합니다:

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

Path.**absolute**()

Make the path absolute, without normalization or resolving symlinks. Returns a new path object:

```
>>> p = Path('tests')
>>> p
PosixPath('tests')
>>> p.absolute()
PosixPath('/home/antoine/pathlib/tests')
```

Path.**resolve**(*strict=False*)

심볼릭 링크를 결정하여, 경로를 절대적으로 만듭니다. 새로운 경로 객체가 반환됩니다:

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

"..." 구성 요소도 제거됩니다 (이것이 이렇게 하는 유일한 메서드입니다):

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

If a path doesn' t exist or a symlink loop is encountered, and *strict* is `True`, *OSError* is raised. If *strict* is `False`, the path is resolved as far as possible and any remainder is appended without checking whether it exists.

버전 3.6에서 변경: The *strict* parameter was added (pre-3.6 behavior is strict).

버전 3.13에서 변경: Symlink loops are treated like other errors: *OSError* is raised in strict mode, and no exception is raised in non-strict mode. In previous versions, *RuntimeError* is raised no matter the value of *strict*.

Path.**readlink**()

심볼릭 링크가 가리키는 경로를 반환합니다 (*os.readlink()* 가 반환하는 것과 유사합니다):

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.readlink()
PosixPath('setup.py')
```

Added in version 3.9.

버전 3.13에서 변경: Raises *UnsupportedOperation* if *os.readlink()* is not available. In previous versions, *NotImplementedError* was raised.

Querying file type and status

버전 3.8에서 변경: *exists()*, *is_dir()*, *is_file()*, *is_mount()*, *is_symlink()*, *is_block_device()*, *is_char_device()*, *is_fifo()*, *is_socket()* now return `False` instead of raising an exception for paths that contain characters unrepresentable at the OS level.

버전 3.14에서 변경: The methods given above now return `False` instead of raising any *OSError* exception from the operating system. In previous versions, some kinds of *OSError* exception are raised, and others suppressed. The new behaviour is consistent with *os.path.exists()*, *os.path.isdir()*, etc. Use *stat()* to retrieve the file status without suppressing exceptions.

Path.**stat**(*, *follow_symlinks=True*)

Return an *os.stat_result* object containing information about this path, like *os.stat()*. The result is looked up at each call to this method.

This method normally follows symlinks; to stat a symlink add the argument `follow_symlinks=False`, or use *lstat()*.

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

버전 3.10에서 변경: The *follow_symlinks* parameter was added.

Path.**lstat**()

Path.stat() 과 비슷하지만, 경로가 심볼릭 링크를 가리키면, 대상이 아닌 심볼릭 링크의 정보를 반환합니다.

Path.**exists**(*, *follow_symlinks=True*)

Return `True` if the path points to an existing file or directory. `False` will be returned if the path is invalid, inaccessible or missing. Use *Path.stat()* to distinguish between these cases.

This method normally follows symlinks; to check if a symlink exists, add the argument `follow_symlinks=False`.

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

버전 3.12에서 변경: The *follow_symlinks* parameter was added.

`Path.is_file(*, follow_symlinks=True)`

Return `True` if the path points to a regular file. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a regular file. Use `Path.stat()` to distinguish between these cases.

This method normally follows symlinks; to exclude symlinks, add the argument `follow_symlinks=False`.

버전 3.13에서 변경: The *follow_symlinks* parameter was added.

`Path.is_dir(*, follow_symlinks=True)`

Return `True` if the path points to a directory. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a directory. Use `Path.stat()` to distinguish between these cases.

This method normally follows symlinks; to exclude symlinks to directories, add the argument `follow_symlinks=False`.

버전 3.13에서 변경: The *follow_symlinks* parameter was added.

`Path.is_symlink()`

Return `True` if the path points to a symbolic link, even if that symlink is broken. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a symbolic link. Use `Path.stat()` to distinguish between these cases.

`Path.is_junction()`

Return `True` if the path points to a junction, and `False` for any other type of file. Currently only Windows supports junctions.

Added in version 3.12.

`Path.is_mount()`

Return `True` if the path is a *mount point*: a point in a file system where a different file system has been mounted. On POSIX, the function checks whether *path*' s parent, `path/..`, is on a different device than *path*, or whether `path/..` and *path* point to the same i-node on the same device — this should detect mount points for all Unix and POSIX variants. On Windows, a mount point is considered to be a drive letter root (e.g. `c:\`), a UNC share (e.g. `\\server\share`), or a mounted filesystem directory.

Added in version 3.7.

버전 3.12에서 변경: Windows support was added.

`Path.is_socket()`

Return `True` if the path points to a Unix socket. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a Unix socket. Use `Path.stat()` to distinguish between these cases.

`Path.is_fifo()`

Return `True` if the path points to a FIFO. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a FIFO. Use `Path.stat()` to distinguish between these cases.

`Path.is_block_device()`

Return `True` if the path points to a block device. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a block device. Use `Path.stat()` to distinguish between these cases.

`Path.is_char_device()`

Return `True` if the path points to a character device. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a character device. Use `Path.stat()` to distinguish between these cases.

`Path.samefile(other_path)`

이 경로가 *other_path*와 같은 파일을 가리키는지를 반환합니다. *other_path*는 Path 객체이거나 문자열일 수 있습니다. 의미는 `os.path.samefile()`과 `os.path.samestat()`과 유사합니다.

어떤 이유로 파일에 액세스할 수 없으면 `OSError`가 발생할 수 있습니다.

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

Added in version 3.5.

`Path.info`

A `PathInfo` object that supports querying file type information. The object exposes methods that cache their results, which can help reduce the number of system calls needed when switching on file type. For example:

```
>>> p = Path('src')
>>> if p.info.is_symlink():
...     print('symlink')
... elif p.info.is_dir():
...     print('directory')
... elif p.info.exists():
...     print('something else')
... else:
...     print('not found')
...
directory
```

If the path was generated from `Path.iterdir()` then this attribute is initialized with some information about the file type gleaned from scanning the parent directory. Merely accessing `Path.info` does not perform any filesystem queries.

To fetch up-to-date information, it' s best to call `Path.is_dir()`, `is_file()` and `is_symlink()` rather than methods of this attribute. There is no way to reset the cache; instead you can create a new path object with an empty info cache via `p = Path(p)`.

Added in version 3.14.

Reading and writing files

`Path.open`(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)

내장 `open()` 함수처럼, 경로가 가리키는 파일을 엽니다:

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
...
'#!/usr/bin/env python3\n'
```

`Path.read_text`(*encoding=None*, *errors=None*, *newline=None*)

가리키는 파일의 디코딩된 내용을 문자열로 반환합니다:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

파일이 열린 다음에 닫힙니다. 선택적 매개 변수는 `open()`과 같은 의미입니다.

Added in version 3.5.

버전 3.13에서 변경: The *newline* parameter was added.

Path.**read_bytes**()

가리키는 파일의 바이너리 내용을 바이트열 객체로 반환합니다:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

Added in version 3.5.

Path.**write_text**(*data*, *encoding=None*, *errors=None*, *newline=None*)

가리키는 파일을 텍스트 모드로 열고, *data*를 쓴 다음, 파일을 닫습니다:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

같은 이름의 기존 파일을 덮어씁니다. 선택적 매개 변수는 `open()`에서와 같은 의미입니다.

Added in version 3.5.

버전 3.10에서 변경: The *newline* parameter was added.

Path.**write_bytes**(*data*)

가리키는 파일을 바이너리 모드로 열고, *data*를 쓴 다음, 파일을 닫습니다:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

같은 이름의 기존 파일을 덮어씁니다.

Added in version 3.5.

Reading directories

Path.**iterdir**()

경로가 디렉터리를 가리킬 때, 디렉터리 내용의 경로 객체를 산출합니다:

```
>>> p = Path('docs')
>>> for child in p.iterdir(): child
...
PosixPath('docs/conf.py')
PosixPath('docs/_templates')
PosixPath('docs/make.bat')
PosixPath('docs/index.rst')
PosixPath('docs/_build')
PosixPath('docs/_static')
PosixPath('docs/Makefile')
```

The children are yielded in arbitrary order, and the special entries `'.'` and `'..'` are not included. If a file is removed from or added to the directory after creating the iterator, it is unspecified whether a path object for that file is included.

If the path is not a directory or otherwise inaccessible, `OSError` is raised.

`Path.glob`(*pattern*, *, *case_sensitive=None*, *recurse_symlinks=False*)

이 경로로 표현되는 디렉터리에서, 주어진 상대 *pattern*을 glob 하여, 일치하는 모든 파일을 (종류와 관계없이) 산출합니다:

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('*/*.py'))
[PosixPath('docs/conf.py')]
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

더 보기

Pattern language documentation.

By default, or when the *case_sensitive* keyword-only argument is set to `None`, this method matches paths using platform-specific casing rules: typically, case-sensitive on POSIX, and case-insensitive on Windows. Set *case_sensitive* to `True` or `False` to override this behaviour.

By default, or when the *recurse_symlinks* keyword-only argument is set to `False`, this method follows symlinks except when expanding "`**`" wildcards. Set *recurse_symlinks* to `True` to always follow symlinks.

인자 `self`, `pattern`으로 감사 이벤트 `pathlib.Path.glob`을 발생시킵니다.

버전 3.12에서 변경: The *case_sensitive* parameter was added.

버전 3.13에서 변경: The *recurse_symlinks* parameter was added.

버전 3.13에서 변경: The *pattern* parameter accepts a *path-like object*.

버전 3.13에서 변경: Any `OSError` exceptions raised from scanning the filesystem are suppressed. In previous versions, such exceptions are suppressed in many cases, but not all.

`Path.rglob`(*pattern*, *, *case_sensitive=None*, *recurse_symlinks=False*)

Glob the given relative *pattern* recursively. This is like calling `Path.glob()` with "`**/`" added in front of the *pattern*.

더 보기

Pattern language and `Path.glob()` documentation.

인자 `self`, `pattern`으로 감사 이벤트 `pathlib.Path.rglob`을 발생시킵니다.

버전 3.12에서 변경: The *case_sensitive* parameter was added.

버전 3.13에서 변경: The *recurse_symlinks* parameter was added.

버전 3.13에서 변경: The *pattern* parameter accepts a *path-like object*.

`Path.walk`(*top_down=True*, *on_error=None*, *follow_symlinks=False*)

Generate the file names in a directory tree by walking the tree either top-down or bottom-up.

For each directory in the directory tree rooted at *self* (including *self* but excluding '.' and '..'), the method yields a 3-tuple of `(dirpath, dirnames, filenames)`.

dirpath is a `Path` to the directory currently being walked, *dirnames* is a list of strings for the names of subdirectories in *dirpath* (excluding `'.'` and `'..'`), and *filenames* is a list of strings for the names of the non-directory

files in *dirpath*. To get a full path (which begins with *self*) to a file or directory in *dirpath*, do `dirpath / name`. Whether or not the lists are sorted is file system-dependent.

If the optional argument *top_down* is true (which is the default), the triple for a directory is generated before the triples for any of its subdirectories (directories are walked top-down). If *top_down* is false, the triple for a directory is generated after the triples for all of its subdirectories (directories are walked bottom-up). No matter the value of *top_down*, the list of subdirectories is retrieved before the triples for the directory and its subdirectories are walked.

When *top_down* is true, the caller can modify the *dirnames* list in-place (for example, using `del` or slice assignment), and `Path.walk()` will only recurse into the subdirectories whose names remain in *dirnames*. This can be used to prune the search, or to impose a specific order of visiting, or even to inform `Path.walk()` about directories the caller creates or renames before it resumes `Path.walk()` again. Modifying *dirnames* when *top_down* is false has no effect on the behavior of `Path.walk()` since the directories in *dirnames* have already been generated by the time *dirnames* is yielded to the caller.

By default, errors from `os.scandir()` are ignored. If the optional argument *on_error* is specified, it should be a callable; it will be called with one argument, an `OSError` instance. The callable can handle the error to continue the walk or re-raise it to stop the walk. Note that the filename is available as the `filename` attribute of the exception object.

By default, `Path.walk()` does not follow symbolic links, and instead adds them to the *filenames* list. Set *follow_symlinks* to true to resolve symlinks and place them in *dirnames* and *filenames* as appropriate for their targets, and consequently visit directories pointed to by symlinks (where supported).

> **참고**
>
> Be aware that setting *follow_symlinks* to true can lead to infinite recursion if a link points to a parent directory of itself. `Path.walk()` does not keep track of the directories it has already visited.

> **참고**
>
> `Path.walk()` assumes the directories it walks are not modified during execution. For example, if a directory from *dirnames* has been replaced with a symlink and *follow_symlinks* is false, `Path.walk()` will still try to descend into it. To prevent such behavior, remove directories from *dirnames* as appropriate.

> **참고**
>
> Unlike `os.walk()`, `Path.walk()` lists symlinks to directories in *filenames* if *follow_symlinks* is false.

This example displays the number of bytes used by all files in each directory, while ignoring `__pycache__` directories:

```
from pathlib import Path
for root, dirs, files in Path("cpython/Lib/concurrent").walk(on_error=print):
  print(
      root,
      "consumes",
      sum((root / file).stat().st_size for file in files),
      "bytes in",
      len(files),
      "non-directory files"
  )
  if '__pycache__' in dirs:
        dirs.remove('__pycache__')
```

This next example is a simple implementation of `shutil.rmtree()`. Walking the tree bottom-up is essential as `rmdir()` doesn' t allow deleting a directory before it is empty:

```
# Delete everything reachable from the directory "top".
# CAUTION:  This is dangerous! For example, if top == Path('/'),
# it could delete all of your files.
for root, dirs, files in top.walk(top_down=False):
    for name in files:
        (root / name).unlink()
    for name in dirs:
        (root / name).rmdir()
```

Added in version 3.12.

Creating files and directories

`Path.touch`(*mode=0o666*, *exist_ok=True*)

Create a file at this given path. If *mode* is given, it is combined with the process' s `umask` value to determine the file mode and access flags. If the file already exists, the function succeeds when *exist_ok* is true (and its modification time is updated to the current time), otherwise `FileExistsError` is raised.

> **더 보기**
>
> The `open()`, `write_text()` and `write_bytes()` methods are often used to create files.

`Path.mkdir`(*mode=0o777*, *parents=False*, *exist_ok=False*)

Create a new directory at this given path. If *mode* is given, it is combined with the process' s `umask` value to determine the file mode and access flags. If the path already exists, `FileExistsError` is raised.

*parents*가 참이면, 이 경로의 누락 된 부모를 필요하면 만듭니다; 이것들은 *mode*를 고려하지 않고 기본 권한으로 만들어집니다 (POSIX `mkdir -p` 명령을 모방합니다).

*parents*가 거짓(기본값)이면, 누락된 부모가 `FileNotFoundError`를 발생시킵니다.

*exist_ok*가 거짓(기본값)이면, 대상 디렉터리가 이미 존재하면 `FileExistsError`가 발생합니다.

If *exist_ok* is true, `FileExistsError` will not be raised unless the given path already exists in the file system and is not a directory (same behavior as the POSIX `mkdir -p` command).

버전 3.5에서 변경: *exist_ok* 매개 변수가 추가되었습니다.

`Path.symlink_to`(*target*, *target_is_directory=False*)

Make this path a symbolic link pointing to *target*.

On Windows, a symlink represents either a file or a directory, and does not morph to the target dynamically. If the target is present, the type of the symlink will be created to match. Otherwise, the symlink will be created as a directory if *target_is_directory* is true or a file symlink (the default) otherwise. On non-Windows platforms, *target_is_directory* is ignored.

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
>>> p.lstat().st_size
8
```

> **참고**
>
> 인자의 순서(링크, 대상)는 `os.symlink()`와 반대입니다.

버전 3.13에서 변경: Raises `UnsupportedOperation` if `os.symlink()` is not available. In previous versions, `NotImplementedError` was raised.

`Path.`**`hardlink_to`**(*target*)

Make this path a hard link to the same file as *target*.

> **참고**
>
> The order of arguments (link, target) is the reverse of `os.link()`' s.

Added in version 3.10.

버전 3.13에서 변경: Raises `UnsupportedOperation` if `os.link()` is not available. In previous versions, `NotImplementedError` was raised.

Copying, moving and deleting

`Path.`**`copy`**(*target*, *, *follow_symlinks=True*, *preserve_metadata=False*)

Copy this file or directory tree to the given *target*, and return a new `Path` instance pointing to *target*.

If the source is a file, the target will be replaced if it is an existing file. If the source is a symlink and *follow_symlinks* is true (the default), the symlink' s target is copied. Otherwise, the symlink is recreated at the destination.

If *preserve_metadata* is false (the default), only directory structures and file data are guaranteed to be copied. Set *preserve_metadata* to true to ensure that file and directory permissions, flags, last access and modification times, and extended attributes are copied where supported. This argument has no effect when copying files on Windows (where metadata is always preserved).

> **참고**
>
> Where supported by the operating system and file system, this method performs a lightweight copy, where data blocks are only copied when modified. This is known as copy-on-write.

Added in version 3.14.

`Path.`**`copy_into`**(*target_dir*, *, *follow_symlinks=True*, *preserve_metadata=False*)

Copy this file or directory tree into the given *target_dir*, which should be an existing directory. Other arguments are handled identically to `Path.copy()`. Returns a new `Path` instance pointing to the copy.

Added in version 3.14.

`Path.`**`rename`**(*target*)

Rename this file or directory to the given *target*, and return a new `Path` instance pointing to *target*. On Unix, if *target* exists and is a file, it will be replaced silently if the user has permission. On Windows, if *target* exists, `FileExistsError` will be raised. *target* can be either a string or another path object:

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
PosixPath('bar')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> target.open().read()
'some text'
```

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the `Path` object.

It is implemented in terms of `os.rename()` and gives the same guarantees.

버전 3.8에서 변경: Added return value, return the new `Path` instance.

`Path.replace(target)`

Rename this file or directory to the given *target*, and return a new `Path` instance pointing to *target*. If *target* points to an existing file or empty directory, it will be unconditionally replaced.

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, *not* the directory of the `Path` object.

버전 3.8에서 변경: Added return value, return the new `Path` instance.

`Path.move(target)`

Move this file or directory tree to the given *target*, and return a new `Path` instance pointing to *target*.

If the *target* doesn' t exist it will be created. If both this path and the *target* are existing files, then the target is overwritten. If both paths point to the same file or directory, or the *target* is a non-empty directory, then `OSError` is raised.

If both paths are on the same filesystem, the move is performed with `os.replace()`. Otherwise, this path is copied (preserving metadata and symlinks) and then deleted.

Added in version 3.14.

`Path.move_into(target_dir)`

Move this file or directory tree into the given *target_dir*, which should be an existing directory. Returns a new `Path` instance pointing to the moved path.

Added in version 3.14.

`Path.unlink(missing_ok=False)`

이 파일이나 심볼릭 링크를 제거합니다. 경로가 디렉터리를 가리키면, `Path.rmdir()`을 대신 사용하십시오.

*missing_ok*가 거짓(기본값)이면, 경로가 없을 때 `FileNotFoundError`가 발생합니다.

*missing_ok*가 참이면, `FileNotFoundError` 예외는 무시됩니다 (POSIX `rm -f` 명령과 같은 동작).

버전 3.8에서 변경: *missing_ok* 매개 변수가 추가되었습니다.

`Path.rmdir()`

이 디렉터리를 제거합니다. 디렉터리는 비어 있어야 합니다.

Permissions and ownership

`Path.owner(*, follow_symlinks=True)`

Return the name of the user owning the file. `KeyError` is raised if the file' s user identifier (UID) isn' t found in the system database.

This method normally follows symlinks; to get the owner of the symlink, add the argument `follow_symlinks=False`.

버전 3.13에서 변경: Raises `UnsupportedOperation` if the `pwd` module is not available. In earlier versions, `NotImplementedError` was raised.

버전 3.13에서 변경: The *follow_symlinks* parameter was added.

`Path.group(*, follow_symlinks=True)`

Return the name of the group owning the file. `KeyError` is raised if the file' s group identifier (GID) isn' t found in the system database.

This method normally follows symlinks; to get the group of the symlink, add the argument `follow_symlinks=False`.

버전 3.13에서 변경: Raises `UnsupportedOperation` if the `grp` module is not available. In earlier versions, `NotImplementedError` was raised.

버전 3.13에서 변경: The *follow_symlinks* parameter was added.

`Path.chmod(mode, *, follow_symlinks=True)`

Change the file mode and permissions, like `os.chmod()`.

This method normally follows symlinks. Some Unix flavours support changing permissions on the symlink itself; on these platforms you may add the argument `follow_symlinks=False`, or use `lchmod()`.

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

버전 3.10에서 변경: The *follow_symlinks* parameter was added.

`Path.lchmod(mode)`

`Path.chmod()`와 비슷하지만, 경로가 심볼릭 링크를 가리키면, 대상이 아닌 심볼릭 링크의 모드가 변경됩니다.

11.1.5 Pattern language

The following wildcards are supported in patterns for `full_match()`, `glob()` and `rglob()`:

`**` (entire segment)
Matches any number of file or directory segments, including zero.

`*` (entire segment)
Matches one file or directory segment.

`*` (part of a segment)
Matches any number of non-separator characters, including zero.

`?`
Matches one non-separator character.

`[seq]`
Matches one character in *seq*, where *seq* is a sequence of characters. Range expressions are supported; for example, `[a-z]` matches any lowercase ASCII letter. Multiple ranges can be combined: `[a-zA-Z0-9_]` matches any ASCII letter, digit, or underscore.

`[!seq]`
Matches one character not in *seq*, where *seq* follows the same rules as above.

For a literal match, wrap the meta-characters in brackets. For example, `"[?]"` matches the character `"?"`.

The "`**`" wildcard enables recursive globbing. A few examples:

Pattern	Meaning
"`**/*`"	Any path with at least one segment.
"`**/*.py`"	Any path with a final segment ending "`.py`".
"`assets/**`"	Any path starting with "`assets/`".
"`assets/**/*`"	Any path starting with "`assets/`", excluding "`assets/`" itself.

> **참고**
>
> Globbing with the "**" wildcard visits every directory in the tree. Large directory trees may take a long time to search.

버전 3.13에서 변경: Globbing with a pattern that ends with "**" returns both files and directories. In previous versions, only directories were returned.

In `Path.glob()` and `rglob()`, a trailing slash may be added to the pattern to match only directories.

버전 3.11에서 변경: Globbing with a pattern that ends with a pathname components separator (`sep` or `altsep`) returns only directories.

11.1.6 Comparison to the `glob` module

The patterns accepted and results generated by `Path.glob()` and `Path.rglob()` differ slightly from those by the `glob` module:

1. Files beginning with a dot are not special in pathlib. This is like passing `include_hidden=True` to `glob.glob()`.
2. "**" pattern components are always recursive in pathlib. This is like passing `recursive=True` to `glob.glob()`.
3. "**" pattern components do not follow symlinks by default in pathlib. This behaviour has no equivalent in `glob.glob()`, but you can pass `recurse_symlinks=True` to `Path.glob()` for compatible behaviour.
4. Like all `PurePath` and `Path` objects, the values returned from `Path.glob()` and `Path.rglob()` don' t include trailing slashes.
5. The values returned from pathlib' s `path.glob()` and `path.rglob()` include the *path* as a prefix, unlike the results of `glob.glob(root_dir=path)`.
6. The values returned from pathlib' s `path.glob()` and `path.rglob()` may include *path* itself, for example when globbing "**", whereas the results of `glob.glob(root_dir=path)` never include an empty string that would correspond to *path*.

11.1.7 Comparison to the `os` and `os.path` modules

pathlib implements path operations using `PurePath` and `Path` objects, and so it' s said to be *object-oriented*. On the other hand, the `os` and `os.path` modules supply functions that work with low-level `str` and `bytes` objects, which is a more *procedural* approach. Some users consider the object-oriented style to be more readable.

Many functions in `os` and `os.path` support `bytes` paths and *paths relative to directory descriptors*. These features aren' t available in pathlib.

Python' s `str` and `bytes` types, and portions of the `os` and `os.path` modules, are written in C and are very speedy. pathlib is written in pure Python and is often slower, but rarely slow enough to matter.

pathlib' s path normalization is slightly more opinionated and consistent than `os.path`. For example, whereas `os.path.abspath()` eliminates "`..`" segments from a path, which may change its meaning if symlinks are involved, `Path.absolute()` preserves these segments for greater safety.

pathlib' s path normalization may render it unsuitable for some applications:

1. pathlib normalizes `Path("my_folder/")` to `Path("my_folder")`, which changes a path' s meaning when supplied to various operating system APIs and command-line utilities. Specifically, the absence of a trailing separator may allow the path to be resolved as either a file or directory, rather than a directory only.
2. pathlib normalizes `Path("./my_program")` to `Path("my_program")`, which changes a path' s meaning when used as an executable search path, such as in a shell or when spawning a child process. Specifically, the absence of a separator in the path may force it to be looked up in `PATH` rather than the current directory.

As a consequence of these differences, pathlib is not a drop-in replacement for `os.path`.

Corresponding tools

아래는 다양한 `os` 함수를 해당 `PurePath`/`Path` 대응 물에 매핑하는 표입니다.

`os` and `os.path`	`pathlib`
`os.path.dirname()`	`PurePath.parent`
`os.path.basename()`	`PurePath.name`
`os.path.splitext()`	`PurePath.stem`, `PurePath.suffix`
`os.path.join()`	`PurePath.joinpath()`
`os.path.isabs()`	`PurePath.is_absolute()`
`os.path.relpath()`	`PurePath.relative_to()`[1]
`os.path.expanduser()`	`Path.expanduser()`[2]
`os.path.realpath()`	`Path.resolve()`
`os.path.abspath()`	`Path.absolute()`[3]
`os.path.exists()`	`Path.exists()`
`os.path.isfile()`	`Path.is_file()`
`os.path.isdir()`	`Path.is_dir()`
`os.path.islink()`	`Path.is_symlink()`
`os.path.isjunction()`	`Path.is_junction()`
`os.path.ismount()`	`Path.is_mount()`
`os.path.samefile()`	`Path.samefile()`
`os.getcwd()`	`Path.cwd()`
`os.stat()`	`Path.stat()`
`os.lstat()`	`Path.lstat()`
`os.listdir()`	`Path.iterdir()`
`os.walk()`	`Path.walk()`[4]
`os.mkdir()`, `os.makedirs()`	`Path.mkdir()`
`os.link()`	`Path.hardlink_to()`
`os.symlink()`	`Path.symlink_to()`
`os.readlink()`	`Path.readlink()`
`os.rename()`	`Path.rename()`
`os.replace()`	`Path.replace()`
`os.remove()`, `os.unlink()`	`Path.unlink()`
`os.rmdir()`	`Path.rmdir()`
`os.chmod()`	`Path.chmod()`
`os.lchmod()`	`Path.lchmod()`

11.1.8 Protocols

The `pathlib.types` module provides types for static type checking.

Added in version 3.14.

class pathlib.types.**PathInfo**

A `typing.Protocol` describing the `Path.info` attribute. Implementations may return cached results from their methods.

exists(*, *follow_symlinks=True*)

Return `True` if the path is an existing file or directory, or any other kind of file; return `False` if the path doesn' t exist.

[1] `os.path.relpath()` calls `abspath()` to make paths absolute and remove "`..`" parts, whereas `PurePath.relative_to()` is a lexical operation that raises `ValueError` when its inputs' anchors differ (e.g. if one path is absolute and the other relative.)

[2] `os.path.expanduser()` returns the path unchanged if the home directory can' t be resolved, whereas `Path.expanduser()` raises `RuntimeError`.

[3] `os.path.abspath()` removes "`..`" components without resolving symlinks, which may change the meaning of the path, whereas `Path.absolute()` leaves any "`..`" components in the path.

[4] `os.walk()` always follows symlinks when categorizing paths into *dirnames* and *filenames*, whereas `Path.walk()` categorizes all symlinks into *filenames* when *follow_symlinks* is false (the default.)

If *follow_symlinks* is `False`, return `True` for symlinks without checking if their targets exist.

is_dir(*, *follow_symlinks=True*)

Return `True` if the path is a directory, or a symbolic link pointing to a directory; return `False` if the path is (or points to) any other kind of file, or if it doesn' t exist.

If *follow_symlinks* is `False`, return `True` only if the path is a directory (without following symlinks); return `False` if the path is any other kind of file, or if it doesn' t exist.

is_file(*, *follow_symlinks=True*)

Return `True` if the path is a file, or a symbolic link pointing to a file; return `False` if the path is (or points to) a directory or other non-file, or if it doesn' t exist.

If *follow_symlinks* is `False`, return `True` only if the path is a file (without following symlinks); return `False` if the path is a directory or other non-file, or if it doesn' t exist.

is_symlink()

Return `True` if the path is a symbolic link (even if broken); return `False` if the path is a directory or any kind of file, or if it doesn' t exist.

11.2 os.path — 일반적인 경로명 조작

소스 코드: Lib/genericpath.py, Lib/posixpath.py (POSIX용) 및 Lib/ntpath.py (윈도우용)

이 모듈은 경로명에 유용한 함수를 구현합니다. 파일을 읽거나 쓰려면 *open()*을 참조하고, 파일 시스템에 액세스하려면 *os* 모듈을 참조하십시오. 경로 매개 변수는 문자열, 바이트열 또는 *os.PathLike* 프로토콜을 구현하는 모든 객체로 전달할 수 있습니다.

유닉스 셸과 달리, 파이썬은 어떤 자동 경로 확장도 수행하지 않습니다. *expanduser()*와 *expandvars()*와 같은 함수는 응용 프로그램이 셸과 같은 경로 확장을 원할 때 명시적으로 호출할 수 있습니다. (*glob* 모듈도 참조하십시오.)

더 보기

pathlib 모듈은 고수준의 경로 객체를 제공합니다.

참고

이 모든 함수는 매개 변수가 모두 바이트열 객체이거나 모두 문자열 객체인 것만 허락합니다. 경로나 파일 이름이 반환되면, 결과는 같은 형의 객체입니다.

참고

운영 체제마다 경로 이름 규칙이 다르기 때문에, 표준 라이브러리에 이 모듈의 여러 버전이 있습니다. *os.path* 모듈은 항상 파이썬이 실행 중인 운영 체제에 적합한 경로 모듈이고, 따라서 지역 경로에 사용할 수 있습니다. 그러나, 항상 다른 형식 중 하나인 경로를 조작하려면 개별 모듈을 임포트 해서 사용할 수도 있습니다. 그들은 모두 같은 인터페이스를 가지고 있습니다:

- `posixpath` for UNIX-style paths
- `ntpath` for Windows paths

버전 3.8에서 변경: *exists()*, *lexists()*, *isdir()*, *isfile()*, *islink()* 및 *ismount()*는 이제 OS 수준에서 표현할 수 없는 문자나 바이트를 포함하는 경로에 대해 예외를 발생시키는 대신 `False`를 반환합니다.

os.path.**abspath**(*path*)

경로명 *path*의 정규화된 절대 버전을 반환합니다. 대부분의 플랫폼에서, 이는 다음과 같이 *normpath()* 함수를 호출하는 것과 동등합니다: `normpath(join(os.getcwd(), path))`.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**basename**(*path*, /)

경로명 *path*의 기본 이름을 반환합니다. 이것은 *path*를 함수 *split()*에 전달하여 반환된 쌍의 두 번째 요소입니다. 이 함수의 결과는 유닉스 **basename** 프로그램과 다름에 유의하십시오; `'/foo/bar/'`에 대해 **basename**은 `'bar'`를 반환하고, *basename()* 함수는 빈 문자열(`''`)을 반환합니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**commonpath**(*paths*)

이터러블 *paths*에 있는 각 경로명의 가장 긴 공통 하위 경로(sub-path)를 반환합니다. *paths*에 절대 경로명과 상대 경로명이 모두 있거나, *paths*가 다른 드라이브에 있거나, *paths*가 비어 있으면 *ValueError*를 발생시킵니다. *commonprefix()*와 달리, 유효한 경로를 반환합니다.

Added in version 3.5.

버전 3.6에서 변경: 경로류 객체의 시퀀스를 받아들입니다.

버전 3.13에서 변경: Any iterable can now be passed, rather than just sequences.

os.path.**commonprefix**(*list*, /)

*list*에 있는 모든 경로의 접두사인 가장 긴 경로 접두사(문자 단위로 취합니다)를 반환합니다. *list*가 비어 있으면, 빈 문자열(`''`)을 반환합니다.

> **참고**
>
> 이 함수는 한 번에 한 문자씩 다루기 때문에 유효하지 않은 경로를 반환할 수 있습니다. 유효한 경로를 얻으려면, *commonpath()*를 참조하십시오.
>
> ```
> >>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
> '/usr/l'
>
> >>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
> '/usr'
> ```

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**dirname**(*path*, /)

경로명 *path*의 디렉터리 이름을 반환합니다. 이것은 *path*를 함수 *split()*에 전달하여 반환된 쌍의 첫 번째 요소입니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**exists**(*path*)

*path*가 기존 경로나 열린 파일 기술자를 참조하면 `True`를 반환합니다. 깨진 심볼릭 링크에 대해서는 `False`를 반환합니다. 일부 플랫폼에서, *path*가 물리적으로 존재하더라도, 요청된 파일에 대해 *os.stat()*을 실행할 권한이 없으면 이 함수는 `False`를 반환할 수 있습니다.

버전 3.3에서 변경: *path*는 이제 정수가 될 수 있습니다: 열린 파일 기술자이면 `True`가 반환되고, 그렇지 않으면 `False`가 반환됩니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**lexists**(*path*)

*path*가 깨진 심볼릭 링크를 포함하여 기존 경로를 참조하면 `True`를 반환합니다. *os.lstat()*이 없는 플랫폼에서 *exists()*와 동등합니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**expanduser**(*path*)

유닉스와 윈도우에서, ~나 ~user의 초기 구성 요소가 해당 사용자의 홈 디렉터리로 치환된 인자를 반환합니다.

유닉스에서, 초기 ~는 환경 변수 HOME이 설정되어 있다면 그것으로 치환됩니다; 그렇지 않으면 현재 사용자의 홈 디렉터리가 내장 모듈 *pwd*를 통해 비밀번호 디렉터리에서 조회됩니다. 초기 ~user는 비밀번호 디렉터리에서 직접 조회됩니다.

윈도우에서, USERPROFILE이 설정되었으면 이것이 사용됩니다, 그렇지 않으면 HOMEPATH와 HOMEDRIVE의 조합이 사용됩니다. 초기 ~user는 현재 사용자의 홈 디렉터리의 마지막 디렉터리 구성 요소가 USERNAME과 일치하는지 확인하고, 일치하면 교체하는 방식으로 처리됩니다.

확장이 실패하거나 경로가 물결표로 시작하지 않으면, 경로는 변경되지 않은 상태로 반환됩니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.8에서 변경: 더는 윈도우에서 HOME을 사용하지 않습니다.

os.path.**expandvars**(*path*)

환경 변수로 확장된 인자를 반환합니다. $name이나 ${name} 형식의 부분 문자열이 환경 변수 *name*의 값으로 치환됩니다. 잘못된 변수 이름과 존재하지 않는 변수에 대한 참조는 변경되지 않고 남습니다.

윈도우에서, $name과 ${name} 외에 %name% 확장이 지원됩니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**getatime**(*path*, /)

*path*의 마지막 액세스 시간을 반환합니다. 반환 값은 에포크(epoch) 이후 초 수를 나타내는 부동 소수점 숫자입니다 (*time* 모듈을 참조하십시오). 파일이 없거나 액세스할 수 없으면 *OSError*를 발생시킵니다.

os.path.**getmtime**(*path*, /)

*path*를 마지막으로 수정한 시간을 반환합니다. 반환 값은 에포크(epoch) 이후 초 수를 나타내는 부동 소수점 숫자입니다 (*time* 모듈을 참조하십시오). 파일이 없거나 액세스할 수 없으면 *OSError*를 발생시킵니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**getctime**(*path*, /)

시스템의 ctime을 반환하는데, 일부 시스템(가령 유닉스)에서는 마지막 메타 데이터 변경 시간이고, 다른 시스템(가령 윈도우)에서는 *path* 생성 시간입니다. 반환 값은 에포크(epoch) 이후 초 수를 나타내는 부동 소수점 숫자입니다 (*time* 모듈을 참조하십시오). 파일이 없거나 액세스할 수 없으면 *OSError*를 발생시킵니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**getsize**(*path*, /)

*path*의 크기를 바이트 단위로 반환합니다. 파일이 없거나 액세스할 수 없으면 *OSError*를 발생시킵니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**isabs**(*path*, /)

*path*가 절대 경로명이면 True를 반환합니다. 유닉스에서는 슬래시로 시작하고, 윈도우에서는 두 개의 (역)슬래시나 드라이브 문자, 콜론 및 (역)슬래시를 합친 것으로 시작함을 의미합니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.13에서 변경: On Windows, returns False if the given path starts with exactly one (back)slash.

os.path.**isfile**(*path*)

*path*가 존재하는 일반 파일이면 True를 반환합니다. 이것은 심볼릭 링크를 따르므로, 같은 경로에 대해 *islink()*와 *isfile()*이 모두 참일 수 있습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**isdir**(*path, /*)

*path*가 존재하는 디렉터리이면 `True`를 반환합니다. 이것은 심볼릭 링크를 따르므로, 같은 경로에 대해 `islink()`와 `isdir()`이 모두 참일 수 있습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**isjunction**(*path*)

*path*가 정션(junction)인 존재하는 디렉터리 항목을 가리키면 `True`를 반환합니다. 현재 플랫폼에서 정션을 지원하지 않으면 항상 `False`를 반환합니다.

Added in version 3.12.

os.path.**islink**(*path*)

*path*가 심볼릭 링크인 존재하는 디렉터리 항목을 가리키면 `True`를 반환합니다. 파이썬 런타임에서 심볼릭 링크를 지원하지 않으면 항상 `False`입니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**ismount**(*path*)

Return `True` if pathname *path* is a *mount point*: a point in a file system where a different file system has been mounted. On POSIX, the function checks whether *path*' s parent, `path/..`, is on a different device than *path*, or whether `path/..` and *path* point to the same i-node on the same device — this should detect mount points for all Unix and POSIX variants. It is not able to reliably detect bind mounts on the same filesystem. On Linux systems, it will always return `True` for btrfs subvolumes, even if they aren' t mount points. On Windows, a drive letter root and a share UNC are always mount points, and for any other path `GetVolumePathName` is called to see if it is different from the input path.

버전 3.4에서 변경: 윈도우에서 비 루트 마운트 지점 감지 지원을 추가했습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**isdevdrive**(*path*)

Return `True` if pathname *path* is located on a Windows Dev Drive. A Dev Drive is optimized for developer scenarios, and offers faster performance for reading and writing files. It is recommended for use for source code, temporary build directories, package caches, and other IO-intensive operations.

May raise an error for an invalid path, for example, one without a recognizable drive, but returns `False` on platforms that do not support Dev Drives. See the Windows documentation for information on enabling and creating Dev Drives.

Added in version 3.12.

버전 3.13에서 변경: The function is now available on all platforms, and will always return `False` on those that have no support for Dev Drives

os.path.**isreserved**(*path*)

Return `True` if *path* is a reserved pathname on the current system.

On Windows, reserved filenames include those that end with a space or dot; those that contain colons (i.e. file streams such as "name:stream"), wildcard characters (i.e. `'*?"<>'`), pipe, or ASCII control characters; as well as DOS device names such as "NUL", "CON", "CONIN$", "CONOUT$", "AUX", "PRN", "COM1", and "LPT1".

> **참고**
>
> This function approximates rules for reserved paths on most Windows systems. These rules change over time in various Windows releases. This function may be updated in future Python releases as changes to the rules become broadly available.

가용성: Windows.

Added in version 3.13.

os.path.**join**(*path, /, *paths*)

하나 이상의 경로 세그먼트를 지능적으로 결합합니다. 반환 값은 마지막을 제외한 *path*와 **paths*의 모든 멤버에 대해 비어 있지 않은 각 부분 다음에 정확히 하나의 디렉터리 구분자가 오도록 이어붙인 것입니다. 즉 마지막 부분이 비어 있거나 구분자로 끝날 때만 결과가 구분자로 끝남을 의미합니다. 세그먼트가 절대 경로이면 (윈도우에서는 드라이브와 루트가 모두 필요합니다), 그 앞의 모든 세그먼트가 무시되고 절대 경로 세그먼트에서부터 결합이 계속됩니다.

윈도우에서 루트 경로 세그먼트(예를 들어 `r'\foo'`)를 만날 때 드라이브가 재설정되지 않습니다. 세그먼트가 다른 드라이브이거나 절대 경로면, 이전의 모든세그먼트는 무시되고 드라이브를 재설정합니다. 각 드라이브에 현재 디렉터리가 있기 때문에, `os.path.join("c:", "foo")`는 `c:\foo`가 아니라 드라이브 `c:`의 현재 디렉터리에 상대적인 경로를 나타냅니다 (`c:foo`).

버전 3.6에서 변경: *path*와 *paths*에 대해 경로류 객체를 받아들입니다.

os.path.**normcase**(*path, /*)

경로명의 대소 문자를 정규화합니다. 윈도우에서는, 경로명의 모든 문자를 소문자로 변환하고, 슬래시도 역 슬래시로 변환합니다. 다른 운영 체제에서는, 경로를 변경하지 않고 반환합니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**normpath**(*path*)

중복된 구분자와 상위 수준 참조를 접어 경로명을 정규화합니다. 그래서 `A//B`, `A/B/`, `A/./B` 및 `A/foo/../B`가 모두 `A/B`가 됩니다. 이 문자열 조작은 심볼릭 링크가 포함된 경로의 의미를 변경할 수 있습니다. 윈도우에서는, 슬래시를 역 슬래시로 변환합니다. 대소 문자를 정규화하려면, *normcase()*를 사용하십시오.

> **참고**
>
> On POSIX systems, in accordance with IEEE Std 1003.1 2013 Edition; 4.13 Pathname Resolution, if a pathname begins with exactly two slashes, the first component following the leading characters may be interpreted in an implementation-defined manner, although more than two leading characters shall be treated as a single character.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**realpath**(*path, /, *, strict=False*)

(운영 체제에서 지원한다면) 경로에서 발견된 심볼릭 링크를 제거해서 지정된 파일명의 규범적 (canonical) 경로를 반환합니다. 윈도우에서, 이 함수는 또한 `C:\\PROGRA~1`와 같은 MS-DOS (8.3 이라고도 합니다) 스타일 이름을 `C:\\Program Files`로 해석합니다.

By default, the path is evaluated up to the first component that does not exist, is a symlink loop, or whose evaluation raises *OSError*. All such components are appended unchanged to the existing part of the path.

Some errors that are handled this way include "access denied", "not a directory", or "bad argument to internal function". Thus, the resulting path may be missing or inaccessible, may still contain links or loops, and may traverse non-directories.

This behavior can be modified by keyword arguments:

If *strict* is `True`, the first error encountered when evaluating the path is re-raised. In particular, *FileNotFoundError* is raised if *path* does not exist, or another *OSError* if it is otherwise inaccessible.

If *strict* is *os.path.ALLOW_MISSING*, errors other than *FileNotFoundError* are re-raised (as with `strict=True`). Thus, the returned path will not contain any symbolic links, but the named file and some of its parent directories may be missing.

> **참고**
>
> This function emulates the operating system' s procedure for making a path canonical, which differs slightly between Windows and UNIX with respect to how links and subsequent path components interact.

> Operating system APIs make paths canonical as needed, so it' s not normally necessary to call this function.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.8에서 변경: 윈도우에서 심볼릭 링크와 정션(junctions)이 이제 해석됩니다.

버전 3.10에서 변경: The *strict* parameter was added.

버전 3.14에서 변경: The `ALLOW_MISSING` value for the *strict* parameter was added.

os.path.**ALLOW_MISSING**

Special value used for the *strict* argument in `realpath()`.

Added in version 3.14.

os.path.**relpath**(*path*, *start=os.curdir*)

현재 디렉터리나 선택적 *start* 디렉터리로부터 *path*로의 상대 파일 경로를 반환합니다. 이것은 경로 계산입니다: *path*나 *start*의 존재나 특성을 확인하기 위해 파일 시스템을 액세스하지 않습니다. 윈도우에서, *path*와 *start*가 다른 드라이브에 있을 때 `ValueError`가 발생합니다.

*start*의 기본값은 `os.curdir`입니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**samefile**(*path1*, *path2*, /)

두 경로명 인자가 같은 파일이나 디렉터리를 가리키면 `True`를 반환합니다. 장치 번호와 i-노드 번호로 결정하며 경로명 중 어느 하나에 대해 `os.stat()` 호출이 실패하면 예외를 발생시킵니다.

버전 3.2에서 변경: 윈도우 지원이 추가되었습니다.

버전 3.4에서 변경: 윈도우는 이제 다른 모든 플랫폼과 같은 구현을 사용합니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**sameopenfile**(*fp1*, *fp2*)

파일 기술자 *fp1*과 *fp2*가 같은 파일을 가리키면 `True`를 반환합니다.

버전 3.2에서 변경: 윈도우 지원이 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**samestat**(*stat1*, *stat2*, /)

stat 튜플 *stat1*과 *stat2*가 같은 파일을 가리키면 `True`를 반환합니다. 이러한 구조는 `os.fstat()`, `os.lstat()` 또는 `os.stat()`에 의해 반환되었을 수 있습니다. 이 함수는 `samefile()`과 `sameopenfile()`에서 사용하는 하부 비교를 구현합니다.

버전 3.4에서 변경: 윈도우 지원이 추가되었습니다.

os.path.**split**(*path*, /)

path 경로명을 `(head, tail)` 쌍으로 분할합니다. 여기서 *tail*은 마지막 경로명 구성 요소이고 *head*는 그 앞에 오는 모든 것입니다. *tail* 부분에는 슬래시가 포함되지 않습니다; *path*가 슬래시로 끝나면, *tail*은 비어 있습니다. *path*에 슬래시가 없으면, *head*는 비어 있습니다. *path*가 비어 있으면, *head*와 *tail*이 모두 비어 있습니다. 후행 슬래시는 루트(하나나 그 이상의 슬래시로만 구성됩니다)가 아니라면 *head*에서 제거됩니다. 모든 경우에, `join(head, tail)`은 *path*와 같은 위치에 대한 경로를 반환합니다(하지만 문자열은 다를 수 있습니다). `dirname()`과 `basename()` 함수도 참조하십시오.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**splitdrive**(*path*, /)

경로명 *path*를 쌍 `(drive, tail)`로 분할합니다. 여기서 *drive*는 마운트 지점이나 빈 문자열입니다. 드라이브 지정을 사용하지 않는 시스템에서 *drive*는 항상 빈 문자열입니다. 모든 경우에, `drive + tail`은 *path*와 같습니다.

윈도우에서는, 경로명을 드라이브/UNC 공유 지점과 상대 경로로 분할합니다.

경로에 드라이브 문자가 포함되면, drive는 콜론까지의 콜론을 포함하는 모든 것을 포함합니다:

```
>>> splitdrive("c:/dir")
("c:", "/dir")
```

경로에 UNC 경로가 포함되면, drive는 호스트 이름과 공유를 포함합니다:

```
>>> splitdrive("//host/computer/dir")
("//host/computer", "/dir")
```

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**splitroot**(*path*, /)

경로명 *path*를 3항 튜플 `(drive, root, tail)`로 분할합니다. 여기서 *drive*는 장치 이름이나 마운트 지점, *root*는 드라이브 뒤에 오는 구분자 문자열, *tail*은 루트 뒤의 모든 것입니다. 이 항목들 모두 빈 문자열일 수 있습니다. 모든 경우에, `drive + root + tail`은 *path*와 같습니다.

On POSIX systems, *drive* is always empty. The *root* may be empty (if *path* is relative), a single forward slash (if *path* is absolute), or two forward slashes (implementation-defined per IEEE Std 1003.1-2017; 4.13 Pathname Resolution.) For example:

```
>>> splitroot('/home/sam')
('', '/', 'home/sam')
>>> splitroot('//home/sam')
('', '//', 'home/sam')
>>> splitroot('///home/sam')
('', '/', '//home/sam')
```

On Windows, *drive* may be empty, a drive-letter name, a UNC share, or a device name. The *root* may be empty, a forward slash, or a backward slash. For example:

```
>>> splitroot('C:/Users/Sam')
('C:', '/', 'Users/Sam')
>>> splitroot('//Server/Share/Users/Sam')
('//Server/Share', '/', 'Users/Sam')
```

Added in version 3.12.

os.path.**splitext**(*path*, /)

경로명 *path*를 `root + ext == path`가 되도록 쌍 `(root, ext)`로 분할하는데, 확장자 *ext*는 비어 있거나 마침표로 시작하고 최대 하나의 마침표를 포함합니다.

If the path contains no extension, *ext* will be `''`:

```
>>> splitext('bar')
('bar', '')
```

If the path contains an extension, then *ext* will be set to this extension, including the leading period. Note that previous periods will be ignored:

```
>>> splitext('foo.bar.exe')
('foo.bar', '.exe')
>>> splitext('/foo/bar.exe')
('/foo/bar', '.exe')
```

Leading periods of the last component of the path are considered to be part of the root:

```
>>> splitext('.cshrc')
('.cshrc', '')
>>> splitext('/foo/....jpg')
('/foo/....jpg', '')
```

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.path.**supports_unicode_filenames**

(파일 시스템에 의해 부과된 제한 내에서) 임의의 유니코드 문자열을 파일 이름으로 사용할 수 있으면 True.

11.3 stat — stat() 결과 해석하기

소스 코드: Lib/stat.py

---

stat 모듈은 *os.stat()*, *os.fstat()* 및 *os.lstat()*의 (이들이 존재한다면) 결과를 해석하기 위한 상수와 함수를 정의합니다. stat(), fstat() 및 lstat() 호출에 대한 자세한 내용은 여러분의 시스템 설명서를 참조하십시오.

버전 3.4에서 변경: stat 모듈은 C 구현으로 지원됩니다.

stat 모듈은 특정 파일 유형을 검사하기 위해 다음 함수를 정의합니다:

stat.**S_ISDIR**(*mode*)

mode가 디렉터리로부터 왔으면 0이 아닌 값을 반환합니다.

stat.**S_ISCHR**(*mode*)

mode가 문자 특수 장치(character special device) 파일로부터 왔으면 0이 아닌 값을 반환합니다.

stat.**S_ISBLK**(*mode*)

mode가 블록 특수 장치(block special device) 파일로부터 왔으면 0이 아닌 값을 반환합니다.

stat.**S_ISREG**(*mode*)

mode가 일반 파일로부터 왔으면 0이 아닌 값을 반환합니다.

stat.**S_ISFIFO**(*mode*)

mode가 FIFO(네임드 파이프)로부터 왔으면 0이 아닌 값을 반환합니다.

stat.**S_ISLNK**(*mode*)

mode가 심볼릭 링크로부터 왔으면 0이 아닌 값을 반환합니다.

stat.**S_ISSOCK**(*mode*)

mode가 소켓으로부터 왔으면 0이 아닌 값을 반환합니다.

stat.**S_ISDOOR**(*mode*)

mode가 door로부터 왔으면 0이 아닌 값을 반환합니다.

Added in version 3.4.

stat.**S_ISPORT**(*mode*)

mode가 이벤트 포트(event port)로부터 왔으면 0이 아닌 값을 반환합니다.

Added in version 3.4.

stat.**S_ISWHT**(*mode*)

mode가 화이트 아웃(whiteout)으로부터 왔으면 0이 아닌 값을 반환합니다.

Added in version 3.4.

파일의 모드(mode)를 보다 일반적으로 조작하기 위한 두 가지 추가 함수가 정의됩니다:

stat.**S_IMODE**(*mode*)

*os.chmod()*로 설정할 수 있는 파일 모드 부분을 반환합니다 —즉, 파일의 권한(permission) 비트, 끈끈한(sticky) 비트, set-group-id 및 set-user-id 비트 (지원하는 시스템에서).

stat.**S_IFMT**(*mode*)

파일 유형을 기술하는 파일 모드 부분을 반환합니다 (위의 S_IS*() 함수에서 사용됩니다).

일반적으로, 파일 유형을 검사하는 데 os.path.is*() 함수를 사용합니다; 이 함수들은 같은 파일에 대해 여러 개의 검사를 수행하고, 검사마다 stat() 시스템 호출 하는 오버헤드를 피하려고 할 때 유용합니다. 또한, 블록과 문자 장치 검사와 같이, *os.path*에서 처리되지 않는 파일에 대한 정보를 확인할 때 유용합니다.

예제:

```
import os, sys
from stat import *

def walktree(top, callback):
    '''top에 뿌리를 둔 디렉터리 트리를 재귀적으로 탐색해 내려가면서,
       각 일반 파일에 대해 콜백 함수를 호출합니다'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.lstat(pathname).st_mode
        if S_ISDIR(mode):
            # 디렉터리입니다, 재귀적으로 내려갑니다
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # 파일입니다, 콜백 함수를 호출합니다.
            callback(pathname)
        else:
            # 알 수 없는 파일 유형입니다, 메시지를 인쇄합니다
            print('Skipping %s' % pathname)

def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

파일의 모드를 사람이 읽을 수 있는 문자열로 변환하기 위한 추가 유틸리티 함수가 제공됩니다:

stat.**filemode**(*mode*)

파일의 mode를 '-rwxrwxrwx' 형식의 문자열로 변환합니다.

Added in version 3.3.

버전 3.4에서 변경: 이 함수는 *S_IFDOOR*, *S_IFPORT* 및 *S_IFWHT*를 지원합니다.

아래의 모든 변수는 단순히 *os.stat()*, *os.fstat()* 또는 *os.lstat()*에 의해 반환된 10-튜플에 대한 기호 인덱스입니다.

stat.**ST_MODE**

아이 노드(inode) 보호 모드.

stat.**ST_INO**

아이 노드(inode) 번호.

stat.**ST_DEV**

아이 노드(inode)가 위치한 장치.

stat.**ST_NLINK**

아이 노드(inode)에 대한 링크 수.

stat.**ST_UID**

소유자의 사용자 id.

stat.**ST_GID**
소유자의 그룹 id.

stat.**ST_SIZE**
일반 파일의 크기(바이트); 일부 특수 파일에서는 대기중인 데이터의 양.

stat.**ST_ATIME**
마지막 액세스 시간.

stat.**ST_MTIME**
마지막 수정 시간.

stat.**ST_CTIME**
운영 체제에서 보고한 "ctime". (유닉스와 같은) 일부 시스템에서는 마지막 메타 데이터 변경 시간이고, (윈도우와 같은) 다른 시스템에서는 생성 시간입니다 (자세한 내용은 플랫폼 설명서를 참조하십시오).

"파일 크기"의 해석은 파일 유형에 따라 달라집니다. 일반 파일에서는 바이트로 표현한 파일의 크기입니다. 대부분의 유닉스(특히 리눅스를 포함하는)의 FIFO와 소켓에서, "크기"는 `os.stat()`, `os.fstat()` 또는 `os.lstat()`를 호출한 시점에 읽기 대기 중인 바이트 수입니다; 이것은 때때로, 특히 비 블로킹으로 연 후에 이러한 특수 파일 중 하나를 폴링할 때 유용할 수 있습니다. 다른 문자와 블록 장치에서 크기 필드의 의미는 하부 시스템 호출의 구현에 따라 더 다양합니다.

아래의 변수는 `ST_MODE` 필드에서 사용되는 플래그를 정의합니다.

첫 번째 플래그 집합을 사용하는 것보다 위의 함수를 사용하는 것이 더 이식성 있습니다:

stat.**S_IFSOCK**
소켓.

stat.**S_IFLNK**
심볼릭 링크.

stat.**S_IFREG**
일반 파일.

stat.**S_IFBLK**
블록 장치.

stat.**S_IFDIR**
디렉터리.

stat.**S_IFCHR**
문자 장치.

stat.**S_IFIFO**
FIFO.

stat.**S_IFDOOR**
Door.

Added in version 3.4.

stat.**S_IFPORT**
이벤트 포트.

Added in version 3.4.

stat.**S_IFWHT**
화이트 아웃(whiteout).

Added in version 3.4.

> **참고**
>
> 플랫폼이 파일 유형을 지원하지 않으면, *S_IFDOOR*, *S_IFPORT* 또는 *S_IFWHT*는 0으로 정의됩니다.

다음 플래그는 *os.chmod()*의 *mode* 인자에서도 사용할 수 있습니다:

stat.**S_ISUID**

Set-user-ID 비트.

stat.**S_ISGID**

Set-group-ID 비트. 이 비트는 몇 가지 특별한 용도로 사용됩니다. 디렉터리에서는 그 디렉터리가 BSD의 의미가 있음을 나타냅니다: 여기에 만들어진 파일은 만드는 프로세스의 유효 그룹 ID가 아니라 디렉터리에서 그룹 ID를 상속받고, *S_ISGID* 비트 설정도 얻습니다. 그룹 실행 비트(*S_IXGRP*)가 설정되지 않은 파일의 경우, set-group-ID 비트는 필수 파일/레코드 잠금을 나타냅니다 (*S_ENFMT*도 참조하십시오).

stat.**S_ISVTX**

끈끈한(sticky) 비트. 이 비트가 디렉터리에 설정되면, 해당 디렉터리의 파일은 파일의 소유자, 디렉터리의 소유자 또는 권한 있는(privileged) 프로세스에 의해서만 이름이 바뀌거나 삭제될 수 있음을 의미합니다.

stat.**S_IRWXU**

파일 소유자 권한(permission) 마스크.

stat.**S_IRUSR**

소유자에게 읽기 권한이 있습니다.

stat.**S_IWUSR**

소유자에게 쓰기 권한이 있습니다.

stat.**S_IXUSR**

소유자에게 실행 권한이 있습니다.

stat.**S_IRWXG**

그룹 권한 마스크.

stat.**S_IRGRP**

그룹에 읽기 권한이 있습니다.

stat.**S_IWGRP**

그룹에 쓰기 권한이 있습니다.

stat.**S_IXGRP**

그룹에 실행 권한이 있습니다.

stat.**S_IRWXO**

다른 사용자(그룹에 없는)에 대한 권한 마스크.

stat.**S_IROTH**

다른 사용자에게 읽기 권한이 있습니다.

stat.**S_IWOTH**

다른 사용자에게 쓰기 권한이 있습니다.

stat.**S_IXOTH**

다른 사용자에게 실행 권한이 있습니다.

stat.**S_ENFMT**

System V 파일 잠금 강제. 이 플래그는 *S_ISGID*와 공유됩니다: 파일/레코드 잠금이 그룹 실행 비트(*S_IXGRP*)가 설정되지 않은 파일에 적용됩니다.

stat.**S_IREAD**

*S_IRUSR*에 대한 유닉스 V7 동의어.

stat.**S_IWRITE**

*S_IWUSR*에 대한 유닉스 V7 동의어.

stat.**S_IEXEC**

*S_IXUSR*에 대한 유닉스 V7 동의어.

다음 플래그는 *os.chflags()*의 *flags* 인자에서 사용될 수 있습니다:

stat.**UF_SETTABLE**

All user settable flags.

Added in version 3.13.

stat.**UF_NODUMP**

파일을 덤프하지 마십시오.

stat.**UF_IMMUTABLE**

파일을 변경할 수 없습니다.

stat.**UF_APPEND**

파일은 덧붙이기만 할 수 있습니다.

stat.**UF_OPAQUE**

디렉터리는 유니언 스택(union stack)을 통해 볼 때 불투명합니다.

stat.**UF_NOUNLINK**

파일의 이름을 변경하거나 삭제할 수 없습니다.

stat.**UF_COMPRESSED**

파일은 압축되어 저장됩니다 (맥 OS 10.6+).

stat.**UF_TRACKED**

Used for handling document IDs (macOS)

Added in version 3.13.

stat.**UF_DATAVAULT**

The file needs an entitlement for reading or writing (macOS 10.13+)

Added in version 3.13.

stat.**UF_HIDDEN**

파일을 GUI에 표시하면 안 됩니다 (맥 OS 10.5+).

stat.**SF_SETTABLE**

All super-user changeable flags

Added in version 3.13.

stat.**SF_SUPPORTED**

All super-user supported flags

가용성: macOS

Added in version 3.13.

stat.**SF_SYNTHETIC**

All super-user read-only synthetic flags

가용성: macOS

Added in version 3.13.

stat.**SF_ARCHIVED**
파일을 보관(archive)할 수 있습니다.

stat.**SF_IMMUTABLE**
파일을 변경할 수 없습니다.

stat.**SF_APPEND**
파일은 덧붙이기만 할 수 있습니다.

stat.**SF_RESTRICTED**
The file needs an entitlement to write to (macOS 10.13+)

Added in version 3.13.

stat.**SF_NOUNLINK**
파일의 이름을 변경하거나 삭제할 수 없습니다.

stat.**SF_SNAPSHOT**
파일은 스냅숏(snapshot) 파일입니다.

stat.**SF_FIRMLINK**
The file is a firmlink (macOS 10.15+)

Added in version 3.13.

stat.**SF_DATALESS**
The file is a dataless object (macOS 10.15+)

Added in version 3.13.

자세한 정보는 *BSD나 맥 OS 시스템 매뉴얼 페이지 *chflags(2)*를 참조하십시오.

윈도우에서 *os.stat()*에 의해 반환된 `st_file_attributes` 멤버의 비트를 검사할 때 다음 파일 어트리뷰트 상수를 사용할 수 있습니다. 이러한 상수의 의미에 대한 자세한 내용은 Windows API documentation을 참조하십시오.

stat.**FILE_ATTRIBUTE_ARCHIVE**

stat.**FILE_ATTRIBUTE_COMPRESSED**

stat.**FILE_ATTRIBUTE_DEVICE**

stat.**FILE_ATTRIBUTE_DIRECTORY**

stat.**FILE_ATTRIBUTE_ENCRYPTED**

stat.**FILE_ATTRIBUTE_HIDDEN**

stat.**FILE_ATTRIBUTE_INTEGRITY_STREAM**

stat.**FILE_ATTRIBUTE_NORMAL**

stat.**FILE_ATTRIBUTE_NOT_CONTENT_INDEXED**

stat.**FILE_ATTRIBUTE_NO_SCRUB_DATA**

stat.**FILE_ATTRIBUTE_OFFLINE**

stat.**FILE_ATTRIBUTE_READONLY**

stat.**FILE_ATTRIBUTE_REPARSE_POINT**

stat.**FILE_ATTRIBUTE_SPARSE_FILE**

stat.**FILE_ATTRIBUTE_SYSTEM**

stat.**FILE_ATTRIBUTE_TEMPORARY**

stat.**FILE_ATTRIBUTE_VIRTUAL**
Added in version 3.5.

윈도우에서, *os.lstat()*이 반환한 `st_reparse_tag` 멤버와 비교하기 위해 다음 상수를 사용할 수 있습니다. 이것들은 잘 알려진 상수이지만, 완전한 목록은 아닙니다.

stat.**IO_REPARSE_TAG_SYMLINK**

stat.**IO_REPARSE_TAG_MOUNT_POINT**

stat.**IO_REPARSE_TAG_APPEXECLINK**

Added in version 3.8.

11.4 filecmp — 파일과 디렉터리 비교

소스 코드: Lib/filecmp.py

---

filecmp 모듈은 다양한 선택적 시간/정확도 절충을 통해 파일과 디렉터리를 비교하는 함수를 정의합니다. 파일 비교에 대해서는, *difflib* 모듈을 참조하십시오.

filecmp 모듈은 다음 함수를 정의합니다:

filecmp.**cmp**(*f1*, *f2*, *shallow=True*)

*f1*와 *f2*로 이름이 지정된 파일을 비교하여, 같아 보이면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

*shallow*가 참이고 두 파일의 *os.stat()* 서명(파일 유형, 크기, 수정 시간)이 동일하면, 파일을 같다고 취급합니다.

그렇지 않으면, 파일의 크기나 내용이 다를 때 다르다고 취급합니다.

이 함수는 외부 프로그램을 호출하지 않으므로 이식성과 효율성을 제공합니다.

이 함수는 과거 비교와 결과에 대해 캐시를 사용합니다. 파일에 대한 *os.stat()* 정보가 변경되면 캐시 항목이 무효화 됩니다. 전체 캐시는 *clear_cache()*를 사용하여 지울 수 있습니다.

filecmp.**cmpfiles**(*dir1*, *dir2*, *common*, *shallow=True*)

두 디렉터리 *dir1* 과 *dir2*에 있는 이름이 *common*으로 지정된 파일들을 비교합니다.

파일 이름의 세 가지 리스트를 반환합니다: *match*, *mismatch*, *errors*. *match*는 일치하는 파일 리스트를 포함하고, *mismatch*는 일치하지 않는 파일의 이름을 포함하며, *errors*는 비교할 수 없는 파일의 이름을 나열합니다. 파일이 디렉터리 중 하나에 없거나, 사용자가 읽을 수 있는 권한이 없거나, 다른 이유로 인해 비교를 수행할 수 없으면 파일은 *errors*에 나열됩니다.

shallow 매개 변수는 *filecmp.cmp()*와 같은 의미와 기본값을 가집니다.

예를 들어, `cmpfiles('a', 'b', ['c', 'd/e'])`는 `a/c`와 `b/c`, `a/d/e`와 `b/d/e`를 비교합니다. `'c'`와 `'d/e'`는 각각 반환된 세 개의 리스트 중 하나에 포함됩니다.

filecmp.**clear_cache**()

filecmp 캐시를 지웁니다. 파일이 수정된 후 너무 빨리 비교되어 하부 파일 시스템의 mtime 해상도 내에 있을 때 유용합니다.

Added in version 3.4.

11.4.1 dircmp 클래스

class filecmp.**dircmp**(*a*, *b*, *ignore=None*, *hide=None*, **, shallow=True*)

*a*와 *b* 디렉터리를 비교하기 위한, 새로운 디렉터리 비교 객체를 만듭니다. *ignore*는 무시할 이름 리스트며, 기본값은 *filecmp.DEFAULT_IGNORES*입니다. *hide*는 숨길 이름 리스트며 기본값은 `[os.curdir, os.pardir]`입니다.

dircmp 클래스는 (*shallow* 매개 변수를 지원하지만) 기본적으로는 *filecmp.cmp()*에서 설명하는 얕은(*shallow*) 비교를 통해 파일을 비교합니다.

버전 3.13에서 변경: *shallow* 매개 변수를 추가했습니다.

dircmp 클래스는 다음 메서드를 제공합니다:

report()

*a*와 *b* 사이의 비교를 (*sys.stdout*로) 인쇄합니다.

report_partial_closure()

*a*와 *b* 및 공통 직접 하위 디렉터리 사이의 비교를 인쇄합니다.

report_full_closure()

*a*와 *b* 및 공통 하위 디렉터리 (재귀적으로) 사이의 비교를 인쇄합니다.

dircmp 클래스는 비교되는 디렉터리 트리에 대한 다양한 정보 비트를 얻는 데 사용될 수 있는 여러 가지 흥미로운 어트리뷰트를 제공합니다.

`__getattr__()` 훅을 통해, 모든 어트리뷰트가 느긋하게(lazily) 계산되므로, 계산하기가 가벼운 어트리뷰트만 사용하면 속도가 저하되지 않습니다.

left

디렉터리 *a*.

right

디렉터리 *b*.

left_list

hide 와 *ignore*로 필터링 된, *a*의 파일과 하위 디렉터리.

right_list

hide 와 *ignore*로 필터링 된, *b*의 파일과 하위 디렉터리.

common

a 와 *b*의 공통 파일과 하위 디렉터리.

left_only

*a*에만 있는 파일과 하위 디렉터리.

right_only

*b*에만 있는 파일과 하위 디렉터리.

common_dirs

a 및 *b*의 공통 하위 디렉터리.

common_files

a 와 *b*의 공통 파일.

common_funny

*a*와 *b*의 공통 이름으로, 디렉터리 간에 유형이 다르거나, *os.stat()*가 에러를 보고하는 이름.

same_files

a 와 *b*에 모두 있고, 클래스의 파일 비교 연산자를 사용할 때 같은 파일.

diff_files

a 및 *b*에 모두 있고, 클래스의 파일 비교 연산자를 사용할 때 내용이 다른 파일.

funny_files

a 및 *b*에 모두 있지만, 비교할 수 없는 파일.

subdirs

*common_dirs*의 이름을 *dircmp* 인스턴스로 (또는 이 인스턴스가 *dircmp*의 서브 클래스인 MyDirCmp 형이면 MyDirCmp 인스턴스로) 매핑하는 딕셔너리.

버전 3.10에서 변경: Previously entries were always *dircmp* instances. Now entries are the same type as *self*, if *self* is a subclass of *dircmp*.

filecmp.**DEFAULT_IGNORES**

Added in version 3.4.

*dircmp*에 의해 기본적으로 무시되는 디렉터리 리스트.

다음은 이름이 같지만, 내용이 다른 파일을 표시하기 위해, `subdirs` 어트리뷰트로 두 개의 디렉터리를 재귀적으로 검색하는 간단한 예제입니다:

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print("diff_file %s found in %s and %s" % (name, dcmp.left,
...               dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

11.5 tempfile — 임시 파일과 디렉터리 생성

소스 코드: Lib/tempfile.py

이 모듈은 임시 파일과 디렉터리를 만듭니다. 지원되는 모든 플랫폼에서 작동합니다. *TemporaryFile*, *NamedTemporaryFile*, *TemporaryDirectory* 및 *SpooledTemporaryFile*은 자동 정리를 제공하고 컨텍스트 관리자로 사용할 수 있는 고수준 인터페이스입니다. *mkstemp()*와 *mkdtemp()*는 수동 정리가 필요한 저수준 함수입니다.

사용자가 호출할 수 있는 모든 함수와 생성자는 임시 파일과 디렉터리의 위치와 이름을 직접 제어할 수 있도록 하는 추가 인자를 취합니다. 이 모듈에서 사용하는 파일 이름에는 무작위 문자의 문자열이 포함되어있어 공유 임시 디렉터리에서 해당 파일을 안전하게 만들 수 있도록 합니다. 이전 버전과의 호환성을 유지하기 위해, 인자 순서는 다소 이상합니다; 명확성을 위해 키워드 인자를 사용하는 것이 좋습니다.

이 모듈은 다음과 같은 사용자 호출 가능 항목을 정의합니다:

tempfile.**TemporaryFile**(*mode='w+b', buffering=-1, encoding=None, newline=None, suffix=None, prefix=None, dir=None, *, errors=None*)

임시 저장 영역으로 사용할 수 있는 파일류 객체를 반환합니다. *mkstemp()*와 같은 규칙을 사용하여 파일이 안전하게 만들어집니다. 닫히는 즉시 삭제됩니다 (객체가 가비지 수집될 때 묵시적인 닫기를 포함합니다). 유닉스에서, 파일의 디렉터리 항목은 전혀 만들어지지 않거나 파일이 만들어진 직후에 제거됩니다. 다른 플랫폼은 이를 지원하지 않습니다; 코드는 이 함수를 사용하여 만들어진 임시 파일이 파일 시스템에서 보이는 이름이 있거나 없는지에 의존해서는 안 됩니다.

결과 객체는 컨텍스트 관리자로 사용할 수 있습니다 (예를 참조하십시오). 컨텍스트가 완료되거나 파일 객체가 파괴되면 임시 파일이 파일 시스템에서 제거됩니다.

mode 매개 변수는 기본적으로 `'w+b'`로 설정되므로 만들어진 파일을 닫지 않고 읽고 쓸 수 있습니다. 저장된 데이터와 관계없이 모든 플랫폼에서 일관되게 작동하도록 바이너리 모드가 사용됩니다. *buffering*, *encoding*, *errors* 및 *newline*은 *open()*처럼 해석됩니다.

dir, *prefix* 및 *suffix* 매개 변수는 *mkstemp()*와 같은 의미와 기본값을 갖습니다.

반환된 객체는 POSIX 플랫폼에서 실제 파일 객체입니다. 다른 플랫폼에서는, `file` 어트리뷰트가 하부 실제 파일 객체인 파일류 객체입니다.

os.O_TMPFILE 플래그는 사용할 수 있고 작동하면 사용됩니다 (리눅스 특정, 리눅스 커널 3.11 이상이 필요합니다).

On platforms that are neither Posix nor Cygwin, TemporaryFile is an alias for NamedTemporaryFile.

인자 `fullpath`로 감사 이벤트 `tempfile.mkstemp`를 발생시킵니다.

버전 3.5에서 변경: 사용할 수 있으면 *os.O_TMPFILE* 플래그가 사용됩니다.

버전 3.8에서 변경: *errors* 매개 변수를 추가했습니다.

tempfile.**NamedTemporaryFile**(*mode='w+b', buffering=-1, encoding=None, newline=None, suffix=None, prefix=None, dir=None, delete=True, *, errors=None, delete_on_close=True*)

이 함수는 다음과 같은 차이를 제외하고는 *TemporaryFile()*과 똑같이 작동합니다:

- 이 함수는 파일 시스템에서 보이는 이름을 가지는 것이 보장되는 파일을 반환합니다.
- To manage the named file, it extends the parameters of `TemporaryFile()` with *delete* and *delete_on_close* parameters that determine whether and how the named file should be automatically deleted.

반환된 객체는 항상 `file` 어트리뷰트가 하부 실제 파일 객체인 파일류 객체입니다. 이 파일류 객체는 일반 파일과 마찬가지로 `with` 문에서 사용할 수 있습니다. 임시 파일의 이름은 반환된 파일류 객체의 `name` 어트리뷰트에서 꺼낼 수 있습니다. 유닉스에서, `TemporaryFile()`과 달리, 파일 생성 직후에 디렉터리 항목이 삭제(unlink)되지 않습니다.

If *delete* is true (the default) and *delete_on_close* is true (the default), the file is deleted as soon as it is closed. If *delete* is true and *delete_on_close* is false, the file is deleted on context manager exit only, or else when the *file-like object* is finalized. Deletion is not always guaranteed in this case (see `object.__del__()`). If *delete* is false, the value of *delete_on_close* is ignored.

Therefore to use the name of the temporary file to reopen the file after closing it, either make sure not to delete the file upon closure (set the *delete* parameter to be false) or, in case the temporary file is created in a `with` statement, set the *delete_on_close* parameter to be false. The latter approach is recommended as it provides assistance in automatic cleaning of the temporary file upon the context manager exit.

Opening the temporary file again by its name while it is still open works as follows:

- On POSIX the file can always be opened again.
- On Windows, make sure that at least one of the following conditions are fulfilled:
  - *delete* is false
  - additional open shares delete access (e.g. by calling `os.open()` with the flag `O_TEMPORARY`)
  - *delete* is true but *delete_on_close* is false. Note, that in this case the additional opens that do not share delete access (e.g. created via builtin `open()`) must be closed before exiting the context manager, else the `os.unlink()` call on context manager exit will fail with a `PermissionError`.

On Windows, if *delete_on_close* is false, and the file is created in a directory for which the user lacks delete access, then the `os.unlink()` call on exit of the context manager will fail with a `PermissionError`. This cannot happen when *delete_on_close* is true because delete access is requested by the open, which fails immediately if the requested access is not granted.

On POSIX (only), a process that is terminated abruptly with SIGKILL cannot automatically delete any NamedTemporaryFiles it created.

인자 `fullpath`로 감사 이벤트 `tempfile.mkstemp`를 발생시킵니다.

버전 3.8에서 변경: *errors* 매개 변수를 추가했습니다.

버전 3.12에서 변경: *delete_on_close* 매개 변수를 추가했습니다.

class tempfile.**SpooledTemporaryFile**(*max_size=0, mode='w+b', buffering=-1, encoding=None, newline=None, suffix=None, prefix=None, dir=None, *, errors=None*)

이 클래스는 파일 크기가 *max_size*를 초과할 때까지, 또는 파일의 `fileno()` 메서드가 호출될 때까지 데이터가 메모리에 스풀링 되는 것을 제외하고는 `TemporaryFile()`과 똑같이 작동합니다. 이 시점에서 내용은 디스크에 기록되고 `TemporaryFile()`처럼 작업이 진행됩니다.

rollover()

결과 파일에는 추가 메서드인 `rollover()`가 있으며, 파일 크기와 관계없이 파일을 디스크 상의 파일로 롤오버(roll over) 합니다.

반환된 객체는 파일류 객체인데, `rollover()`가 호출되었는지에 따라 `_file` 어트리뷰트는 `io.BytesIO`나 `io.TextIOWrapper` 객체(바이너리나 텍스트 *mode*가 지정되었는지에 따라)이거나 실제 파일 객체입니다. 이 파일류 객체는 일반 파일과 마찬가지로 `with` 문에서 사용할 수 있습니다.

버전 3.3에서 변경: truncate 메서드는 이제 *size* 인자를 허용합니다.

버전 3.8에서 변경: *errors* 매개 변수를 추가했습니다.

버전 3.11에서 변경: Fully implements the `io.BufferedIOBase` and `io.TextIOBase` abstract base classes (depending on whether binary or text *mode* was specified).

class tempfile.**TemporaryDirectory**(*suffix=None, prefix=None, dir=None, ignore_cleanup_errors=False, *, delete=True*)

이 클래스는 `mkdtemp()`와 같은 규칙을 사용하여 임시 디렉터리를 안전하게 만듭니다. 결과 객체는 컨텍스트 관리자로 사용할 수 있습니다 (예를 참조하십시오). 컨텍스트가 완료되거나 임시 디렉터리 객체가 파괴되면 새로 만들어진 임시 디렉터리와 모든 내용이 파일 시스템에서 제거됩니다.

name

반환된 객체의 `name` 어트리뷰트에서 디렉터리 이름을 꺼낼 수 있습니다. 반환된 객체가 컨텍스트 관리자로 사용될 때, `name`은 `with` 문의 `as` 절의 대상에 (있다면) 대입됩니다.

cleanup()

The directory can be explicitly cleaned up by calling the `cleanup()` method. If *ignore_cleanup_errors* is true, any unhandled exceptions during explicit or implicit cleanup (such as a `PermissionError` removing open files on Windows) will be ignored, and the remaining removable items deleted on a "best-effort" basis. Otherwise, errors will be raised in whatever context cleanup occurs (the `cleanup()` call, exiting the context manager, when the object is garbage-collected or during interpreter shutdown).

The *delete* parameter can be used to disable cleanup of the directory tree upon exiting the context. While it may seem unusual for a context manager to disable the action taken when exiting the context, it can be useful during debugging or when you need your cleanup behavior to be conditional based on other logic.

인자 `fullpath`로 감사 이벤트 `tempfile.mkdtemp`를 발생시킵니다.

Added in version 3.2.

버전 3.10에서 변경: *ignore_cleanup_errors* 매개 변수를 추가했습니다.

버전 3.12에서 변경: *delete* 매개 변수를 추가했습니다.

tempfile.**mkstemp**(*suffix=None, prefix=None, dir=None, text=False*)

가장 안전한 방식으로 임시 파일을 만듭니다. 플랫폼이 `os.open()`에서 `os.O_EXCL` 플래그를 올바르게 구현한다고 가정할 때, 파일 생성에 경쟁 조건이 없습니다. 파일은 만드는 사용자 ID만 읽고 쓸 수 있습니다. 플랫폼이 권한 비트를 사용하여 파일이 실행 가능한지를 나타내면, 파일은 아무도 실행할 수 없습니다. 파일 기술자는 자식 프로세스에 의해 상속되지 않습니다.

`TemporaryFile()`과 달리, `mkstemp()`의 사용자는 임시 파일로의 작업을 끝내면 파일을 삭제해야 합니다.

*suffix*가 `None`이 아니면, 파일 이름은 해당 접미사로 끝납니다, 그렇지 않으면 접미사가 없습니다. `mkstemp()`는 파일 이름과 접미사 사이에 점을 넣지 않습니다; 필요하면 *suffix*의 시작 부분에 넣으십시오.

*prefix*가 `None`이 아니면, 파일 이름은 해당 접두사로 시작합니다; 그렇지 않으면 기본 접두사가 사용됩니다. 기본값은 `gettempprefix()`나 `gettempprefixb()` 중 적절한 것의 반환 값입니다.

*dir*이 `None`이 아니면, 파일은 해당 디렉터리에 만들어집니다; 그렇지 않으면 기본 디렉터리가 사용됩니다. 기본 디렉터리는 플랫폼별 목록에서 선택되지만, 응용 프로그램 사용자는 *TMPDIR*, *TEMP* 또는 *TMP* 환경 변수를 설정하여 디렉터리 위치를 제어할 수 있습니다. 따라서 생성된 파일명이 `os.popen()`을 통해 외부 명령에 전달될 때 따옴표 처리할 필요가 없는 것과 같은 멋진 속성을 가질 것이라는 보장은 없습니다.

suffix, *prefix* 및 *dir* 중 어느 것이라도 `None`이 아니면, 그들은 같은 형이어야 합니다. 이들이 바이트열이면, 반환되는 이름은 str 대신 바이트열입니다. 기본 동작으로 바이트열 반환 값을 강제하려면 `suffix=b''`를 전달하십시오.

*text*가 지정되고 참이면, 파일은 텍스트 모드로 열립니다. 그렇지 않으면 (기본값) 파일은 바이너리 모드로 열립니다.

`mkstemp()`는 열린 파일에 대한 OS 수준 핸들(`os.open()`에서 반환하는 것)과 해당 파일의 절대 경로를 이 순서대로 포함하는 튜플을 반환합니다.

인자 `fullpath`로 감사 이벤트 `tempfile.mkstemp`를 발생시킵니다.

버전 3.5에서 변경: 바이트열 반환 값을 얻기 위해 *suffix*, *prefix* 및 *dir*를 이제 바이트열로 제공할 수 있습니다. 이전에는, str만 허용되었습니다. *suffix*와 *prefix*는 이제 기본값이 `None`이고 적절한 기본값이 사용되도록 합니다.

버전 3.6에서 변경: *dir* 매개 변수는 이제 경로류 객체를 받아들입니다.

`tempfile.`**`mkdtemp`**(*suffix=None*, *prefix=None*, *dir=None*)

가장 안전한 방식으로 임시 디렉터리를 만듭니다. 디렉터리 생성에 경쟁 조건이 없습니다. 디렉터리는 만드는 사용자 ID만 읽고 쓰고 검색할 수 있습니다.

`mkdtemp()`의 사용자는 임시 디렉터리로의 작업을 끝내면 임시 디렉터리와 디렉터리의 내용을 삭제해야 합니다.

prefix, *suffix* 및 *dir* 인자는 `mkstemp()`와 같습니다.

`mkdtemp()`는 새 디렉터리의 절대 경로명을 반환합니다.

인자 `fullpath`로 감사 이벤트 `tempfile.mkdtemp`를 발생시킵니다.

버전 3.5에서 변경: 바이트열 반환 값을 얻기 위해 *suffix*, *prefix* 및 *dir*를 이제 바이트열로 제공할 수 있습니다. 이전에는, str만 허용되었습니다. *suffix*와 *prefix*는 이제 기본값이 `None`이고 적절한 기본값이 사용되도록 합니다.

버전 3.6에서 변경: *dir* 매개 변수는 이제 경로류 객체를 받아들입니다.

버전 3.12에서 변경: 이제 *dir*이 상대 경로일 때도, `mkdtemp()`는 항상 절대 경로를 반환합니다.

`tempfile.`**`gettempdir`**()

임시 파일에 사용된 디렉터리 이름을 반환합니다. 이것은 이 모듈의 모든 함수에 대한 *dir* 인자의 기본값을 정의합니다.

파이썬은 표준 디렉터리 목록을 검색하여 호출하는 사용자가 파일을 만들 수 있는 디렉터리를 찾습니다. 목록은 다음과 같습니다:

1. `TMPDIR` 환경 변수로 명명된 디렉터리.
2. `TEMP` 환경 변수로 명명된 디렉터리.
3. `TMP` 환경 변수로 명명된 디렉터리.
4. 플랫폼별 위치:
   - 윈도우에서, 디렉터리 `C:\TEMP`, `C:\TMP`, `\TEMP` 및 `\TMP`, 이 순서대로.
   - 다른 모든 플랫폼에서, 디렉터리 `/tmp`, `/var/tmp` 및 `/usr/tmp`, 이 순서대로.
5. 최후의 수단으로, 현재 작업 디렉터리.

이 검색 결과는 캐시 됩니다, 아래 `tempdir` 설명을 참조하십시오.

버전 3.10에서 변경: Always returns a str. Previously it would return any `tempdir` value regardless of type so long as it was not `None`.

`tempfile.`**`gettempdirb`**()

`gettempdir()`과 같지만, 반환 값이 바이트열입니다.

Added in version 3.5.

`tempfile.`**`gettempprefix`**()

임시 파일을 만드는 데 사용된 파일명 접두사를 반환합니다. 디렉터리 구성 요소가 포함되어 있지 않습니다.

`tempfile.`**`gettempprefixb`**()

`gettempprefix()`와 같지만, 반환 값이 바이트열입니다.

Added in version 3.5.

모듈은 전역 변수를 사용하여 `gettempdir()`이 반환한 임시 파일에 사용되는 디렉터리의 이름을 저장합니다. 선택 절차를 무시하도록 직접 설정할 수 있지만, 권장하지 않습니다. 이 모듈의 모든 함수는 디렉터리를

지정하는 데 사용할 수 있는 *dir* 인자를 사용합니다. 이는 전역 API 동작을 변경하여 주의하지 않는 다른 코드를 놀라게 하지 않도록 권장되는 방법입니다.

tempfile.**tempdir**

None 이외의 값으로 설정되면, 이 변수는 이 모듈에 정의된 함수의 *dir* 인자의 기본값을 정의합니다 (형이 바이트열인지 문자열인지도 정의합니다). 경로류 객체일 수 없습니다.

*gettempprefix()*를 제외한 위의 함수를 호출할 때 tempdir이 None(기본값)이면 *gettempdir()*에 설명된 알고리즘에 따라 초기화됩니다.

> **참고**
>
> Beware that if you set tempdir to a bytes value, there is a nasty side effect: The global default return type of *mkstemp()* and *mkdtemp()* changes to bytes when no explicit prefix, suffix, or dir arguments of type str are supplied. Please do not write code expecting or depending on this. This awkward behavior is maintained for compatibility with the historical implementation.

11.5.1 예

다음은 *tempfile* 모듈의 일반적인 사용법에 대한 몇 가지 예입니다:

```
>>> import tempfile

# 임시 파일을 만들고 약간의 데이터를 기록합니다
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# 파일에서 데이터를 읽습니다
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# 파일을 닫습니다, 삭제됩니다
>>> fp.close()

# 컨텍스트 관리자를 사용해서 임시 파일을 만듭니다
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# 파일은 이제 닫히고 삭제됩니다

# 컨텍스트 관리자를 사용해서 임시 파일을 만듭니다
# 파일을 닫고, 이름을 사용해서 파일을 다십 엽니다
>>> with tempfile.NamedTemporaryFile(delete_on_close=False) as fp:
...     fp.write(b'Hello world!')
...     fp.close()
... # 파일은 닫히지만, 삭제되지는 않습니다
... # 이름을 사용해서 파일을 다시 엽니다
...     with open(fp.name, mode='rb') as f:
...         f.read()
b'Hello world!'
>>>
# 파일은 이제 삭제됩니다

# 컨텍스트 관리자를 사용해서 임시 디렉터리를 만듭니다
>>> with tempfile.TemporaryDirectory() as tmpdirname:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     print('created temporary directory', tmpdirname)
>>>
# 디렉터리와 내용은 삭제되었습니다
```

11.5.2 폐지된 함수와 변수

임시 파일을 만드는 역사적인 방법은 먼저 `mktemp()` 함수를 사용하여 파일 이름을 생성한 다음 이 이름을 사용하여 파일을 만드는 것입니다. 불행히도 `mktemp()` 호출과 파일을 만들려는 후속 시도 사이에 다른 프로세스가 이 이름으로 파일을 만들 수 있어서 이 방법은 안전하지 않습니다. 해결책은 두 단계를 결합하고 파일을 즉시 만드는 것입니다. 이 접근법이 `mkstemp()`와 위에서 설명한 다른 함수에서 사용됩니다.

tempfile.**mktemp**(*suffix=''*, *prefix='tmp'*, *dir=None*)

버전 2.3부터 폐지됨: 대신 `mkstemp()`를 사용하십시오.

호출하는 시점에 존재하지 않는 파일의 절대 경로명을 반환합니다. *prefix*, *suffix* 및 *dir* 인자는 바이트열 파일 이름, `suffix=None` 및 `prefix=None`이 지원되지 않는다는 점을 제외하고 `mkstemp()`의 같은 인자와 유사합니다.

경고

이 함수를 사용하면 프로그램에 보안 허점이 생길 수 있습니다. 반환된 파일 이름으로 무언가를 하면서 시간을 보내는 동안, 다른 누군가가 당신에게 펀치를 날릴 수 있습니다. `mktemp()` 사용은 `delete=False` 매개 변수를 전달하여 `NamedTemporaryFile()`로 쉽게 대체할 수 있습니다:

```
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmptjujjt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

11.6 glob — 유닉스 스타일 경로명 패턴 확장

소스 코드: Lib/glob.py

`glob` 모듈은 유닉스 셸이 사용하는 규칙에 따라 지정된 패턴과 일치하는 모든 경로명을 찾습니다. 하지만 결과는 임의의 순서로 반환됩니다. 물결표(tilde) 확장은 수행되지 않지만, `*`, `?` 및 `[]`로 표시되는 문자 범위는 올바르게 일치합니다. 이는 서브 셸을 실제로 호출하지 않고 `os.scandir()` 과 `fnmatch.fnmatch()` 함수를 사용하여 수행됩니다.

`fnmatch.fnmatch()`나 `pathlib.Path.glob()`과 달리, 점(`.`)으로 시작하는 파일은 점으로 시작하는 패턴으로만 일치시킬 수 있습니다. (물결표와 셸 변수 확장은 `os.path.expanduser()` 와 `os.path.expandvars()`를 사용하십시오.)

리터럴 일치를 위해서는, 대괄호 안에 메타 문자를 넣습니다. 예를 들어, `'[?]'`는 `'?'` 문자와 일치합니다.

The `glob` module defines the following functions:

glob.**glob**(*pathname*, ***, *root_dir=None*, *dir_fd=None*, *recursive=False*, *include_hidden=False*)

경로 지정을 포함하는 문자열인 *pathname*에 일치하는 경로 이름의 비어있을 수 있는 리스트를 반환합니다. *pathname*은 절대(`/usr/src/Python-1.5/Makefile`처럼)나 상대(`../../Tools/*/*.gif`처럼)일 수 있으며, 셸 스타일 와일드카드를 포함할 수 있습니다. 깨진 심볼릭 링크가 결과에 포함됩

니다 (셸과 마찬가지로). 결과가 정렬되는지는 파일 시스템에 따라 다릅니다. 이 함수 호출 중에 조건을 만족하는 파일이 제거되거나 추가되면, 해당 파일의 경로 이름이 포함될지는 지정되지 않습니다.

If *root_dir* is not `None`, it should be a *path-like object* specifying the root directory for searching. It has the same effect on `glob()` as changing the current directory before calling it. If *pathname* is relative, the result will contain paths relative to *root_dir*.

This function can support *paths relative to directory descriptors* with the *dir_fd* parameter.

*recursive*가 참이면, "`**`" 패턴은 모든 파일과 0개 이상의 디렉터리, 서브 디렉터리 및 디렉터리로의 심볼릭 링크와 일치합니다. 패턴 다음에 `os.sep`이나 `os.altsep`이 오면, 파일은 일치하지 않습니다.

If *include_hidden* is true, "`**`" pattern will match hidden directories.

`pathname`, `recursive`를 인자로 감사 이벤트 (*auditing event*) `glob.glob`을 발생시킵니다.

`pathname`, `recursive`, `root_dir`, `dir_fd`를 인자로 감사 이벤트 (*auditing event*) `glob.glob/2`을 발생시킵니다.

> **참고**
>
> 커다란 디렉터리 트리에서 "`**`" 패턴을 사용하면 과도한 시간이 걸릴 수 있습니다.

> **참고**
>
> This function may return duplicate path names if *pathname* contains multiple "`**`" patterns and *recursive* is true.

버전 3.5에서 변경: "`**`" 를 사용하는 재귀적 glob 지원.

버전 3.10에서 변경: *root_dir* 과 *dir_fd* 매개 변수를 추가했습니다.

버전 3.11에서 변경: *include_hidden* 매개 변수를 추가했습니다.

`glob.`**`iglob`**(*pathname*, *, *root_dir=None*, *dir_fd=None*, *recursive=False*, *include_hidden=False*)

실제로 동시에 저장하지 않고 `glob()`과 같은 값을 산출하는 이터레이터를 반환합니다.

`pathname`, `recursive`를 인자로 감사 이벤트 (*auditing event*) `glob.glob`을 발생시킵니다.

`pathname`, `recursive`, `root_dir`, `dir_fd`를 인자로 감사 이벤트 (*auditing event*) `glob.glob/2`을 발생시킵니다.

> **참고**
>
> This function may return duplicate path names if *pathname* contains multiple "`**`" patterns and *recursive* is true.

버전 3.5에서 변경: "`**`" 를 사용하는 재귀적 glob 지원.

버전 3.10에서 변경: *root_dir* 과 *dir_fd* 매개 변수를 추가했습니다.

버전 3.11에서 변경: *include_hidden* 매개 변수를 추가했습니다.

`glob.`**`escape`**(*pathname*)

모든 특수 문자(`'?'`, `'*'` 및 `'['`)를 이스케이프 처리합니다. 이것은 특수 문자가 들어있을 수 있는 임의의 리터럴 문자열을 일치시키려는 경우에 유용합니다. 드라이브/UNC 셰어 포인트의 특수 문자는 이스케이프 되지 않습니다, 예를 들어, 윈도우에서 `escape('//?/c:/Quo vadis?.txt')`는 `'//?/c:/Quo vadis[?].txt'`를 반환합니다.

Added in version 3.4.

glob.**translate**(*pathname*, *, *recursive=False*, *include_hidden=False*, *seps=None*)

Convert the given path specification to a regular expression for use with `re.match()`. The path specification can contain shell-style wildcards.

예를 들어:

```
>>> import glob, re
>>>
>>> regex = glob.translate('**/*.txt', recursive=True, include_hidden=True)
>>> regex
'(?s:(?:.+/)?[^/]*\\.txt)\\z'
>>> reobj = re.compile(regex)
>>> reobj.match('foo/bar/baz.txt')
<re.Match object; span=(0, 15), match='foo/bar/baz.txt'>
```

Path separators and segments are meaningful to this function, unlike `fnmatch.translate()`. By default wildcards do not match path separators, and `*` pattern segments match precisely one path segment.

If *recursive* is true, the pattern segment "`**`" will match any number of path segments.

If *include_hidden* is true, wildcards can match path segments that start with a dot (`.`).

A sequence of path separators may be supplied to the *seps* argument. If not given, `os.sep` and `altsep` (if available) are used.

> **더 보기**
>
> `pathlib.PurePath.full_match()` and `pathlib.Path.glob()` methods, which call this function to implement pattern matching and globbing.

Added in version 3.13.

11.6.1 예제

다음과 같은 파일을 포함하는 디렉터리를 고려하십시오: `1.gif`, `2.txt`, `card.gif` 및 `3.txt` 파일 만 포함하는 서브 디렉터리 `sub`. `glob()`은 다음과 같은 결과를 산출합니다. 경로의 선행 구성 요소가 보존되는 방법에 유의하십시오.

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
>>> glob.glob('**/*.txt', recursive=True)
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

디렉터리에 `.`으로 시작하는 파일이 있으면, 기본적으로 일치하지 않습니다. 예를 들어, `card.gif` 와 `.card.gif`를 포함하는 디렉터리를 고려하십시오:

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

더 보기

fnmatch 모듈은 셸 스타일 파일명 (경로가 아님) 확장을 제공합니다.

더 보기

pathlib 모듈은 고수준의 경로 객체를 제공합니다.

11.7 fnmatch — 유닉스 파일명 패턴 일치

소스 코드: Lib/fnmatch.py

이 모듈은 유닉스 셸 스타일의 와일드카드를 지원하며, 이는 정규식(*re* 모듈에서 설명합니다)과는 다릅니다. 셸 스타일 와일드카드에 사용되는 특수 문자는 다음과 같습니다:

패턴	의미
`*`	모든 것과 일치합니다
`?`	모든 단일 문자와 일치합니다
`[seq]`	*seq*의 모든 문자와 일치합니다.
`[!seq]`	*seq*에 없는 모든 문자와 일치합니다

리터럴 일치의 경우, 대괄호 안에 메타 문자를 넣습니다. 예를 들어, `'[?]'`는 `'?'` 문자와 일치합니다.

파일명 분리 기호(유닉스에서 `'/'`)는 이 모듈에서 특수하지 않습니다. 경로명 확장은 모듈 *glob*을 참조하십시오 (*glob*은 경로명 세그먼트와 일치시키기 위해 *filter()*를 사용합니다). 마찬가지로, 마침표로 시작하는 파일명은 이 모듈에서 특수하지 않으며, `*` 및 `?` 패턴과 일치합니다.

Unless stated otherwise, "filename string" and "pattern string" either refer to *str* or `ISO-8859-1` encoded *bytes* objects. Note that the functions documented below do not allow to mix a `bytes` pattern with a `str` filename, and vice-versa.

Finally, note that *functools.lru_cache()* with a *maxsize* of 32768 is used to cache the (typed) compiled regex patterns in the following functions: *fnmatch()*, *fnmatchcase()*, *filter()*, *filterfalse()*.

fnmatch.**fnmatch**(*name*, *pat*)

파일명 문자열 *name*이 패턴 문자열 *pat*와 일치하는지를 검사하여, `True` 나 `False`를 반환합니다. 두 매개 변수는 모두 *os.path.normcase()*를 사용하여 대소 문자를 정규화합니다. *fnmatchcase()*는 운영 체제의 표준인지에 관계없이, 대소문자를 구분하는 비교를 수행하는 데 사용할 수 있습니다.

이 예제는 현재 디렉터리의 확장자 `.txt` 인 모든 파일 이름을 인쇄합니다:

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print(file)
```

fnmatch.**fnmatchcase**(*name*, *pat*)

파일명 문자열 *name*이 패턴 문자열 *pat*와 일치하는지를 검사하여, `True` 나 `False`를 반환합니다; 비교는 대소 문자를 구분하며, *os.path.normcase()*를 적용하지 않습니다.

fnmatch.**filter**(*names*, *pat*)

패턴 문자열 *pat*와 일치하는 파일명 문자열의 이터러블 *names*의 요소로 리스트를 구축합니다. `[n for n in names if fnmatch(n, pat)]`과 같지만, 더 효율적으로 구현됩니다.

fnmatch.**filterfalse**(*names*, *pat*)

Construct a list from those elements of the *iterable* of filename strings *names* that do not match the pattern string *pat*. It is the same as `[n for n in names if not fnmatch(n, pat)]`, but implemented more efficiently.

Added in version 3.14.

fnmatch.**translate**(*pat*)

셸 스타일의 패턴 *pat*를 `re.match()`에서 사용하기 위해 정규식으로 변환한 값을 반환합니다. 패턴은 `str`일 것으로 기대합니다.

예제:

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'(?s:.*\\.txt)\\z'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<re.Match object; span=(0, 10), match='foobar.txt'>
```

더 보기

모듈 *glob*
유닉스 셸 스타일 경로 확장.

11.8 linecache — 텍스트 줄에 대한 무작위 액세스

소스 코드: Lib/linecache.py

`linecache` 모듈은 파이썬 소스 파일에서 임의의 줄을 가져올 수 있도록 하는데, 캐시를 사용하여 단일 파일에서 여러 줄을 읽는 일반적인 상황을 내부적으로 최적화하려고 시도합니다. 이것은 `traceback` 모듈에서 포맷된 트레이스백에 포함할 소스 줄을 가져오는 데 사용됩니다.

`tokenize.open()` 함수가 파일을 여는 데 사용됩니다. 이 함수는 `tokenize.detect_encoding()`를 사용하여 파일의 인코딩을 가져옵니다; 인코딩 토큰이 없으면, 파일 인코딩의 기본값은 UTF-8입니다.

`linecache` 모듈은 다음 함수를 정의합니다:

linecache.**getline**(*filename*, *lineno*, *module_globals=None*)

filename 파일에서 *lineno* 줄을 가져옵니다. 이 함수는 절대 예외를 발생시키지 않을 것입니다 — 에러 시 `''`를 반환합니다 (발견된 줄의 줄 바꿈 문자는 포함됩니다).

If *filename* indicates a frozen module (starting with `'<frozen '`), the function will attepmt to get the real file name from `module_globals['__file__']` if *module_globals* is not `None`.

*filename*이라는 파일이 없으면, 이 함수는 먼저 *module_globals*에서 **PEP 302** `__loader__`를 확인합니다. 그런 로더가 있고 `get_source` 메서드를 정의하고 있으면, 그것이 소스 줄을 결정합니다 (`get_source()`가 `None`을 반환하면, `''`이 반환됩니다). 마지막으로, *filename*이 상대 파일명이면, 모듈 검색 경로, `sys.path`, 에 있는 항목들에 상대적으로 검색됩니다.

버전 3.14에서 변경: Support *filename* of frozen modules.

linecache.**clearcache**()

캐시를 지웁니다. 이전에 `getline()`를 사용하여 읽은 파일의 줄이 더는 필요하지 않으면 이 함수를 사용하십시오.

linecache.**checkcache**(*filename=None*)

캐시의 유효성을 확인합니다. 캐시의 파일이 디스크에서 변경되었을 수 있고, 갱신된 버전이 필요하면 이 함수를 사용하십시오. *filename*이 생략되면, 캐시의 모든 항목을 검사합니다.

linecache.**lazycache**(*filename, module_globals*)

이후 호출에서 *module_globals*가 `None`이더라도 `getline()`을 통해 나중에 해당 줄을 가져올 수 있도록, 파일 기반이 아닌 모듈에 대한 충분한 정보를 캡처합니다. 이렇게 하면 라인이 실제로 필요할 때까지 모듈 전역을 무기한으로 들고 있지 않고도 I/O를 회피합니다.

Added in version 3.5.

예제:

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

11.9 shutil — 고수준 파일 연산

소스 코드: Lib/shutil.py

`shutil` 모듈은 파일과 파일 모음에 대한 여러 가지 고수준 연산을 제공합니다. 특히, 파일 복사와 삭제를 지원하는 함수가 제공됩니다. 개별 파일에 대한 연산에 대해서는, `os` 모듈도 참조하십시오.

> **경고**
>
> 더 고수준의 파일 복사 함수(`shutil.copy()`, `shutil.copy2()`)조차도 모든 파일 메타 데이터를 복사할 수는 없습니다.
>
> POSIX 플랫폼에서, 이는 ACL뿐만 아니라 파일 소유자와 그룹이 유실됨을 의미합니다. Mac OS에서는, 리소스 포크(resource fork)와 기타 메타 데이터가 사용되지 않습니다. 이는 리소스가 손실되고 파일 유형과 작성자 코드가 올바르지 않음을 의미합니다. 윈도우에서는, 파일 소유자, ACL 및 대체 데이터 스트림(alternate data streams)이 복사되지 않습니다.

11.9.1 디렉터리와 파일 연산

shutil.**copyfileobj**(*fsrc, fdst*[, *length*])

파일류 객체 *fsrc*의 내용을 파일류 객체 *fdst*에 복사합니다. 주어지면, 정수 *length*는 버퍼 크기입니다. 특히, 음의 *length* 값은 청크 단위로 소스 데이터를 반복하지 않고 데이터를 복사하는 것을 의미합니다; 기본적으로 제어되지 않은 메모리 소비를 피하고자 데이터를 청크로 읽습니다. *fsrc* 객체의 현재 파일 위치가 0이 아니면, 현재 파일 위치에서 파일 끝까지의 내용만 복사됨에 유의하십시오.

`copyfileobj()` will *not* guarantee that the destination stream has been flushed on completion of the copy. If you want to read from the destination at the completion of the copy operation (for example, reading the contents of a temporary file that has been copied from a HTTP stream), you must ensure that you have called `flush()` or `close()` on the file-like object before attempting to read the destination file.

shutil.**copyfile**(*src, dst, *, follow_symlinks=True*)

가능한 가장 효율적인 방법으로 이름이 *src* 인 파일의 내용을 (메타 데이터 없이) 이름이 *dst* 인 파일에 복사하고 *dst*를 반환합니다. *src*와 *dst*는 경로류 객체나 문자열로 지정된 경로 이름입니다.

*dst*는 완전한 대상 파일 이름이어야 합니다; 대상 디렉터리 경로를 허용하는 복사는 `copy()`를 참조하십시오. *src*와 *dst*가 같은 파일을 지정하면, `SameFileError`가 발생합니다.

대상 위치는 쓰기 가능해야 합니다; 그렇지 않으면, `OSError` 예외가 발생합니다. *dst*가 이미 존재하면, 교체됩니다. 문자나 블록 장치 및 파이프와 같은 특수 파일은 이 함수로 복사할 수 없습니다.

*follow_symlinks*가 거짓이고 *src*가 심볼릭 링크이면, *src*가 가리키는 파일을 복사하는 대신 새 심볼릭 링크가 만들어집니다.

인자 `src`, `dst`로 감사 이벤트 `shutil.copyfile`을 발생시킵니다.

버전 3.3에서 변경: 예전에는 `OSError` 대신 `IOError`를 발생시켰습니다. *follow_symlinks* 인자가 추가되었습니다. 이제 *dst*를 반환합니다.

버전 3.4에서 변경: `Error` 대신 `SameFileError`를 발생시킵니다. 후자는 전자의 서브 클래스라서, 이 변경은 이전 버전과 호환됩니다.

버전 3.8에서 변경: 파일을 더 효율적으로 복사하기 위해 플랫폼별 빠른 복사(fast-copy) 시스템 호출을 내부적으로 사용할 수 있습니다. 플랫폼 의존적 효율적인 복사 연산 섹션을 참조하십시오.

exception `shutil.`**`SpecialFileError`**

This exception is raised when `copyfile()` or `copytree()` attempt to copy a named pipe.

Added in version 2.7.

exception `shutil.`**`SameFileError`**

이 예외는 `copyfile()`의 소스와 대상이 같은 파일일 때 발생합니다.

Added in version 3.4.

`shutil.`**`copymode`**(*src*, *dst*, **, *follow_symlinks=True*)

*src*에서 *dst*로 권한 비트를 복사합니다. 파일 내용, 소유자 및 그룹은 영향을 받지 않습니다. *src*와 *dst*는 경로류 객체나 문자열로 지정된 경로 이름입니다. *follow_symlinks*가 거짓이고 *src*와 *dst*가 모두 심볼릭 링크이면, `copymode()`는 (가리키는 파일이 아니라) *dst* 자체의 모드를 수정하려고 시도합니다. 이 기능이 모든 플랫폼에서 사용 가능한 것은 아닙니다; 자세한 내용은 `copystat()`을 참조하십시오. `copymode()`가 로컬 플랫폼에서 심볼릭 링크를 수정할 수 없고, 그렇게 하도록 요청받으면, 아무것도 하지 않고 반환합니다.

인자 `src`, `dst`로 감사 이벤트 `shutil.copymode`를 발생시킵니다.

버전 3.3에서 변경: *follow_symlinks* 인자가 추가되었습니다.

`shutil.`**`copystat`**(*src*, *dst*, **, *follow_symlinks=True*)

권한 비트, 마지막 액세스 시간, 마지막 수정 시간 및 플래그를 *src*에서 *dst*로 복사합니다. 리눅스에서 `copystat()`은 가능하면 "확장 어트리뷰트(extended attributes)"도 복사합니다. 파일 내용, 소유자 및 그룹은 영향을 받지 않습니다. *src*와 *dst*는 경로류 객체나 문자열로 지정된 경로 이름입니다.

*follow_symlinks*가 거짓이고 *src*와 *dst*가 모두 심볼릭 링크를 참조하면, `copystat()`은 심볼릭 링크가 참조하는 파일이 아닌 심볼릭 링크 자체에 대해 작동합니다 - *src* 심볼릭 링크에서 정보를 읽고, *dst* 심볼릭 링크로 정보를 씁니다.

> **참고**
>
> 모든 플랫폼이 심볼릭 링크를 검사하고 수정할 수 있는 기능을 제공하지는 않습니다. 파이썬 자체는 어떤 기능이 로컬에서 사용 가능한지 알려줄 수 있습니다.
>
> - `os.chmod in os.supports_follow_symlinks`가 `True`이면, `copystat()`은 심볼릭 링크의 권한 비트를 수정할 수 있습니다.
> - `os.utime in os.supports_follow_symlinks`가 `True`이면, `copystat()`은 심볼릭 링크의 마지막 액세스와 수정 시간을 수정할 수 있습니다.
> - `os.chflags in os.supports_follow_symlinks`가 `True`이면, `copystat()`은 심볼릭 링크의 플래그를 수정할 수 있습니다. (`os.chflags`가 모든 플랫폼에서 사용 가능한 것은 아닙니다.)
>
> 이 기능 중 일부나 전부를 사용할 수 없는 플랫폼에서, 심볼릭 링크를 수정하라는 요청을 하면, `copystat()`은 가능한 모든 것들을 복사합니다. `copystat()`은 절대 실패를 반환하지 않습니다.
>
> 자세한 내용은 `os.supports_follow_symlinks`를 참조하십시오.

인자 src, dst로 감사 이벤트 shutil.copystat을 발생시킵니다.

버전 3.3에서 변경: *follow_symlinks* 인자와 리눅스 확장 어트리뷰트 지원을 추가했습니다.

shutil.**copy**(*src*, *dst*, **, *follow_symlinks=True*)

파일 *src*를 파일이나 디렉터리 *dst*에 복사합니다. *src*와 *dst*는 경로류 객체나 문자열이어야 합니다. *dst*가 디렉터리를 지정하면, 파일은 *src*의 기본 파일명을 사용하여 *dst*로 복사됩니다. *dst*가 이미 존재하는 파일을 지정하면 대체됩니다. 새로 만든 파일의 경로를 반환합니다.

*follow_symlinks*가 거짓이고, *src*가 심볼릭 링크이면, *dst*는 심볼릭 링크로 만들어집니다. *follow_symlinks*가 참이고 *src*가 심볼릭 링크이면, *dst*는 *src*가 참조하는 파일의 사본이 됩니다.

*copy()*는 파일 데이터와 파일의 권한 모드(*os.chmod()*를 참조하십시오)를 복사합니다. 파일의 생성과 수정 시간과 같은 다른 메타 데이터는 유지되지 않습니다. 원본의 모든 파일 메타 데이터를 유지하려면 대신 *copy2()*를 사용하십시오.

인자 src, dst로 감사 이벤트 shutil.copyfile을 발생시킵니다.

인자 src, dst로 감사 이벤트 shutil.copymode를 발생시킵니다.

버전 3.3에서 변경: *follow_symlinks* 인자가 추가되었습니다. 이제 새로 만든 파일의 경로를 반환합니다.

버전 3.8에서 변경: 파일을 더 효율적으로 복사하기 위해 플랫폼별 빠른 복사(fast-copy) 시스템 호출을 내부적으로 사용할 수 있습니다. 플랫폼 의존적 효율적인 복사 연산 섹션을 참조하십시오.

shutil.**copy2**(*src*, *dst*, **, *follow_symlinks=True*)

*copy2()*가 파일 메타 데이터 보존도 시도한다는 점을 제외하고는 *copy()*와 동일합니다.

*follow_symlinks*가 거짓이고, *src*가 심볼릭 링크이면, *copy2()*는 *src* 심볼릭 링크의 모든 메타 데이터를 새로 만들어진 *dst* 심볼릭 링크로 복사하려고 시도합니다. 그러나, 이 기능이 모든 플랫폼에서 사용 가능한 것은 아닙니다. 이 기능의 일부나 전부를 사용할 수 없는 플랫폼에서, *copy2()*는 가능한 모든 메타 데이터를 보존합니다; *copy2()*는 파일 메타 데이터를 보존할 수 없다는 이유로 예외를 발생시키지 않습니다.

*copy2()*는 *copystat()*을 사용하여 파일 메타 데이터를 복사합니다. 심볼릭 링크 메타 데이터 수정을 위한 플랫폼 지원에 대한 자세한 정보는 *copystat()*을 참조하십시오.

인자 src, dst로 감사 이벤트 shutil.copyfile을 발생시킵니다.

인자 src, dst로 감사 이벤트 shutil.copystat을 발생시킵니다.

버전 3.3에서 변경: *follow_symlinks* 인자를 추가하고, 확장 파일 시스템 어트리뷰트도 복사하려고 합니다 (현재 리눅스만 해당합니다). 이제 새로 만든 파일의 경로를 반환합니다.

버전 3.8에서 변경: 파일을 더 효율적으로 복사하기 위해 플랫폼별 빠른 복사(fast-copy) 시스템 호출을 내부적으로 사용할 수 있습니다. 플랫폼 의존적 효율적인 복사 연산 섹션을 참조하십시오.

shutil.**ignore_patterns**(**patterns*)

이 팩토리 함수는 *copytree()*의 *ignore* 인자를 위한 콜러블 함수로 사용할 수 있는 함수를 만드는데, 제공된 glob 스타일 *patterns* 중 하나와 일치하는 파일과 디렉터리를 무시하도록 합니다. 아래 예를 참조하십시오.

shutil.**copytree**(*src*, *dst*, *symlinks=False*, *ignore=None*, *copy_function=copy2*, *ignore_dangling_symlinks=False*, *dirs_exist_ok=False*)

*src*를 루트로 하는 전체 디렉터리 트리를 *dst*라는 디렉터리에 재귀적으로 복사하고 대상 디렉터리를 반환합니다. *dst*를 포함해야 하는 모든 중간 디렉터리도 기본적으로 생성됩니다.

디렉터리의 권한과 시간은 *copystat()*으로 복사되고, 개별 파일은 *copy2()*를 사용하여 복사됩니다.

*symlinks*가 참이면, 소스 트리의 심볼릭 링크가 새 트리에서 심볼릭 링크로 표시되고 플랫폼이 허용하는 한 원래 링크의 메타 데이터가 복사됩니다; 거짓이거나 생략되면, 링크된 파일의 내용과 메타 데이터가 새 트리에 복사됩니다.

*symlinks*가 거짓이면, 심볼릭 링크가 가리키는 파일이 존재하지 않으면, 복사 과정 종료 시 발생하는 `Error` 예외의 에러 리스트에 예외가 추가됩니다. 이 예외를 침묵시키려면 선택적 *ignore_dangling_symlinks* 플래그를 참으로 설정할 수 있습니다. 이 옵션은 `os.symlink()`를 지원하지 않는 플랫폼에는 영향을 미치지 않음에 주목하십시오.

*ignore*가 주어지면, `copytree()`가 방문하는 디렉터리와 `os.listdir()`가 반환한 이 디렉터리 내용의 리스트를 인자로 수신하는 콜러블 이어야 합니다. `copytree()`는 재귀적으로 호출되기 때문에, 복사되는 디렉터리마다 *ignore* 콜러블이 한 번 호출됩니다. 콜러블은 현재 디렉터리에 상대적인 디렉터리와 파일 이름의 시퀀스를 반환해야 합니다 (즉, 두 번째 인자에 있는 항목의 부분집합); 이 이름들은 복사 과정에서 무시됩니다. `ignore_patterns()`를 사용하여 glob 스타일 패턴을 기반으로 이름을 무시하는 콜러블을 만들 수 있습니다.

예외가 발생하면, 이유 리스트와 함께 `Error`가 발생합니다.

*copy_function*이 제공되면, 각 파일을 복사하는 데 사용되는 콜러블 이어야 합니다. 소스 경로와 대상 경로를 인자로 호출됩니다. 기본적으로, `copy2()`가 사용되지만, 같은 서명을 지원하는 (`copy()`와 같은) 모든 함수를 사용할 수 있습니다.

If *dirs_exist_ok* is false (the default) and *dst* already exists, a `FileExistsError` is raised. If *dirs_exist_ok* is true, the copying operation will continue if it encounters existing directories, and files within the *dst* tree will be overwritten by corresponding files from the *src* tree.

인자 `src`, `dst`로 감사 이벤트 `shutil.copytree`를 발생시킵니다.

버전 3.2에서 변경: 사용자 정의 복사 함수를 제공할 수 있도록 *copy_function* 인자를 추가했습니다. *symlinks*가 거짓일 때 연결이 끊긴(dangling) 심볼릭 링크 에러를 침묵시키도록 *ignore_dangling_symlinks* 인자를 추가했습니다.

버전 3.3에서 변경: *symlinks*가 거짓일 때 메타 데이터를 복사합니다. 이제 *dst*를 반환합니다.

버전 3.8에서 변경: 파일을 더 효율적으로 복사하기 위해 플랫폼별 빠른 복사(fast-copy) 시스템 호출을 내부적으로 사용할 수 있습니다. 플랫폼 의존적 효율적인 복사 연산 섹션을 참조하십시오.

버전 3.8에서 변경: *dirs_exist_ok* 매개 변수를 추가했습니다.

`shutil.rmtree`(*path*, *ignore_errors=False*, *onerror=None*, *, *onexc=None*, *dir_fd=None*)

전체 디렉터리 트리를 삭제합니다; *path*는 디렉터리를 가리켜야 합니다 (하지만 디렉터리에 대한 심볼릭 링크는 아닙니다). *ignore_errors*가 참이면, 삭제 실패로 인한 에러는 무시됩니다; 거짓이거나 생략되면, 이러한 에러는 *onexc*나 *onerror*로 지정된 처리기를 호출하여 처리하거나, 모두 생략하면 예외는 호출자에게 전파됩니다.

This function can support *paths relative to directory descriptors*.

> **참고**
>
> 필요한 fd 기반 함수를 지원하는 플랫폼에서는 기본적으로 `rmtree()`의 심볼릭 링크 공격 방지 버전이 사용됩니다. 다른 플랫폼에서는, `rmtree()` 구현이 심볼릭 링크 공격에 취약합니다: 적절한 타이밍과 상황에 따라, 공격자는 파일 시스템에서 심볼릭 링크를 조작하여 다른 방법으로는 액세스할 수 없는 파일을 삭제할 수 있습니다. 응용 프로그램은 `rmtree.avoids_symlink_attacks` 함수 어트리뷰트를 사용하여 어떤 버전이 사용되는지 판별할 수 있습니다.

*onexc*가 제공되면, 세 가지 매개 변수를 받아들이는 콜러블 이어야 합니다: *function*, *path* 및 *excinfo*.

첫 번째 매개 변수 *function*은 예외를 발생시킨 함수입니다; 플랫폼과 구현에 따라 다릅니다. 두 번째 매개 변수 *path*는 *function*에 전달된 경로 이름입니다. 세 번째 매개 변수 *excinfo*는 발생한 예외입니다. *onexc*에 의해 발생한 예외는 포착되지 않습니다.

The deprecated *onerror* is similar to *onexc*, except that the third parameter it receives is the tuple returned from `sys.exc_info()`.

> **더 보기**

rmtree 예 for an example of handling the removal of a directory tree that contains read-only files.

인자 `path`, `dir_fd`로 감사 이벤트 `shutil.rmtree`를 발생시킵니다.

버전 3.3에서 변경: 플랫폼이 fd 기반 함수를 지원하면 자동으로 사용되는 심볼릭 링크 공격 방지 버전을 추가했습니다.

버전 3.8에서 변경: 윈도우에서, 정션(junction)을 제거하기 전에 더는 디렉터리 정션의 내용을 삭제하지 않습니다.

버전 3.11에서 변경: *dir_fd* 매개 변수를 추가했습니다.

버전 3.12에서 변경: Added the *onexc* parameter, deprecated *onerror*.

버전 3.13에서 변경: `rmtree()` now ignores `FileNotFoundError` exceptions for all but the top-level path. Exceptions other than `OSError` and subclasses of `OSError` are now always propagated to the caller.

`rmtree.`**`avoids_symlink_attacks`**

현재 플랫폼과 구현이 `rmtree()`의 심볼릭 링크 공격 방지 버전을 제공하는지를 나타냅니다. 현재 이것은 fd 기반 디렉터리 액세스 함수를 지원하는 플랫폼에서만 참입니다.

Added in version 3.3.

`shutil.`**`move`**(*src*, *dst*, *copy_function=copy2*)

파일이나 디렉터리(*src*)를 다른 위치로 재귀적으로 옮기고 대상을 반환합니다.

대상이 기존 디렉터리이거나 디렉터리를 가리키는 심볼릭 링크면, *src*가 해당 디렉터리 내로 이동됩니다. 해당 디렉터리 내의 대상 경로는 이미 존재해서는 안 됩니다.

*dst*가 이미 존재하지만, 디렉터리가 아니면, `os.rename()` 의미에 따라 덮어쓸 수 있습니다.

대상이 현재 파일 시스템에 있으면, `os.rename()`이 사용됩니다. 그렇지 않으면, *copy_function*을 사용하여 *src*를 대상으로 복사한 다음 제거합니다. 심볼릭 링크의 경우, *src*의 대상을 가리키는 새 심볼릭 링크가 대상으로 만들어지고 *src*가 제거됩니다.

*copy_function*이 제공되면, *src*와 대상 두 개의 인자를 취하는 콜러블 이어야 하며, `os.rename()`을 사용할 수 없을 때 *src*를 대상으로 복사하는 데 사용됩니다. 소스가 디렉터리이면, `copytree()`가 호출되고 *copy_function*을 전달합니다. 기본 *copy_function*은 `copy2()` 입니다. *copy_function*으로 `copy()`를 사용하면 메타 데이터를 복사하지 않는 비용을 지불하는 대신 메타 데이터도 복사할 때 실패하는 이동이 성공할 수 있습니다.

인자 `src`, `dst`로 감사 이벤트 `shutil.move`를 발생시킵니다.

버전 3.3에서 변경: 외부 파일 시스템에 대한 명시적 심볼릭 링크 처리를 추가하여, GNU **mv**의 동작에 맞게 조정했습니다. 이제 *dst*를 반환합니다.

버전 3.5에서 변경: *copy_function* 키워드 인자를 추가했습니다.

버전 3.8에서 변경: 파일을 더 효율적으로 복사하기 위해 플랫폼별 빠른 복사(fast-copy) 시스템 호출을 내부적으로 사용할 수 있습니다. 플랫폼 의존적 효율적인 복사 연산 섹션을 참조하십시오.

버전 3.9에서 변경: *src*와 *dst* 모두에 대해 경로류 객체를 받아들입니다.

`shutil.`**`disk_usage`**(*path*)

지정된 경로(path)에 대한 디스크 사용량 통계를 *total*, *used* 및 *free* 어트리뷰트를 갖는 네임드 튜플로 반환합니다. 이들은 바이트 단위의 총, 사용된, 여유 공간의 양입니다. *path*는 파일이나 디렉터리일 수 있습니다.

> **참고**
>
> On Unix filesystems, *path* must point to a path within a **mounted** filesystem partition. On those platforms, CPython doesn' t attempt to retrieve disk usage information from non-mounted filesystems.

Added in version 3.3.

버전 3.8에서 변경: 윈도우에서, *path*는 이제 파일이나 디렉터리일 수 있습니다.

가용성: Unix, Windows.

shutil.**chown**(*path*, *user=None*, *group=None*, **, dir_fd=None*, *follow_symlinks=True*)

주어진 *path*의 소유자 *user* 및/또는 *group*을 변경합니다.

*user*는 시스템 사용자 이름이나 uid 일 수 있습니다; *group*도 마찬가지입니다. 최소한 하나의 인자가 필요합니다.

하부 함수인 `os.chown()`도 참조하십시오.

인자 `path`, `user`, `group`으로 감사 이벤트 `shutil.chown`을 발생시킵니다.

가용성: Unix.

Added in version 3.3.

버전 3.13에서 변경: *dir_fd* 와 *follow_symlinks* 매개 변수가 추가되었습니다.

shutil.**which**(*cmd*, *mode=os.F_OK | os.X_OK*, *path=None*)

주어진 *cmd*가 호출되면 실행될 실행 파일의 경로를 반환합니다. 아무런 *cmd*도 호출되지 않을 것이라면, `None`을 반환합니다.

*mode*는 `os.access()`에 전달되는 권한 마스크입니다, 기본적으로 파일이 존재하고 실행 가능한지를 판단합니다.

path is a "`PATH` string" specifying the directories to look in, delimited by `os.pathsep`. When no *path* is specified, the `PATH` environment variable is read from `os.environ`, falling back to `os.defpath` if it is not set.

If *cmd* contains a directory component, `which()` only checks the specified path directly and does not search the directories listed in *path* or in the system' s `PATH` environment variable.

On Windows, the current directory is prepended to the *path* if *mode* does not include `os.X_OK`. When the *mode* does include `os.X_OK`, the Windows API `NeedCurrentDirectoryForExePathW` will be consulted to determine if the current directory should be prepended to *path*. To avoid consulting the current working directory for executables: set the environment variable `NoDefaultCurrentDirectoryInExePath`.

또한 윈도우에서, `PATHEXT` 환경 변수는 확장자가 포함되지 않은 명령을 해석하는 데 사용됩니다. 예를 들어, `shutil.which("python")`을 호출하면, `which()`는 `PATHEXT`를 검색하여 *path* 디렉터리에서 `python.exe`를 찾아야 한다는 것을 알 수 있습니다. 예를 들어, 윈도우에서:

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

This is also applied when *cmd* is a path that contains a directory component:

```
>>> shutil.which("C:\\Python33\\python")
'C:\\Python33\\python.EXE'
```

Added in version 3.3.

버전 3.8에서 변경: 이제 `bytes` 형을 받아들입니다. *cmd* 형이 `bytes`이면 결과 형도 `bytes`입니다.

버전 3.12에서 변경: On Windows, the current directory is no longer prepended to the search path if *mode* includes `os.X_OK` and WinAPI `NeedCurrentDirectoryForExePathW(cmd)` is false, else the current directory is prepended even if it is already in the search path; `PATHEXT` is used now even when *cmd* includes a directory component or ends with an extension that is in `PATHEXT`; and filenames that have no extension can now be found.

exception shutil.**Error**

이 예외는 다중 파일 연산 중에 발생한 예외를 수집합니다. `copytree()`의 경우, 예외 인자는 3-튜플 (*srcname*, *dstname*, *exception*)의 리스트입니다.

플랫폼 의존적 효율적인 복사 연산

파이썬 3.8부터, 파일 복사를 수반하는 모든 함수(*copyfile()*, *copy()*, *copy2()*, *copytree()* 및 *move()*)는 파일을 더 효율적으로 복사하기 위해 플랫폼별 "빠른 복사(fast-copy)" 시스템 호출을 사용할 수 있습니다 (bpo-33671을 참조하십시오). "빠른 복사"는 복사 연산이 커널 내에서 발생하여, "`outfd.write(infd.read())`"와 같이 파이썬에서 사용자 공간 버퍼 사용을 피함을 의미합니다.

macOS에서는 fcopyfile이 (메타 데이터가 아닌) 파일 내용을 복사하는 데 사용됩니다.

On Linux *os.copy_file_range()* or *os.sendfile()* is used.

On Solaris *os.sendfile()* is used.

윈도우에서는 *shutil.copyfile()*이 더 큰 기본 버퍼 크기(64 KiB 대신 1 MiB)를 사용하고 *shutil.copyfileobj()*의 *memoryview()* 기반 변형이 사용됩니다.

빠른 복사 연산이 실패하고 대상 파일에 아무런 데이터도 기록되지 않았으면 shutil은 내부적으로 덜 효율적인 *copyfileobj()* 함수를 사용하여 조용히 폴 백 됩니다.

버전 3.8에서 변경.

버전 3.14에서 변경: Solaris now uses *os.sendfile()*.

버전 3.14에서 변경: Copy-on-write or server-side copy may be used internally via *os.copy_file_range()* on supported Linux filesystems.

copytree 예

ignore_patterns() 도우미를 사용하는 예:

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

이것은 `.pyc` 파일과 이름이 `tmp`로 시작하는 파일이나 디렉터리를 제외한 모든 것을 복사합니다.

로깅 호출을 추가하기 위해 *ignore* 인자를 사용하는 또 다른 예:

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # 아무것도 무시되지 않습니다

copytree(source, destination, ignore=_logpath)
```

rmtree 예

이 예는 윈도우에서 일부 파일에 읽기 전용 비트가 설정된 디렉터리 트리를 삭제하는 방법을 보여줍니다. onexc 콜백을 사용하여 읽기 전용 비트를 지우고 삭제를 다시 시도합니다. 후속 실패는 전파됩니다.

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "읽기 전용 비트를 지우고 삭제를 다시 시도합니다"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onexc=remove_readonly)
```

11.9.2 아카이브 연산

Added in version 3.2.

버전 3.5에서 변경: *xztar* 형식에 대한 지원이 추가되었습니다.

압축 및 아카이브 된 파일을 만들고 읽는 고수준 유틸리티도 제공됩니다. 이들은 `zipfile`과 `tarfile` 모듈에 의존합니다.

`shutil.make_archive`(*base_name*, *format*[, *root_dir*[, *base_dir*[, *verbose*[, *dry_run*[, *owner*[, *group*[, *logger*]]]]]]])

아카이브 파일(가령 zip이나 tar)을 만들고 이름을 반환합니다.

*base_name*은 만들 파일의 이름인데, 경로를 포함하고 형식별 확장자는 제외합니다.

format is the archive format: one of "zip" (if the `zlib` module is available), "tar", "gztar" (if the `zlib` module is available), "bztar" (if the `bz2` module is available), "xztar" (if the `lzma` module is available), or "zstdtar" (if the `compression.zstd` module is available).

*root_dir*은 아카이브의 루트 디렉터리가 될 디렉터리입니다, 아카이브의 모든 경로는 이것에 상대적입니다; 예를 들어, 보통 아카이브를 만들기 전에 *root_dir*로 chdir 합니다.

*base_dir*은 아카이브를 시작할 디렉터리입니다; 즉, *base_dir*은 아카이브에 있는 모든 파일과 디렉터리의 공통 접두사가 됩니다. *base_dir*은 *root_dir*에 상대적으로 제공되어야 합니다. *base_dir*과 *root_dir*을 함께 사용하는 방법은 *base_dir을 사용한 아카이브 예*를 참조하십시오.

*root_dir*과 *base_dir*은 모두 현재 디렉터리가 기본값입니다.

*dry_run*이 참이면, 아카이브가 만들어지지 않지만, 실행될 연산이 *logger*에 로그 됩니다.

tar 아카이브를 만들 때 *owner*와 **group*이 사용됩니다. 기본적으로, 현재 소유자와 그룹을 사용합니다.

*logger*는 **PEP 282**와 호환되는 객체여야 합니다, 일반적으로 `logging.Logger`의 인스턴스.

verbose 인자는 사용되지 않으며 폐지되었습니다.

인자 `base_name`, `format`, `root_dir`, `base_dir`로 감사 이벤트 `shutil.make_archive`를 발생시킵니다.

> **참고**
>
> This function is not thread-safe when custom archivers registered with `register_archive_format()` do not support the *root_dir* argument. In this case it temporarily changes the current working directory of the process to *root_dir* to perform archiving.

버전 3.8에서 변경: `format="tar"`로 만들어진 아카이브에 기존 GNU 형식 대신 최신 pax (POSIX.1-2001) 형식이 사용됩니다.

버전 3.10.6에서 변경: This function is now made thread-safe during creation of standard `.zip` and tar archives.

`shutil.get_archive_formats`()

아카이브에 지원되는 형식의 리스트를 반환합니다. 반환된 시퀀스의 각 요소는 튜플 `(name, description)` 입니다.

기본적으로 `shutil`은 다음 형식을 제공합니다:

- *zip*: ZIP 파일 (`zlib` 모듈을 사용할 수 있으면).
- *tar*: 압축되지 않은 tar 파일. 새 아카이브에 POSIX.1-2001 pax 형식을 사용합니다.
- *gztar*: gzip 된 tar 파일 (`zlib` 모듈을 사용할 수 있으면).
- *bztar*: bzip2 된 tar 파일 (`bz2` 모듈을 사용할 수 있으면).
- *xztar*: xz 된 tar 파일 (`lzma` 모듈을 사용할 수 있으면).
- *zstdtar*: Zstandard compressed tar-file (if the `compression.zstd` module is available).

register_archive_format() 을 사용하여, 새 형식을 등록하거나 기존 형식에 대해 여러분 자신의 아카이버를 제공할 수 있습니다.

shutil.**register_archive_format**(*name*, *function*[, *extra_args*[, *description*]])

name 형식을 위한 아카이버를 등록합니다.

*function*은 아카이브를 만드는 데 사용되는 콜러블입니다. 콜러블은 만들 파일의 *base_name*과 그 뒤에 아카이브를 시작할 *base_dir*(기본값은 os.curdir)를 받습니다. 추가 인자는 키워드 인자로 전달됩니다: *owner*, *group*, *dry_run* 및 *logger* (make_archive() 로 전달됩니다).

If *function* has the custom attribute function.supports_root_dir set to True, the *root_dir* argument is passed as a keyword argument. Otherwise the current working directory of the process is temporarily changed to *root_dir* before calling *function*. In this case make_archive() is not thread-safe.

주어지면, *extra_args*는 아카이버 콜러블이 사용될 때 추가 키워드 인자로 사용되는 (name, value) 쌍의 시퀀스입니다.

*description*은 아카이버 목록을 반환하는 get_archive_formats() 에서 사용됩니다. 기본값은 빈 문자열입니다.

버전 3.12에서 변경: *root_dir* 인자를 지원하는 함수들에 대한 지원이 추가되었습니다.

shutil.**unregister_archive_format**(*name*)

지원되는 형식 리스트에서 *name* 아카이브 형식을 제거합니다.

shutil.**unpack_archive**(*filename*[, *extract_dir*[, *format*[, *filter*]]])

아카이브를 풉니다. *filename*은 아카이브의 전체 경로입니다.

*extract_dir*은 아카이브가 풀리는 대상 디렉터리의 이름입니다. 제공되지 않으면, 현재 작업 디렉터리가 사용됩니다.

format is the archive format: one of "zip", "tar", "gztar", "bztar", "xztar", or "zstdtar". Or any other format registered with register_unpack_format(). If not provided, unpack_archive() will use the archive file name extension and see if an unpacker was registered for that extension. In case none is found, a ValueError is raised.

The keyword-only *filter* argument is passed to the underlying unpacking function. For zip files, *filter* is not accepted. For tar files, it is recommended to use 'data' (default since Python 3.14), unless using features specific to tar and UNIX-like filesystems. (See *Extraction filters* for details.)

인자 filename, extract_dir, format으로 감사 이벤트 shutil.unpack_archive를 발생시킵니다.

> **경고**
>
> Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of the path specified in the *extract_dir* argument, e.g. members that have absolute filenames starting with "/" or filenames with two dots "..".
>
> Since Python 3.14, the defaults for both built-in formats (zip and tar files) will prevent the most dangerous of such security issues, but will not prevent *all* unintended behavior. Read the *Hints for further verification* section for tar-specific details.

버전 3.7에서 변경: *filename*과 *extract_dir*에 대해 경로류 객체를 받아들입니다.

버전 3.12에서 변경: *filter* 인자를 추가했습니다.

shutil.**register_unpack_format**(*name*, *extensions*, *function*[, *extra_args*[, *description*]])

아카이브 해제기를 등록합니다. *name*은 형식의 이름이고 *extensions*는 형식에 해당하는 확장자의 리스트입니다, 가령 Zip 파일의 경우 .zip.

*function*은 아카이브를 푸는 데 사용되는 콜러블입니다. 콜러블은 다음을 받습니다:

- 아카이브의 경로, 위치 인자로;
- 아카이브를 추출해야 하는 디렉터리, 위치 인자로;

- possibly a *filter* keyword argument, if it was given to `unpack_archive()`;
- additional keyword arguments, specified by *extra_args* as a sequence of `(name, value)` tuples.

*description*은 형식을 설명하기 위해 제공될 수 있으며, `get_unpack_formats()` 함수에 의해 반환됩니다.

`shutil.`**`unregister_unpack_format`**`(`*name*`)`

아카이브 해제 형식을 등록 취소합니다. *name*은 형식의 이름입니다.

`shutil.`**`get_unpack_formats`**`()`

아카이브 해제를 위해 등록된 모든 형식의 리스트를 반환합니다. 반환된 시퀀스의 각 요소는 튜플 `(name, extensions, description)` 입니다.

기본적으로 `shutil`은 다음 형식을 제공합니다:

- *zip*: ZIP 파일 (압축된 파일의 해제는 해당 모듈을 사용할 수 있을 때만 작동합니다).
- *tar*: 압축되지 않은 tar 파일.
- *gztar*: gzip 된 tar 파일 (`zlib` 모듈을 사용할 수 있으면).
- *bztar*: bzip2 된 tar 파일 (`bz2` 모듈을 사용할 수 있으면).
- *xztar*: xz 된 tar 파일 (`lzma` 모듈을 사용할 수 있으면).
- *zstdtar*: Zstandard compressed tar-file (if the `compression.zstd` module is available).

`register_unpack_format()`을 사용하여, 새 형식을 등록하거나 기존 형식에 대한 여러분 자신의 아카이브 해제기를 제공할 수 있습니다.

아카이브 예

이 예에서는, 사용자의 `.ssh` 디렉터리에 있는 모든 파일을 포함하는 gzip 된 tar 파일 아카이브를 만듭니다:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

결과 아카이브에는 다음이 포함됩니다:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
-rwx------ tarek/staff     668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff     609 2008-06-09 13:26:54 ./id_dsa.pub
-rw------- tarek/staff    1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff     397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff   37192 2010-02-06 18:23:10 ./known_hosts
```

*base_dir*을 사용한 아카이브 예

이 예에서는, 위의 예와 유사하게, `make_archive()`를 사용하는 방법을 보여 주지만, 이번에는 *base_dir*을 사용합니다. 이제 다음 디렉터리 구조가 있습니다:

```
$ tree tmp
tmp
□□□ root
    □□□ structure
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        □□□ content
            □□□ please_add.txt
        □□□ do_not_add.txt
```

최종 아카이브에는, `please_add.txt` 가 포함되어야 하지만, `do_not_add.txt` 는 포함되지 않아야 합니다. 따라서 다음을 사용합니다:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> make_archive(
...     archive_name,
...     'tar',
...     root_dir='tmp/root',
...     base_dir='structure/content',
... )
'/Users/tarek/myarchive.tar'
```

결과 아카이브에 파일을 나열하면 다음과 같습니다:

```
$ python -m tarfile -l /Users/tarek/myarchive.tar
structure/content/
structure/content/please_add.txt
```

11.9.3 출력 터미널의 크기 조회하기

`shutil.`**`get_terminal_size`**(*fallback=(columns, lines)*)

터미널 창의 크기를 가져옵니다.

두 차원 각각에 대해, 환경 변수 `COLUMNS` 와 `LINES` 가 각각 확인됩니다. 변수가 정의되고 값이 양의 정수이면 사용됩니다.

`COLUMNS` 나 `LINES` 가 정의되지 않으면 (이것이 일반적입니다), *`os.get_terminal_size()`* 를 호출하여 *`sys.__stdout__`* 에 연결된 터미널을 조회합니다.

시스템이 조회를 지원하지 않거나, 터미널에 연결되어 있지 않기 때문에 터미널 크기를 성공적으로 조회할 수 없으면, `fallback` 매개 변수에 제공된 값이 사용됩니다. `fallback` 의 기본값은 많은 터미널 에뮬레이터에서 사용하는 기본 크기인 `(80, 24)` 입니다.

반환된 값은 *`os.terminal_size`* 형의 네임드 튜플입니다.

참조: The Single UNIX Specification, Version 2, Other Environment Variables.

Added in version 3.3.

버전 3.11 에서 변경: The `fallback` values are also used if *`os.get_terminal_size()`* returns zeroes.

더 보기

모듈 *`os`*
운영 체제 인터페이스. 파이썬 파일 객체보다 저수준으로 파일을 다루는 함수를 포함합니다.

모듈 *`io`*
파이썬의 내장 I/O 라이브러리. 추상 클래스와 파일 I/O와 같은 구상 클래스를 모두 포함합니다.

내장 함수 *`open()`*
파이썬으로 읽고 쓰기 위해 파일을 여는 표준 방법.

CHAPTER 12

데이터 지속성

이 장에서 설명하는 모듈은 파이썬 데이터를 디스크에 지속적인 형태로 저장하는 것을 지원합니다. `pickle`과 `marshal` 모듈은 많은 파이썬 데이터형을 바이트 스트림으로 바꿀 수 있고 그 바이트열로부터 객체를 재생성할 수 있습니다. 다양한 DBM 관련 모듈은 문자열에서 다른 문자열로의 매핑을 저장하는 일군의 해시 기반 파일 형식을 지원합니다.

이 장에서 설명하는 모듈 목록은 다음과 같습니다:

12.1 `pickle` — 파이썬 객체 직렬화

소스 코드: Lib/pickle.py

`pickle` 모듈은 파이썬 객체 구조의 직렬화와 역 직렬화를 위한 바이너리 프로토콜을 구현합니다. "피클링 (*pickling*)" 은 파이썬 객체 계층 구조가 바이트 스트림으로 변환되는 절차이며, "역 피클링 (*unpickling*)" 은 반대 연산으로, (바이너리 파일 이나 바이트열류 객체로 부터의) 바이트 스트림을 객체 계층 구조로 복원합니다. 피클링 (그리고 역 피클링) 은 "직렬화 (serialization)", "마샬링 (marshalling)"[1] 또는 "평탄화 (flattening)" 라고도 합니다; 그러나, 혼란을 피하고자, 여기에서 사용된 용어는 "피클링" 과 "역 피클링" 입니다.

> **경고**
>
> `pickle` 모듈은 **안전하지 않습니다**. 신뢰할 수 있는 데이터만 언 피클 하십시오.
>
> **언 피클 시 임의의 코드를 실행하는** 악의적인 피클 데이터를 구성할 수 있습니다. 신뢰할 수 없는 출처에서 왔거나 변조되었을 수 있는 데이터를 절대로 언 피클 하지 마십시오.
>
> 변조되지 않았음을 보장하려면 `hmac`으로 데이터에 서명하는 것을 고려하십시오.
>
> 신뢰할 수 없는 데이터를 처리한다면, `json`과 같은 안전한 직렬화 형식이 더 적합 할 수 있습니다. *json 과의 비교*를 참조하십시오.

12.1.1 다른 파이썬 모듈과의 관계

[1] 이것을 `marshal` 모듈과 혼동하지 마십시오.

marshal 과의 비교

파이썬이 *marshal* 이라 불리는 좀 더 원시적인 직렬화 모듈을 가지고 있지만, 일반적으로 *pickle* 은 항상 파이썬 객체를 직렬화하기 위해 선호되는 방법이어야 합니다. *marshal* 은 주로 파이썬의 `.pyc` 파일을 지원하기 위해 존재합니다.

pickle 모듈은 *marshal*과 몇 가지 중요한 점에서 다릅니다:

- *pickle* 모듈은 이미 직렬화된 객체를 추적하므로 나중에 같은 객체에 대한 참조가 다시 직렬화되지 않습니다. *marshal* 은 이렇게 하지 않습니다.

  이는 재귀 객체와 객체 공유에 모두 관련이 있습니다. 재귀 객체는 자신에 대한 참조를 포함하는 객체입니다. 이것은 마샬에 의해 처리되지 않으며, 실제로 재귀 객체를 마샬 하려고 하면 파이썬 인터프리터가 충돌합니다. 객체 공유는 직렬화되는 객체 계층의 다른 위치에서 같은 객체에 대한 다중 참조가 있을 때 발생합니다. *pickle* 은 그러한 객체를 한 번만 저장하고, 다른 모든 참조가 마스터 복사본을 가리키도록 만듭니다. 공유 객체는 공유된 상태로 유지되는데, 가변 객체의 경우 매우 중요할 수 있습니다.

- *marshal*은 사용자 정의 클래스와 인스턴스를 직렬화하는 데 사용할 수 없습니다. *pickle* 은 클래스 인스턴스를 투명하게 저장하고 복원할 수 있지만, 클래스 정의는 객체를 저장할 때와 같은 모듈에 존재하고 임포트 할 수 있어야 합니다.

- *marshal* 직렬화 형식은 파이썬 버전 간에 이식성이 보장되지 않습니다. 가장 중요한 일은 `.pyc` 파일을 지원하는 것이므로, 파이썬 구현자는 필요할 때 직렬화 형식을 과거 호환되지 않는 방식으로 변경할 권리를 갖습니다. *pickle* 직렬화 형식은, 호환성 있는 피클 프로토콜이 선택되고 여러분의 데이터가 파이썬 2와 파이썬 3의 호환되지 않는 언어 경계를 가로지를 때 피클링과 역 피클링 코드가 두 파이썬 형의 차이점을 다루는 한, 파이썬 배포 간의 과거 호환성을 보장합니다.

json 과의 비교

pickle 프로토콜과 JSON (JavaScript Object Notation) 간에는 근본적인 차이가 있습니다:

- JSON은 텍스트 직렬화 형식(유니코드 텍스트를 출력하지만, 대개는 `utf-8` 으로 인코딩됩니다)인 반면, pickle은 바이너리 직렬화 형식입니다.
- JSON은 사람이 읽을 수 있지만, 피클은 그렇지 않습니다.
- JSON은 상호 운용이 가능하며 파이썬 생태계 외부에서 널리 사용되는 반면, 피클은 파이썬으로만 한정됩니다.
- JSON은, 기본적으로, 파이썬 내장형 일부만 표시할 수 있으며 사용자 정의 클래스는 표시할 수 없습니다; 피클은 매우 많은 수의 파이썬 형을 나타낼 수 있습니다 (그중 많은 것들은 파이썬의 인트로스펙션 기능을 영리하게 사용하여 자동으로; 복잡한 경우는 특정 객체 *API* 를 구현해서 해결할 수 있습니다);
- pickle과 달리, 신뢰할 수 없는 JSON의 역 직렬화는 그 자체로 임의 코드 실행 취약점을 만들지는 않습니다.

더 보기

json 모듈: JSON 직렬화와 역 직렬화를 가능하게 하는 표준 라이브러리 모듈.

12.1.2 데이터 스트림 형식

pickle 이 사용하는 데이터 형식은 파이썬에 고유합니다. 이것은 JSON(포인터 공유를 나타낼 수 없음)과 같은 외부 표준에 의해 부과된 제약이 없다는 장점이 있습니다. 그러나 비 파이썬 프로그램은 피클 된 파이썬 객체를 재구성할 수 없다는 것을 의미합니다.

기본적으로, *pickle* 데이터 포맷은 상대적으로 간결한 바이너리 표현을 사용합니다. 최적의 크기 특성이 필요하다면, 피클 된 데이터를 효율적으로 압축 할 수 있습니다.

모듈 *pickletools* 에는 *pickle* 에 의해 생성된 데이터 스트림을 분석하는 도구가 있습니다. *pickletools* 소스 코드에는 피클 프로토콜에서 사용되는 옵코드(opcode)에 대한 광범위한 주석이 있습니다.

현재 피클링에 쓸 수 있는 6가지 프로토콜이 있습니다. 사용된 프로토콜이 높을수록, 생성된 피클을 읽으려면 더 최신 파이썬 버전이 필요합니다.

- 프로토콜 버전 0은 최초의 "사람이 읽을 수 있는" 프로토콜이며 이전 버전의 파이썬과 과거 호환됩니다.
- 프로토콜 버전 1은 역시 이전 버전의 파이썬과 호환되는 오래된 바이너리 형식입니다.
- 프로토콜 버전 2는 파이썬 2.3에서 소개되었습니다. 그것은 훨씬 더 효율적인 뉴스타일 클래스의 피클링을 제공합니다. 프로토콜 2에 의해 개선된 사항에 대한 정보는 **PEP 307**을 참조하십시오.
- 프로토콜 버전 3은 파이썬 3.0에서 추가되었습니다. 명시적으로 `bytes` 객체를 지원하며 파이썬 2.x에서 역 피클 될 수 없습니다. 이것은 파이썬 3.0–3.7에서 기본 프로토콜이었습니다.
- Protocol version 4 was added in Python 3.4. It adds support for very large objects, pickling more kinds of objects, and some data format optimizations. This was the default protocol in Python 3.8–3.13. Refer to **PEP 3154** for information about improvements brought by protocol 4.
- Protocol version 5 was added in Python 3.8. It adds support for out-of-band data and speedup for in-band data. It is the default protocol starting with Python 3.14. Refer to **PEP 574** for information about improvements brought by protocol 5.

> **참고**
>
> 직렬화는 지속성보다 더 원시적인 개념입니다; `pickle` 이 파일 객체를 읽거나 쓰기는 하지만, 지속적인 객체의 이름 지정도 (더 복잡한) 지속적인 객체에 대한 동시 액세스 문제도 처리하지 않습니다. `pickle` 모듈은 복잡한 객체를 바이트 스트림으로 변환할 수 있고 바이트 스트림을 같은 내부 구조를 가진 객체로 변환할 수 있습니다. 아마도 이러한 바이트 스트림으로 할 가장 분명한 작업은 파일에 쓰는 것이겠지만, 네트워크를 통해 보내거나 데이터베이스에 저장하는 것도 고려할 수 있습니다. `shelve` 모듈은 DBM 스타일의 데이터베이스 파일에 객체를 피클/역 피클 하는 간단한 인터페이스를 제공합니다.

12.1.3 모듈 인터페이스

객체 계층 구조를 직렬화하려면, 단순히 `dumps()` 함수를 호출하면 됩니다. 마찬가지로, 데이터 스트림을 역 직렬화하려면 `loads()` 함수를 호출합니다. 그러나, 직렬화와 역 직렬화에 대한 더 많은 제어를 원하면, 각각 `Pickler` 나 `Unpickler` 객체를 만들 수 있습니다.

`pickle` 모듈은 다음과 같은 상수를 제공합니다:

`pickle.`**`HIGHEST_PROTOCOL`**

정수, 사용 가능한 가장 높은 프로토콜 버전. 이 값은 함수 `dump()`와 `dumps()` 그리고 `Pickler` 생성자에 *protocol* 값으로 전달될 수 있습니다.

`pickle.`**`DEFAULT_PROTOCOL`**

An integer, the default *protocol version* used for pickling. May be less than `HIGHEST_PROTOCOL`. Currently the default protocol is 5, introduced in Python 3.8 and incompatible with previous versions. This version introduces support for out-of-band buffers, where **PEP 3118**-compatible data can be transmitted separately from the main pickle stream.

버전 3.0에서 변경: 기본 프로토콜은 3입니다.

버전 3.8에서 변경: 기본 프로토콜은 4입니다.

버전 3.14에서 변경: The default protocol is 5.

`pickle` 모듈은 피클링 절차를 보다 편리하게 하려고 다음과 같은 함수를 제공합니다:

`pickle.`**`dump`**(*obj, file, protocol=None, *, fix_imports=True, buffer_callback=None*)

객체 *obj* 의 피클 된 표현을 열린 파일 객체 *file* 에 씁니다. 이것은 `Pickler(file, protocol).dump(obj)` 와 동등합니다.

인자 *file, protocol, fix_imports* 및 *buffer_callback*은 `Pickler` 생성자에서와 같은 의미입니다.

버전 3.8에서 변경: *buffer_callback* 인자가 추가되었습니다.

pickle.**dumps**(*obj, protocol=None, *, fix_imports=True, buffer_callback=None*)

객체 *obj*의 피클 된 표현을 파일에 쓰는 대신 `bytes` 객체로 반환합니다.

인자 *protocol*, *fix_imports* 및 *buffer_callback*은 `Pickler` 생성자에서와 같은 의미입니다.

버전 3.8에서 변경: *buffer_callback* 인자가 추가되었습니다.

pickle.**load**(*file, *, fix_imports=True, encoding='ASCII', errors='strict', buffers=None*)

열린 파일 객체 *file* 에서 객체의 피클 된 표현을 읽고, 그 안에 지정된 객체 계층 구조를 재구성하여 반환합니다. 이것은 `Unpickler(file).load()` 와 동등합니다.

피클의 프로토콜 버전이 자동으로 감지되므로 프로토콜 인자가 필요하지 않습니다. 객체의 피클 된 표현 뒤에 남는 바이트열은 무시됩니다.

인자 *file*, *fix_imports*, *encoding*, *errors*, *strict* 및 *buffers*는 `Unpickler` 생성자에서와 같은 의미입니다.

버전 3.8에서 변경: *buffers* 인자가 추가되었습니다.

pickle.**loads**(*data, /, *, fix_imports=True, encoding='ASCII', errors='strict', buffers=None*)

객체의 피클 된 표현 *data*의 재구성된 객체 계층 구조를 반환합니다. *data*는 바이트열류 객체여야 합니다.

피클의 프로토콜 버전이 자동으로 감지되므로 프로토콜 인자가 필요하지 않습니다. 객체의 피클 된 표현 뒤에 남는 바이트열은 무시됩니다.

인자 *fix_imports*, *encoding*, *errors*, *strict* 및 *buffers*는 `Unpickler` 생성자에서와 같은 의미입니다.

버전 3.8에서 변경: *buffers* 인자가 추가되었습니다.

`pickle` 모듈은 세 가지 예외를 정의합니다:

exception pickle.**PickleError**

다른 피클링 예외의 공통 베이스 클래스입니다. `Exception`을 상속합니다.

exception pickle.**PicklingError**

`Pickler` 가 피클 가능하지 않은 객체를 만날 때 발생하는 에러. `PickleError` 를 상속합니다.

어떤 종류의 객체가 피클 될 수 있는지 배우려면 어떤 것이 피클 되고 역 피클 될 수 있을까요?를 참조하십시오.

exception pickle.**UnpicklingError**

데이터 손상 또는 보안 위반과 같이 객체를 역 피클 할 때 문제가 있으면 발생하는 에러. `PickleError` 를 상속합니다.

역 피클링 중에 다른 예외도 발생할 수 있음에 유의하십시오. AttributeError, EOFError, ImportError, IndexError 등이 발생할 수 있지만, 이에 국한되지는 않습니다.

`pickle` 모듈은 세 개의 클래스를 노출합니다, `Pickler`, `Unpickler` 및 `PickleBuffer`:

class pickle.**Pickler**(*file, protocol=None, *, fix_imports=True, buffer_callback=None*)

피클 데이터 스트림을 쓸 바이너리 파일을 받아들입니다.

선택적 *protocol* 인자(정수)는 피클러가 주어진 프로토콜을 사용하도록 지시합니다; 지원되는 프로토콜은 0부터 `HIGHEST_PROTOCOL` 입니다. 지정하지 않으면 기본값은 `DEFAULT_PROTOCOL` 입니다. 음수가 지정되면, `HIGHEST_PROTOCOL` 이 선택됩니다.

file 인자에는 단일 바이트열 인자를 받아들이는 write() 메서드가 있어야 합니다. 따라서 바이너리 쓰기를 위해 열린 디스크 상의 파일, `io.BytesIO` 인스턴스 또는 이 인터페이스를 충족시키는 다른 사용자 정의 객체일 수 있습니다.

fix_imports 가 참이고 *protocol* 이 3보다 작으면, pickle은 새로운 파이썬 3 이름을 파이썬 2에서 사용된 이전 모듈 이름에 매핑하려고 시도하여, 파이썬 2에서 피클 데이터 스트림을 읽을 수 있도록 합니다.

*buffer_callback*이 `None` (기본값) 이면, 버퍼 뷰는 피클 스트림의 일부로 *file*로 직렬화됩니다.

*buffer_callback*이 `None`이 아니면, 버퍼 뷰로 여러 번 호출될 수 있습니다. 콜백이 거짓 값(가령 `None`)을 반환하면, 주어진 버퍼는 아웃 오브 밴드입니다; 그렇지 않으면 버퍼는 인 밴드, 즉 피클 스트림 내부에 직렬화됩니다.

buffer_callback 이 None 이 아니고 *protocol* 이 None 이거나 5 보다 작으면 에러입니다.

버전 3.8 에서 변경: *buffer_callback* 인자가 추가되었습니다.

dump(*obj*)

생성자에 주어진 열린 파일 객체에 *obj* 의 피클 된 표현을 씁니다.

persistent_id(*obj*)

기본적으로 아무것도 하지 않습니다. 이것은 서브 클래스가 재정의할 수 있게 하려고 존재합니다.

persistent_id() 가 None 을 반환하면, *obj* 는 보통 때처럼 피클 됩니다. 다른 값은 *Pickler* 가 *obj* 의 지속성 (persistent) ID로 반환 값을 출력하도록 합니다. 이 지속성 ID의 의미는 *Unpickler.persistent_load()* 에 의해 정의되어야 합니다. *persistent_id()* 에 의해 반환된 값 자체는 지속성 ID를 가질 수 없음에 유의하십시오.

자세한 내용과 사용 예는 외부 객체의 지속성를 참조하십시오.

버전 3.13 에서 변경: Add the default implementation of this method in the C implementation of Pickler.

dispatch_table

피클러 객체의 디스패치 테이블은 *copyreg.pickle()* 을 사용하여 선언할 수 있는 환원 함수 (*reduction functions*) 의 등록소입니다. 키가 클래스이고 값이 환원 함수인 매핑입니다. 환원 함수는 관련 클래스의 단일 인자를 취하며 *__reduce__()* 메서드와 같은 인터페이스를 따라야 합니다.

기본적으로, 피클러 객체는 *dispatch_table* 어트리뷰트를 가지지 않을 것이고, 대신 *copyreg* 모듈에 의해 관리되는 전역 디스패치 테이블을 사용할 것입니다. 그러나 특정 피클러 객체의 피클링을 사용자 정의하기 위해서 *dispatch_table* 어트리뷰트를 딕셔너리류 객체로 설정할 수 있습니다. 또는, *Pickler* 의 서브 클래스가 *dispatch_table* 어트리뷰트를 가지고 있다면, 이 클래스의 인스턴스를 위한 기본 디스패치 테이블로 사용됩니다.

사용 예는 디스패치 테이블을 참조하십시오.

Added in version 3.3.

reducer_override(*obj*)

Pickler 서브 클래스에서 정의할 수 있는 특수 환원기 (reducer). 이 메서드는 *dispatch_table*에 있는 모든 감속기보다 우선순위가 높습니다. *__reduce__()* 메서드와 같은 인터페이스를 따라야 하며, 선택적으로 *NotImplemented*를 반환하여 obj를 피클 하기 위해 *dispatch_table* 등록 환원기로 폴백 (fallback) 하도록 할 수 있습니다.

자세한 예는 형, 함수 및 기타 객체에 대한 사용자 정의 환원을 참조하십시오.

Added in version 3.8.

fast

폐지되었습니다. 참값으로 설정된 경우 빠른 모드를 활성화합니다. 빠른 모드는 메모 사용을 비활성화하므로, 불필요한 PUT 옵코드를 생성하지 않아 피클링 절차의 속도를 높입니다. 자신을 참조하는 객체에 사용되면 안 됩니다. 그렇지 않으면 *Pickler* 가 무한 재귀에 빠집니다.

더 간결한 피클이 필요하면 *pickletools.optimize()* 를 사용하십시오.

clear_memo()

Clears the pickler' s "memo".

The memo is the data structure that remembers which objects the pickler has already seen, so that shared or recursive objects are pickled by reference and not by value. This method is useful when re-using picklers.

class pickle.**Unpickler**(*file, *, fix_imports=True, encoding='ASCII', errors='strict', buffers=None*)

피클 데이터 스트림을 읽는 데 사용될 바이너리 파일을 받아들입니다.

피클의 프로토콜 버전이 자동으로 감지되므로 프로토콜 인자가 필요하지 않습니다.

인자 *file* 에는 세 가지 메서드가 있어야 합니다, `io.BufferedIOBase` 인터페이스 처럼, 정수 인자를 받아들이는 read() 메서드, 버퍼 인자를 받아들이는 readinto() 메서드 그리고 인자가 없는 readline() 메서드. 따라서 *file* 은 바이너리 읽기를 위해 열린 디스크 상의 파일, `io.BytesIO` 객체 또는 이 인터페이스를 만족하는 다른 사용자 정의 객체일 수 있습니다.

선택적 인자 *fix_imports*, *encoding* 및 *errors*는 파이썬 2에서 생성된 피클 스트림에 대한 호환성 지원을 제어하는 데 사용됩니다. *fix_imports* 가 참이면, pickle은 이전 파이썬 2 이름을 파이썬 3에서 사용된 새로운 이름으로 매핑하려고 합니다. *encoding* 과 *errors* 는 파이썬 2에 의해 피클 된 8비트 문자열 인스턴스를 디코딩하는 방법을 pickle에게 알려줍니다. 기본값은 각각 'ASCII' 와 'strict' 입니다. *encoding* 은 'bytes' 가 될 수 있는데, 8비트 문자열 인스턴스를 바이트열 객체로 읽습니다. NumPy 배열과 파이썬 2에서 피클 된 `datetime`, `date` 및 `time` 인스턴스를 역 피클링하려면 `encoding='latin1'`을 사용해야 합니다.

*buffers*가 `None` (기본값) 이면, 역 직렬화에 필요한 모든 데이터가 피클 스트림에 포함되어야 합니다. 이것은 `Pickler`가 인스턴스 화 될 때 (또는 `dump()` 나 `dumps()` 가 호출될 때) *buffer_callback* 인자가 `None`이었음을 뜻합니다.

*buffers*가 `None`이 아니면, 피클 스트림이 아웃 오브 밴드 버퍼 뷰를 참조할 때마다 소비되는 버퍼가 활성화된 객체의 이터러블이어야 합니다. 이러한 버퍼는 Pickler 객체의 *buffer_callback*에 순서대로 제공되었습니다.

버전 3.8에서 변경: *buffers* 인자가 추가되었습니다.

load()

생성자에 주어진 열린 파일 객체에서 객체의 피클 된 표현을 읽고, 그 안에 지정된 객체 계층 구조를 재구성하여 반환합니다. 객체의 피클 된 표현 뒤에 남는 바이트열은 무시됩니다.

persistent_load(*pid*)

기본적으로 `UnpicklingError`를 발생시킵니다.

정의되면, `persistent_load()` 는 지속성 ID *pid* 로 지정된 객체를 반환해야 합니다. 유효하지 않은 지속성 ID가 발견되면 `UnpicklingError`를 일으켜야 합니다.

자세한 내용과 사용 예는 외부 객체의 지속성를 참조하십시오.

버전 3.13에서 변경: Add the default implementation of this method in the C implementation of `Unpickler`.

find_class(*module*, *name*)

필요하면 *module* 을 임포트하고 거기에서 *name* 이라는 객체를 반환합니다. 여기서 *module* 및 *name* 인자는 `str` 객체입니다. 그 이름이 제시하는 것과는 달리, `find_class()` 는 함수를 찾는 데에도 사용됨에 유의하십시오.

로드되는 객체의 형과 로드 방법을 제어하기 위해 서브 클래스는 이것을 재정의할 수 있고, 잠재적으로 보안 위험을 감소시킵니다. 자세한 내용은 전역 제한하기를 참조하십시오.

`module`, `name`을 인자로 감사 이벤트(*auditing event*) `pickle.find_class`를 발생시킵니다.

class pickle.**PickleBuffer**(*buffer*)

피클 가능한 데이터를 나타내는 버퍼의 래퍼. *buffer*는 바이트열류 객체나 N-차원 배열과 같은 버퍼 제공 객체여야 합니다.

`PickleBuffer` 자체가 버퍼 제공자이므로, `memoryview`와 같은 버퍼 제공 객체를 기대하는 다른 API로 전달할 수 있습니다.

`PickleBuffer` 객체는 피클 프로토콜 5 이상만 사용하여 직렬화할 수 있습니다. 그들은 아웃 오브 밴드 직렬화 대상입니다.

Added in version 3.8.

raw()

이 버퍼의 하부 메모리 영역의 `memoryview`를 반환합니다. 반환된 객체는 `B` (부호 없는 바이트) 형식의 1-차원 C 연속 메모리 뷰입니다. 버퍼가 C나 포트란 연속적이지 않으면 `BufferError`가 발생합니다.

release()

PickleBuffer 객체에 의해 노출된 하부 버퍼를 해제합니다.

12.1.4 어떤 것이 피클 되고 역 피클 될 수 있을까요?

다음 형을 피클 할 수 있습니다:

- built-in constants (`None`, `True`, `False`, `Ellipsis`, and *`NotImplemented`*);
- 정수, 실수, 복소수;
- 문자열, 바이트열, 바이트 배열(bytearray);
- 피클 가능한 객체만 포함하는 튜플, 리스트, 집합과 딕셔너리;
- 모듈의 최상위 수준에서 액세스할 수 있는 (내장 및 사용자 정의) 함수 (`lambda` 가 아니라 `def` 를 사용하는);
- 모듈의 최상위 수준에서 액세스할 수 있는 클래스;
- 그런 클래스의 인스턴스 중에서 *`__getstate__()`* 를 호출한 결과가 피클 가능한 것들 (자세한 내용은 클래스 인스턴스 피클링 절을 참조하세요).

피클 가능하지 않은 객체를 피클 하려고 하면 *`PicklingError`* 예외가 발생합니다; 이런 일이 일어났을 때, 특정할 수 없는 길이의 바이트열이 하부 파일에 이미 기록되었을 수 있습니다. 매우 재귀적인 데이터 구조를 피클 하려고 하면 최대 재귀 깊이를 초과할 수 있고, 이때 *`RecursionError`* 가 발생합니다. *`sys.setrecursionlimit()`* 을 사용하여 이 제한을 조심스럽게 올릴 수 있습니다.

함수(내장 및 사용자 정의)는 값이 아니라 완전히 정규화된 이름으로 피클 됨에 유의하십시오.[2] 이것은 함수를 포함하는 모듈과 클래스의 이름과 함께 함수의 이름만 피클 된다는 것을 의미합니다. 함수의 코드도 함수 어트리뷰트도 피클 되지 않습니다. 따라서 정의하는 모듈은 역 피클 환경에서 임포트 가능해야 하며, 모듈에는 그 이름의 객체가 있어야 합니다. 그렇지 않으면 예외가 발생합니다.[3]

마찬가지로, 클래스는 완전히 정규화된 이름으로 피클 되므로 역 피클링 환경에서 같은 제한이 적용됩니다. 클래스의 코드 나 데이터가 피클 되지 않음에 유의하세요. 그래서 다음 예제에서 클래스 어트리뷰트 `attr` 은 역 피클링 환경에서 복원되지 않습니다:

```
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

이러한 제한이 피클 가능한 함수와 클래스가 모듈의 최상위 수준에서 정의되어야 하는 이유입니다.

마찬가지로, 클래스 인스턴스가 피클 될 때, 클래스의 코드와 데이터는 함께 피클 되지 않습니다. 인스턴스 데이터만 피클 됩니다. 이는 의도한 것으로, 클래스의 버그를 수정하거나 클래스에 메서드를 추가할 수 있고, 이전 버전의 클래스로 만들어진 객체를 여전히 로드 할 수 있습니다. 여러 버전의 클래스에 걸치는 수명이 긴 객체를 만들 계획이라면, 클래스의 *`__setstate__()`* 메서드로 적절한 변환을 할 수 있도록 객체에 버전 번호를 넣는 것이 좋습니다.

12.1.5 클래스 인스턴스 피클링

이 절에서는 클래스 인스턴스를 피클 및 역 피클 하는 방법을 정의, 사용자 정의 및 제어할 수 있는 일반적인 메커니즘을 설명합니다.

대부분은, 인스턴스를 피클 가능하게 만드는 데 추가 코드가 필요하지 않습니다. 기본적으로, pickle은 인트로스펙션을 통해 인스턴스의 클래스와 어트리뷰트를 조회합니다. 클래스 인스턴스가 역 피클 될 때, `__init__()` 메서드는 보통 호출되지 않습니다. 기본 동작은, 먼저 초기화되지 않은 인스턴스를 만든 다음 저장된 어트리뷰트를 복원합니다. 다음 코드는 이 동작의 구현을 보여줍니다:

[2] 이것이 `lambda` 함수가 pickle 될 수 없는 이유입니다: 모든 `lambda` 함수는 같은 이름을 공유합니다: `<lambda>`.
[3] 발생하는 예외는 *`ImportError`* 나 *`AttributeError`* 일 가능성이 크지만, 그 밖의 다른 것일 수 있습니다.

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def restore(cls, attributes):
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

클래스는 다음과 같은 하나 이상의 특수 메서드를 제공하여 기본 동작을 변경할 수 있습니다:

object.**__getnewargs_ex__**()

프로토콜 2 이상에서, *__getnewargs_ex__()* 메서드를 구현하는 클래스는 역 피클링 때 __new__() 메서드에 전달되는 값을 지시할 수 있습니다. 이 메서드는 (args, kwargs) 쌍을 반환해야 합니다. *args* 는 위치 인자의 튜플이고 *kwargs* 는 이름있는 인자의 딕셔너리인데, 객체를 구성하는 데 사용됩니다. 그것들은 역 피클링 때 __new__() 메서드로 전달될 것입니다.

클래스의 __new__() 메서드에 키워드 전용 인자가 필요하면 이 메서드를 구현해야 합니다. 그렇지 않으면 호환성을 위해 *__getnewargs__()* 를 구현하는 것이 좋습니다.

버전 3.6에서 변경: *__getnewargs_ex__()* 는 이제 프로토콜 2와 3에서 사용됩니다.

object.**__getnewargs__**()

이 메서드는 *__getnewargs_ex__()* 와 비슷한 목적을 수행하지만, 위치 인자만 지원합니다. 역 피클링 때 __new__() 메서드에 전달될 인자의 튜플 args 를 반환해야 합니다.

__getnewargs_ex__() 가 정의되면 *__getnewargs__()* 는 호출되지 않습니다.

버전 3.6에서 변경: 파이썬 3.6 이전에는, 프로토콜 2와 3에서 *__getnewargs_ex__()* 대신 *__getnewargs__()* 가 호출되었습니다.

object.**__getstate__**()

클래스는 *__getstate__()* 메서드를 재정의함으로써 인스턴스가 피클 되는 방식을 추가로 제어할 수 있습니다. 기본 상태 대신, 이 메서드가 호출되고 반환된 객체를 인스턴스의 내용으로 피클 합니다. 다음과 같은 경우가 있습니다:

- For a class that has no instance __dict__ and no __slots__, the default state is None.
- For a class that has an instance __dict__ and no __slots__, the default state is self.__dict__.
- For a class that has an instance __dict__ and __slots__, the default state is a tuple consisting of two dictionaries: self.__dict__, and a dictionary mapping slot names to slot values. Only slots that have a value are included in the latter.
- For a class that has __slots__ and no instance __dict__, the default state is a tuple whose first item is None and whose second item is a dictionary mapping slot names to slot values described in the previous bullet.

버전 3.11에서 변경: Added the default implementation of the __getstate__() method in the *object* class.

object.**__setstate__**(*state*)

역 피클링 때, 클래스가 *__setstate__()* 를 정의하면, 그것은 역 피클 된 상태(state)로 호출됩니다. 이 경우 상태 객체가 딕셔너리일 필요는 없습니다. 그렇지 않으면, 피클 된 상태는 딕셔너리 여야하고 그 항목이 새 인스턴스의 딕셔너리에 삽입됩니다.

> **참고**
>
> *__reduce__()* 가 피클링 때 값이 거짓 None 인 상태를 반환하면, *__setstate__()* 메서드가 역 피클링 때 호출되지 않습니다.

__getstate__() 와 *__setstate__()* 메서드를 사용하는 방법에 대한 더 자세한 정보는 상태 저장 객체 처리 절을 참조하십시오.

> **참고**
>
> 역 피클링 시간에, `__getattr__()`, `__getattribute__()`, 또는 `__setattr__()` 같은 메서드가 인스턴스에 호출될 수 있습니다. 그러한 메서드들이 어떤 내부 불변성이 참인 것에 의존하는 경우, 형은 그런 불변성을 유지하기 위해 `__new__()` 를 구현해야 합니다, 인스턴스를 역 피클링할 때 `__init__()` 가 호출되지 않기 때문입니다.

앞으로 살펴보겠지만, 피클은 위에서 설명한 메서드를 직접 사용하지 않습니다. 사실, 이 메서드들은 *__reduce__()* 특수 메서드를 구현하는 복사 프로토콜의 일부입니다. 복사 프로토콜은 객체를 피클 하고 복사하는 데 필요한 데이터를 조회하기 위한 통일된 인터페이스를 제공합니다.[4]

강력하기는 하지만, 여러분의 클래스에서 직접 *__reduce__()* 를 구현하면 잘못되기 쉽습니다. 이런 이유로, 클래스 설계자는 가능하면 고수준 인터페이스(즉, *__getnewargs_ex__()*, *__getstate__()* 및 *__setstate__()*)를 사용해야 합니다. 하지만, 우리는 `__reduce__()` 를 사용하는 것이 유일한 옵션이거나 더 효율적인 피클링을 제공하거나 혹은 둘 다인 경우를 보여줄 것입니다.

`object.__reduce__()`

인터페이스는 현재 다음과 같이 정의됩니다. *__reduce__()* 메서드는 아무런 인자도 받아들이지 않으며 문자열이나 바람직하게는 튜플을 반환합니다 (반환된 객체는 흔히 "환원 값(reduce value)" 이라고 불립니다).

문자열이 반환되면, 문자열은 전역 변수의 이름으로 해석되어야 합니다. 모듈에 상대적인 객체의 지역 이름이어야 합니다; pickle 모듈은 객체의 모듈을 결정하기 위해 모듈 이름 공간을 검색합니다. 이 동작은 일반적으로 싱글톤에 유용합니다.

튜플이 반환될 때는, 길이가 2나 6이 되어야 합니다. 선택적인 항목은 생략되거나 `None` 이 값으로 제공될 수 있습니다. 각 항목의 의미는 순서대로 다음과 같습니다:

- 객체의 초기 버전을 만들기 위해 호출할 콜러블 객체.
- 콜러블 객체에 대한 인자의 튜플. 콜러블 객체가 인자를 받아들이지 않으면 빈 튜플을 제공해야 합니다.
- 선택적으로, 객체의 상태. 앞에서 설명한 대로 객체의 *__setstate__()* 메서드에 전달됩니다. 객체에 그런 메서드가 없다면, 그 값은 딕셔너리 여야 하며 객체의 `__dict__` 어트리뷰트에 추가됩니다.
- Optionally, an iterator (and not a sequence) yielding successive items. These items will be appended to the object either using `obj.append(item)` or, in batch, using `obj.extend(list_of_items)`. This is primarily used for list subclasses, but may be used by other classes as long as they have *append()* and *extend()* methods with the appropriate signature. (Whether `append()` or `extend()` is used depends on which pickle protocol version is used as well as the number of items to append, so both must be supported.)
- 선택적으로, 연속적인 키-값 쌍을 생성하는 이터레이터(시퀀스가 아닙니다). 이 항목들은 `obj[key] = value` 를 사용하여 객체에 저장됩니다. 이것은 주로 딕셔너리 서브 클래스에 사용되지만, `__setitem__()` 을 구현하는 한 다른 클래스에서 사용될 수 있습니다.
- 선택적으로, `(obj, state)` 서명을 가진 콜러블. 이 콜러블은 `obj`의 정적 *__setstate__()* 메서드 대신에 특정 객체의 상태 갱신 동작을 프로그래밍 방식으로 제어할 수 있도록 합니다. `None`이 아니면, 이 콜러블은 `obj`의 *__setstate__()*보다 우선 합니다.

  Added in version 3.8: 선택적인 여섯 번째 튜플 항목 `(obj, state)` 가 추가되었습니다.

`object.__reduce_ex__(protocol)`

또는, *__reduce_ex__()* 메서드를 정의할 수 있습니다. 유일한 차이점은 이 메서드가 프로토콜 버전인 단일 정수 인자를 받아들여야 한다는 것입니다. 정의되면, pickle은 *__reduce__()* 메서드보다 선호합니다. 또한, *__reduce__()* 는 자동으로 확장 버전의 동의어가 됩니다. 이 메서드의 주된 용도는 구형 파이썬 배포를 위해 과거 호환성 있는 환원 값을 제공하는 것입니다.

[4] *copy* 모듈은 얕거나 깊은 복사 연산에 이 프로토콜을 사용합니다.

외부 객체의 지속성

객체 지속성의 효용을 위해, *pickle* 모듈은 피클 된 데이터 스트림 밖의 객체에 대한 참조 개념을 지원합니다. 이러한 객체는 지속성 ID에 의해 참조되며, 영숫자 문자열(프로토콜 0의 경우)[5] 또는 임의의 객체(모든 최신 프로토콜의 경우)여야 합니다.

그러한 지속성 ID의 해석은 *pickle* 모듈에 의해 정의되지 않습니다; 이 해석을 피클러와 역 피클러의 사용자 정의 메서드에 위임합니다, 각각 *persistent_id()*와 *persistent_load()*.

지속성 ID를 가진 객체를 피클 하기 위해서, 피클러는 객체를 인자로 받아서 그 객체에 대해 `None` 또는 지속성 ID를 반환하는 사용자 정의 *persistent_id()* 메서드가 있어야 합니다. `None` 이 반환되면, 피클러는 단순히 객체를 피클 합니다. 지속성 ID 문자열이 반환되면, 피클러는 마커와 함께 해당 객체를 피클 하여 역 피클러가 이를 지속성 ID로 인식하게 합니다.

외부 객체를 역 피클 하려면, 역 피클러는 지속성 ID 객체를 받아들여 참조된 객체를 반환하는 사용자 정의 *persistent_load()* 메서드를 가져야 합니다.

다음은 지속성 ID를 외부 객체를 참조로 피클 하는데 사용하는 방법을 보여주는 포괄적인 예입니다.

```
# 지속성 ID를 외부 객체를 참조로 피클 하는데 사용하는 방법을 보여주는 간단한 예제.
# external objects by reference.

import pickle
import sqlite3
from collections import namedtuple

# 우리 데이터베이스의 레코드를 나타내는 간단한 클래스.
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # MemoRecord를 일반 클래스 인스턴스로 피클 하는 대신,
        # 지속성 ID를 출력합니다.
        if isinstance(obj, MemoRecord):
            # 여기서, 우리의 지속성 ID는 태그와 데이터베이스의 특정 레코드를 참조하는 키
를 포함하는
            # 단순한 튜플입니다.
            return ("MemoRecord", obj.key)
        else:
            # obj에 지속성 ID가 없으면, None을 반환합니다. 이것은 obj가 평소와 같이 피
클 되어야
            # 함을 의미합니다.
            return None


class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # 이 메서드는, 지속성 ID를 만날 때마다 호출됩니다.
        # 여기에서, pid는 DBPickler가 반환한 튜플입니다.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
```

(다음 페이지에 계속)

[5] 영숫자 문자의 제한은 프로토콜 0에서 지속성 ID가 개행 문자로 구분되기 때문입니다. 따라서 지속성 ID에 개행 문자가 포함되면 결과 피클된 데이터를 읽을 수 없게 됩니다.

(이전 페이지에서 계속)

```
            # 데이터베이스에서 참조 된 레코드를 반입하여 리턴하십시오.
            cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # 올바른 객체를 반환할 수 없으면 항상 에러를 일으켜야 합니다.
            # 그렇지 않으면, 언피클러는 None이 지속성 ID에 의해 참조되는 객체라고
            # 생각할 것입니다.
            raise pickle.UnpicklingError("unsupported persistent object")


def main():
    import io
    import pprint

    # 데이터베이스를 초기화하고 값을 채웁니다.
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)")
    tasks = (
        'give food to fish',
        'prepare group meeting',
        'fight with a zebra',
        )
    for task in tasks:
        cursor.execute("INSERT INTO memos VALUES(NULL, ?)", (task,))

    # 피클 할 레코드를 가져옵니다.
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
    # 사용자 정의 DBPickler를 사용하여 레코드를 저장합니다.
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("Pickled records:")
    pprint.pprint(memos)

    # 확인을 위해 레코드를 갱신합니다.
    cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")

    # 피클 데이터 스트림에서 레코드를 로드합니다.
    file.seek(0)
    memos = DBUnpickler(file, conn).load()

    print("Unpickled records:")
    pprint.pprint(memos)


if __name__ == '__main__':
    main()
```

디스패치 테이블

피클링에 의존하는 다른 코드를 방해하지 않고 일부 클래스의 피클링을 사용자 정의하려면, 사실 디스패치 테이블을 갖는 피클러를 만들 수 있습니다.

copyreg 모듈에 의해 관리되는 전역 디스패치 테이블은 `copyreg.dispatch_table`로 사용 가능합니다.

그러므로, 사설 디스패치 테이블로 copyreg.dispatch_table 의 수정된 복사본을 사용할 수 있습니다. 예를 들면

```
f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
```

는 SomeClass 클래스를 특별히 처리하는 사설 디스패치 테이블을 갖는 *pickle.Pickler* 의 인스턴스를 생성합니다. 또는, 코드

```
class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)
```

가 같은 일을 하지만, MyPickler 의 모든 인스턴스는 기본적으로 사설 디스패치 테이블을 공유합니다. 반면에, 다음 코드

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

는 *copyreg* 모듈의 모든 사용자가 공유하는 전역 디스패치 테이블을 수정합니다.

상태 저장 객체 처리

다음은 클래스의 피클 동작을 수정하는 방법을 보여주는 예제입니다. 아래의 TextReader 클래스는 텍스트 파일을 열고, readline() 메서드가 호출될 때마다 줄 번호와 줄 내용을 반환합니다. TextReader 인스턴스가 피클 되면, 파일 객체 멤버를 제외한 모든 어트리뷰트가 저장됩니다. 인스턴스가 역 피클 될 때, 파일이 다시 열리고, 마지막 위치에서 읽기가 다시 시작됩니다. __setstate__() 와 __getstate__() 메서드가 이 행동을 구현하는 데 사용됩니다.

```
class TextReader:
    """텍스트 파일의 줄을 인쇄하고 번호를 매깁니다."""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0

    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
        return "%i: %s" % (self.lineno, line)

    def __getstate__(self):
        # 우리의 모든 인스턴스 어트리뷰트를 포함하는 self.__dict__ 에서 객체의
        # 상태를 복사합니다. 원래 상태를 수정하지 않으려면 항상 dict.copy()
        # 메서드를 사용하십시오.
        state = self.__dict__.copy()
        # 피클 가능하지 않은 항목을 제거합니다.
        del state['file']
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        return state

    def __setstate__(self, state):
        # 인스턴스 어트리뷰트(즉, filename 과 lineno)를 복원합니다.
        self.__dict__.update(state)
        # 이전에 열린 파일의 상태를 복원합니다. 그렇게 하려면, 다시 열어서 행 수를 복원할
        # 때까지 읽어야 합니다.
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # 마지막으로, 파일을 저장합니다.
        self.file = file
```

사용 예는 다음과 같은 식입니다:

```
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```

12.1.6 형, 함수 및 기타 객체에 대한 사용자 정의 환원

Added in version 3.8.

때로, *dispatch_table*이 충분히 유연하지 않을 수 있습니다. 특히 객체의 형이 아닌 다른 기준에 따라 피클링을 사용자 정의하거나, 함수와 클래스 피클링을 사용자 정의하고 싶을 수 있습니다.

이럴 때, *Pickler* 클래스의 서브 클래스를 만들고 *reducer_override()* 메서드를 구현할 수 있습니다. 이 메서드는 임의의 환원 튜플을 반환할 수 있습니다 (*__reduce__()*를 참조하십시오). 또는 *NotImplemented*를 반환하여 전통적인 동작으로 폴백(fallback)할 수 있습니다.

*dispatch_table*과 *reducer_override()*가 모두 정의되면, *reducer_override()* 메서드가 우선합니다.

> **참고**
>
> 성능상의 이유로, 다음과 같은 객체에 대해서는 *reducer_override()*가 호출되지 않을 수 있습니다: None, True, False 및 *int*, *float*, *bytes*, *str*, *dict*, *set*, *frozenset*, *list* 및 *tuple*의 정확한(exact) 인스턴스.

다음은 주어진 클래스를 피클링하고 재구성할 수 있도록 하는 간단한 예제입니다:

```
import io
import pickle

class MyClass:
    my_attribute = 1

class MyPickler(pickle.Pickler):
    def reducer_override(self, obj):
        """MyClass를 위한 사용자 정의 환원자."""
        if getattr(obj, "__name__", None) == "MyClass":
            return type, (obj.__name__, obj.__bases__,
                          {'my_attribute': obj.my_attribute})
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        else:
            # 다른 모든 객체는, 일반적인 환원을 사용합니다
            return NotImplemented

f = io.BytesIO()
p = MyPickler(f)
p.dump(MyClass)

del MyClass

unpickled_class = pickle.loads(f.getvalue())

assert isinstance(unpickled_class, type)
assert unpickled_class.__name__ == "MyClass"
assert unpickled_class.my_attribute == 1
```

12.1.7 아웃 오브 밴드 버퍼

Added in version 3.8.

일부 상황에서는, `pickle` 모듈을 사용하여 많은 양의 데이터를 전송합니다. 따라서 성능과 자원 소비를 보존하기 위해 메모리 복사 횟수를 최소화하는 것이 중요할 수 있습니다. 그러나, `pickle` 모듈의 정상적인 작동은, 객체의 그래프(graph)적인 구조를 순차적인 바이트 스트림으로 변환하기 때문에, 본질적으로 피클 스트림과의 데이터 복사를 수반합니다.

제공자(*provider*)(전송될 객체 형의 구현)와 소비자(*consumer*)(통신 시스템의 구현)가 모두 피클 프로토콜 5 이상에서 제공되는 아웃 오브 밴드 전송 기능을 지원하면 이 제약 조건을 피할 수 있습니다.

제공자 API

피클 될 대형 데이터 객체는 프로토콜 5 이상에 특화된 `__reduce_ex__()` 메서드를 구현해야 합니다. 이 메서드는 대형 데이터에 대해 (예를 들어 `bytes` 객체 대신) `PickleBuffer` 인스턴스를 반환합니다.

`PickleBuffer` 객체는 하부 버퍼가 아웃 오브 밴드 전송 대상이라는 신호를 보냅니다. 이러한 객체는 `pickle` 모듈의 일반적인 사용과 호환됩니다. 그러나, 소비자는 `pickle`에게 그 버퍼를 스스로 처리하겠다고 알릴 수도 있습니다.

소비자 API

통신 시스템은 객체 그래프를 직렬화할 때 생성된 `PickleBuffer` 객체의 사용자 정의 처리를 활성화할 수 있습니다.

송신 측에서는, *buffer_callback* 인자를 `Pickler`(또는 `dump()`나 `dumps()` 함수)에 전달해야 합니다. 이 인자는 객체 그래프를 피클링할 때 생성된 각 `PickleBuffer`로 호출됩니다. *buffer_callback*에 의해 누적된 버퍼는 피클 스트림으로 복사되지 않고, 저렴한 마커만 삽입됩니다.

수신 측에서는, *buffer_callback*에 전달된 버퍼의 이터러블인 *buffers* 인자를 `Unpickler`(또는 `load()`나 `loads()` 함수)에 전달해야 합니다. 그 이터러블은 *buffer_callback*에 전달된 것과 같은 순서로 버퍼를 만들어야 합니다. 이러한 버퍼는 피클링이 원래 `PickleBuffer` 객체를 생성한 객체의 재구성자가 기대하는 데이터를 제공합니다.

송신 측과 수신 측 사이에서, 통신 시스템은 아웃 오브 밴드 버퍼를 위한 자체 전송 메커니즘을 자유롭게 구현할 수 있습니다. 잠재적인 최적화에는 공유 메모리나 데이터 유형에 따른 압축이 포함됩니다.

예제

다음은 아웃 오브 버퍼 피클링에 참여할 수 있는 `bytearray` 서브 클래스를 구현하는 간단한 예제입니다:

```
class ZeroCopyByteArray(bytearray):

    def __reduce_ex__(self, protocol):
        if protocol >= 5:
            return type(self)._reconstruct, (PickleBuffer(self),), None
        else:
            # PickleBuffer는 피클 프로토콜 <= 4 에서는 금지됩니다.
            return type(self)._reconstruct, (bytearray(self),)

    @classmethod
    def _reconstruct(cls, obj):
        with memoryview(obj) as m:
            # 원래 버퍼 객체에 대한 핸들을 얻습니다
            obj = m.obj
            if type(obj) is cls:
                # 원래 버퍼 객체는 ZeroCopyByteArray 이며,
                # 그대로 반환합니다.
                return obj
            else:
                return cls(obj)
```

재구성자(`_reconstruct` 클래스 메서드)는 올바른 형이면 버퍼를 제공하는 객체를 반환합니다. 이것은 이 장난감 예제에서 제로-복사 동작을 흉내 내는 손쉬운 방법입니다.

소비자 측에서는, 그 객체들을 일반적인 방법으로 피클 할 수 있습니다. 역 직렬화될 때 원래 객체의 사본을 제공합니다:

```
b = ZeroCopyByteArray(b"abc")
data = pickle.dumps(b, protocol=5)
new_b = pickle.loads(data)
print(b == new_b)  # True
print(b is new_b)  # False: 복사가 발생했습니다
```

그러나 *buffer_callback*을 전달하고 역 직렬화할 때 누적된 버퍼를 돌려주면, 원래의 객체를 다시 얻을 수 있습니다:

```
b = ZeroCopyByteArray(b"abc")
buffers = []
data = pickle.dumps(b, protocol=5, buffer_callback=buffers.append)
new_b = pickle.loads(data, buffers=buffers)
print(b == new_b)  # True
print(b is new_b)  # True: 복사가 발생하지 않았습니다
```

이 예제는 `bytearray`가 자체 메모리를 할당한다는 사실로 인해 제한됩니다: 즉, 다른 객체의 메모리를 사용하는 `bytearray` 인스턴스를 만들 수 없습니다. 그러나, NumPy 배열과 같은 제삼자 데이터형에는 이러한 제한이 없으며, 별개의 프로세스나 시스템 간에 전송할 때 제로-복사 피클링(또는 최소한의 복사)을 사용할 수 있습니다.

더 보기

PEP 574 – 아웃 오브 밴드 데이터를 포함하는 피클 프로토콜 5

12.1.8 전역 제한하기

기본적으로, 역 피클링은 피클 데이터에서 찾은 모든 클래스나 함수를 임포트 합니다. 많은 응용 프로그램에서는, 역 피클러가 임의 코드를 임포트하고 호출할 수 있으므로, 이 동작을 받아들일 수 없습니다. 이 손으로 만든 피클 데이터 스트림이 로드될 때 하는 일을 생각해보십시오:

```
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
hello world
0
```

이 예제에서, 역 피클러는 *os.system()* 함수를 임포트하고 문자열 인자 "echo hello world" 를 적용합니다. 이 예제가 공격적이지는 않지만, 어떤 것들은 시스템을 손상할 수 있다고 상상하기 어렵지 않습니다.

이런 이유로, 여러분은 *Unpickler.find_class()* 를 사용자 정의하여 언 피클 되는 것을 제어하고 싶을 수 있습니다. 이름이 제안하는 것과는 달리, *Unpickler.find_class()* 는 전역(즉, 클래스나 함수)이 요청될 때마다 호출됩니다. 따라서 전역을 완전히 금지하거나 안전한 부분집합으로 제한할 수 있습니다.

다음은 *builtins* 모듈에서 몇 가지 안전한 클래스만 로드되도록 허용하는 역 피클러의 예입니다:

```
import builtins
import io
import pickle

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        # builtins의 안전한 클래스만 허용합니다.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # 다른 모든 것을 금지합니다.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                     (module, name))

def restricted_loads(s):
    """pickle.loads()와 유사한 도움 함수."""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

우리의 역 피클러가 의도대로 작동하는 사용 예:

```
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
>>> restricted_loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
...                  b'(S\'getattr(__import__("os"), "system")'
...                  b'("echo hello world")\'\ntR.')
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

예를 통해 알 수 있듯이, 역 피클을 허락하는 것에 주의를 기울여야 합니다. 따라서 보안이 중요하다면, *xmlrpc.client* 나 제삼자 솔루션의 마샬링 API 같은 대안을 고려할 수 있습니다.

12.1.9 성능

최신 버전의 피클 프로토콜(프로토콜 2 이상)은 몇 가지 공통 기능 및 내장형에 대한 효율적인 바이너리 인코딩을 제공합니다. 또한, *pickle* 모듈은 C로 작성된 투명한 최적화기를 가지고 있습니다.

12.1.10 예제

가장 간단한 코드로, *dump()*와 *load()* 함수를 사용하십시오.

```
import pickle

# pickle이 지원하는 임의의 객체 모음.
data = {
    'a': [1, 2.0, 3+4j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # 사용 가능한 가장 높은 프로토콜을 사용하여 'data' 딕셔너리를 피클 합니다.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

다음 예제는 결과로 나온 피클 데이터를 읽습니다.

```
import pickle

with open('data.pickle', 'rb') as f:
    # 사용된 프로토콜 버전이 자동으로 감지되므로,
    # 지정할 필요가 없습니다.
    data = pickle.load(f)
```

12.1.11 Command-line interface

The *pickle* module can be invoked as a script from the command line, it will display contents of the pickle files. However, when the pickle file that you want to examine comes from an untrusted source, `-m pickletools` is a safer option because it does not execute pickle bytecode, see *pickletools CLI usage*.

```
python -m pickle pickle_file [pickle_file ...]
```

The following option is accepted:

pickle_file

A pickle file to read, or `-` to indicate reading from standard input.

더 보기

모듈 ***copyreg***
확장형에 대한 피클 인터페이스 생성자 등록

모듈 ***pickletools***
피클 된 데이터로 작업하고 분석하는 도구.

모듈 ***shelve***
객체의 인덱싱 된 데이터베이스; *pickle*을 사용합니다.

모듈 ***copy***
얕거나 깊은 객체 복사.

모듈 ***marshal***
내장형의 고성능 직렬화.

12.2 copyreg — pickle 지원 함수 등록

소스 코드: Lib/copyreg.py

---

copyreg 모듈은 특정 객체를 피클 하는 동안 사용되는 함수를 정의하는 방법을 제공합니다. pickle과 copy 모듈은 해당 객체를 피클/복사할 때 이 함수를 사용합니다. 이 모듈은 클래스가 아닌 객체 생성자에 대한 구성 정보를 제공합니다. 이러한 생성자는 팩토리 함수나 클래스 인스턴스일 수 있습니다.

copyreg.**constructor**(*object*)

*object*를 유효한 생성자로 선언합니다. *object*가 콜러블이 아니면 (따라서 생성자로 유효하지 않으면), TypeError가 발생합니다.

copyreg.**pickle**(*type*, *function*, *constructor_ob=None*)

*function*이 *type* 형의 객체에 대한 "환원" 함수로 사용되어야 한다고 선언합니다. *function*는 문자열이나 2개에서 또는 6개 사이의 요소를 포함하는 튜플을 반환해야 합니다. *function*의 인터페이스에 대한 자세한 내용은 dispatch_table을 참조하십시오.

The *constructor_ob* parameter is a legacy feature and is now ignored, but if passed it must be a callable.

피클러 객체나 pickle.Pickler의 서브 클래스의 dispatch_table 어트리뷰트도 환원 함수를 선언하는 데 사용될 수 있습니다.

12.2.1 예제

아래 예제는 피클 함수를 등록하는 방법과 사용법을 보여줍니다.:

```
>>> import copyreg, copy, pickle
>>> class C:
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

12.3 shelve — 파이썬 객체 지속성

소스 코드: Lib/shelve.py

"쉘프(shelf)"는 영속적인(persistent) 딕셔너리류 객체입니다. "dbm" 데이터베이스와의 차이점은 쉘프의 값(키가 아닙니다!)이 사실상 임의의 파이썬 객체일 수 있다는 것입니다 — pickle 모듈에서 처리할 수 있는 모든 것입니다. 여기에는 대부분의 클래스 인스턴스, 재귀적 데이터형 및 많은 공유 서브 객체를 포함하는 객체가 포함됩니다. 키는 일반 문자열입니다.

shelve.**open**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

영속적 딕셔너리를 엽니다. 지정된 filename은 하부 데이터베이스의 기본 파일명입니다. 부작용으로, 확장명이 파일명에 추가될 수 있으며 여러 개의 파일이 만들어질 수 있습니다. 기본적으로, 하부 데이터베이스 파일은 읽기와 쓰기 용으로 열립니다. 선택적 *flag* 매개 변수는 dbm.open()의 *flag* 매개 변수와 같게 해석됩니다.

기본적으로, 값을 직렬화하는 데 `pickle.DEFAULT_PROTOCOL`로 생성된 피클이 사용됩니다. 피클 프로토콜의 버전은 *protocol* 매개 변수로 지정할 수 있습니다.

파이썬 의미론 때문에, 쉘프는 가변 영속 딕셔너리 항목이 언제 수정되는지 알 수 없습니다. 기본적으로 수정된 객체는 쉘프에 대입될 때만 기록됩니다 (예제를 참조하십시오). 선택적인 *writeback* 매개 변수가 `True`로 설정되면, 액세스 된 모든 항목도 메모리에 캐시 되고, `sync()`와 `close()`가 호출될 때 다시 기록됩니다; 이것은 영속 딕셔너리의 가변 항목을 변경하는 것을 더 수월하게 만들지만, 많은 항목이 액세스 되면, 캐시를 위해 막대한 양의 메모리를 소비할 수 있으며, 액세스 된 모든 항목을 다시 기록하기 때문에 닫기 연산이 매우 느려질 수 있습니다 (어떤 액세스 된 항목이 가변인지, 어떤 것이 실제로 변경되었는지를 판별할 방법이 없습니다).

버전 3.10에서 변경: `pickle.DEFAULT_PROTOCOL` is now used as the default pickle protocol.

버전 3.11에서 변경: Accepts *path-like object* for filename.

참고

쉘프가 자동으로 닫히는 것에 의지하지 마십시오; 더는 필요 없을 때 `close()`를 명시적으로 호출하거나, `shelve.open()`을 컨텍스트 관리자로 사용하십시오:

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

경고

`shelve` 모듈은 `pickle`로 뒷받침되기 때문에, 신뢰할 수 없는 소스에서 쉘프를 로드하는 것은 안전하지 않습니다. 피클과 마찬가지로, 쉘프를 로드하면 임의의 코드를 실행할 수 있습니다.

쉘프 객체는 딕셔너리에서 지원하는 대부분의 메서드와 연산을 지원합니다 (복사, 생성자 및 연산자 `|`와 `|=`는 제외합니다). 이것은 딕셔너리 기반 스크립트에서 영속적인 저장소를 요구하는 것으로의 전환을 쉽게 만듭니다.

두 가지 추가 메서드가 지원됩니다:

`Shelf.sync()`

*writeback*을 `True`로 설정하여 쉘프를 열었으면, 캐시의 모든 항목을 다시 기록합니다. 또한, 적절하다면, 캐시를 비우고 디스크 상의 영속 딕셔너리를 동기화합니다. `close()`로 쉘프를 닫을 때 자동으로 호출됩니다.

`Shelf.close()`

영구 딕셔너리 객체를 동기화하고 닫습니다. 닫힌 쉘프에 대한 연산은 `ValueError`로 실패합니다.

더 보기

널리 지원되는 저장 형식과 기본 딕셔너리의 속도를 갖춘 Persistent dictionary recipe

12.3.1 제약 사항

- 사용되는 데이터베이스 패키지의 선택(가령 `dbm.ndbm`이나 `dbm.gnu`)은 어떤 인터페이스가 사용 가능한지에 따라 다릅니다. 따라서 `dbm`을 사용하여 데이터베이스를 직접 여는 것은 안전하지 않습니다. 또한, 데이터베이스는 (불행히도) `dbm`이 사용된다면 그것의 제약이 적용됩니다 — 이것은 데이터베이스에 저장되는 객체(의 피클 된 표현이)가 상당히 작아야 하며, 드물긴 하지만 키 충돌로 인해 데이터베이스가 업데이트를 거부할 수 있음을 뜻합니다.
- `shelve` 모듈은 쉘브된 객체에 대한 동시성(*concurrent*) 읽기/쓰기 액세스를 지원하지 않습니다. (여러 동시적인 읽기 액세스는 안전합니다.) 어떤 프로그램이 쓰기 용으로 쉘프를 열고 있으면, 다른 어떤 프로그램도 읽기나 쓰기 용으로 열지 않아야 합니다. 유닉스 파일 잠금을 이 문제를 해결하는 데 사용할 수 있지만, 이것은 유닉스 버전마다 다르며 사용된 데이터베이스 구현에 대한 지식이 필요합니다.

- On macOS `dbm.ndbm` can silently corrupt the database file on updates, which can cause hard crashes when trying to read from the database.

class shelve.**Shelf**(*dict, protocol=None, writeback=False, keyencoding='utf-8'*)

dict 객체에 피클 된 값을 저장하는 `collections.abc.MutableMapping`의 서브 클래스.

기본적으로, 값을 직렬화하는 데 `pickle.DEFAULT_PROTOCOL`로 생성된 피클이 사용됩니다. 피클 프로토콜의 버전은 *protocol* 매개 변수로 지정할 수 있습니다. 피클 프로토콜에 대한 설명은 `pickle` 설명서를 참조하십시오.

writeback 매개 변수가 `True`이면, 객체는 액세스 된 모든 항목의 캐시를 보유하고 sync와 close 할 때 *dict*에 다시 씁니다. 이것은 가변 항목에 대한 자연스러운 연산을 허락하지만, 더 많은 메모리를 소비하고 sync와 close 연산이 오래 걸릴 수 있습니다.

keyencoding 매개 변수는 하부 dict에 사용되기 전에 키를 인코딩하는 데 사용되는 인코딩입니다.

`Shelf` 객체는 컨텍스트 관리자로 사용할 수도 있습니다. 이 경우 `with` 블록이 끝날 때 자동으로 닫힙니다.

버전 3.2에서 변경: *keyencoding* 매개 변수가 추가되었습니다; 이전에는 키가 항상 UTF-8으로 인코딩되었습니다.

버전 3.4에서 변경: 컨텍스트 관리자 지원 추가.

버전 3.10에서 변경: `pickle.DEFAULT_PROTOCOL` is now used as the default pickle protocol.

class shelve.**BsdDbShelf**(*dict, protocol=None, writeback=False, keyencoding='utf-8'*)

`first()`, `next()`, `previous()`, `last()` 및 `set_location()` 메서드를 노출하는 `Shelf`의 서브 클래스. pybsddb의 제삼자 `bsddb` 모듈에서는 사용할 수 있지만 다른 데이터베이스 모듈에서는 사용할 수 없습니다. 생성자에 전달된 *dict* 객체는 이러한 메서드를 지원해야 합니다. 이것은 일반적으로 `bsddb.hashopen()`, `bsddb.btopen()` 또는 `bsddb.rnopen()` 중 하나를 호출하여 수행됩니다. 선택적 *protocol*, *writeback* 및 *keyencoding* 매개 변수는 `Shelf` 클래스와 같게 해석됩니다.

class shelve.**DbfilenameShelf**(*filename, flag='c', protocol=None, writeback=False*)

딕셔너리류 객체 대신에 *filename*을 받아들이는 `Shelf`의 서브 클래스. 하부 파일은 `dbm.open()`을 사용하여 열립니다. 기본적으로, 파일은 읽기와 쓰기가 가능하도록 만들어지고 열립니다. 선택적 *flag* 매개 변수는 `open()` 기능과 같게 해석됩니다. 선택적 *protocol*과 *writeback* 매개 변수는 `Shelf` 클래스와 같게 해석됩니다.

12.3.2 예제

인터페이스를 요약하면 (`key`는 문자열입니다, `data`는 임의의 객체입니다):

```
import shelve

d = shelve.open(filename)  # 열기 -- 파일은 저수준 라이브러리에 의해 추가된 접미사를 얻을
                           # 수 있습니다

d[key] = data              # 키에 데이터를 저장합니다 (기존 키를 사용하면 예전 데이터를
                           # 덮어씁니다)
data = d[key]              # 키에서 데이터의 사본을 꺼냅니다 (그런 키가 없으면 KeyError를
                           # 발생시킵니다)
del d[key]                 # 키에 저장된 데이터를 삭제합니다 (그런 키가 없으면 KeyError를
                           # 발생시킵니다)

flag = key in d            # 키가 존재하면 참입니다
klist = list(d.keys())     # 존재하는 모든 키의 리스트 (느립니다!)

# d가 writeback=True 없이 열렸으므로, 주의하십시오:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
d['xx'] = [0, 1, 2]          # 이것은 예상대로 작동합니다, 하지만...
d['xx'].append(3)            # *이것은 그렇지 않습니다!* -- d['xx']는 여전히 [0, 1, 2]입
니다!

# writeback=True 없이 열린 d에서는, 신중하게 코딩해야 합니다:
temp = d['xx']               # 사본을 꺼냅니다
temp.append(5)               # 사본을 수정합니다
d['xx'] = temp               # 영속적으로 만들기 위해, 사본을 다시 저장합니다

# 또는, d=shelve.open(filename,writeback=True)는 단지 d['xx'].append(5)로
# 코딩해도 기대한 대로 동작하게 만들지만, 더 많은 메모리를 소비하고 d.close() 연산을 더
# 느리게 만듭니다.

d.close()                    # 닫습니다
```

더 보기

모듈 *dbm*
dbm 스타일 데이터베이스에 대한 범용 인터페이스.

모듈 *pickle*
*shelve*에 의해 사용되는 객체 직렬화.

12.4 marshal — 내부 파이썬 객체 직렬화

이 모듈은 바이너리 형식으로 파이썬 값을 읽고 쓸 수 있는 함수를 포함합니다. 형식은 파이썬에만 국한되지만, 기계 아키텍처 문제에는 독립적입니다 (예를 들어, 파이썬 값을 PC의 파일에 기록하고 파일을 맥으로 보낸 다음, 그곳에서 다시 읽을 수 있습니다). 형식의 세부 사항은 의도적으로 설명하지 않습니다; 파이썬 버전 간에 변경될 수 있습니다 (드물긴 하지만).[1]

이것은 범용 "지속성" 모듈이 아닙니다. 범용 지속성과 RPC 호출을 통한 파이썬 객체의 전송에 대해서는, *pickle*과 *shelve* 모듈을 참조하십시오. *marshal* 모듈은 주로 .pyc 파일의 파이썬 모듈에 대한 "의사 컴파일된" 코드 읽기와 쓰기를 지원하기 위해 존재합니다. 따라서, 파이썬 관리자는 필요에 따라 이전 버전과 호환되지 않는 방식으로 마샬 형식을 수정할 수 있는 권한을 갖습니다. 코드 객체의 형식은 형식의 버전이 같더라도 파이썬 버전 간에 호환되지 않습니다. 잘못된 파이썬 버전에서 코드 객체를 역 직렬화하면 정의되지 않은 동작이 발생합니다. 파이썬 객체를 직렬화하고 역 직렬화하는 데는, 대신 *pickle* 모듈을 사용하십시오 – 성능은 비슷하고, 버전 독립성이 보장되며, pickle은 marshal보다 훨씬 넓은 범위의 객체를 지원합니다.

경고

marshal 모듈은 잘못되었거나 악의적으로 구성된 데이터에 대해 보안성을 갖추려는 것이 아닙니다. 신뢰할 수 없거나 인증되지 않은 출처에서 받은 데이터를 역 마샬 하지 마십시오.

파일을 읽고 쓰는 함수는 물론 바이트열류 객체에서 작동하는 함수도 있습니다.

Not all Python object types are supported; in general, only objects whose value is independent from a particular invocation of Python can be written and read by this module. The following types are supported:

- Numeric types: *int*, *bool*, *float*, *complex*.

[1] 이 모듈의 이름은 (다른 것 중에서도) Modula-3의 설계자가 사용하는 약간의 용어에서 유래합니다. 이들은 자급적(self-contained) 형식으로 데이터를 전달하는 데 "마샬링(marshalling)" 이라는 용어를 사용합니다. 엄밀히 말하면, "마샬" 은 내부의 어떤 데이터를 외부 형식(예를 들어 RPC 버퍼에)으로 변환하는 것을, "역 마샬" 은 그 반대 절차를 뜻합니다.

- Strings (`str`) and `bytes`. *Bytes-like objects* like `bytearray` are marshalled as `bytes`.
- Containers: `tuple`, `list`, `set`, `frozenset`, and (since `version` 5), `slice`. It should be understood that these are supported only if the values contained therein are themselves supported. Recursive containers are supported since `version` 3.
- The singletons `None`, `Ellipsis` and `StopIteration`.
- `code` objects, if *allow_code* is true. See note above about version dependence.

버전 3.4에서 변경:

- Added format version 3, which supports marshalling recursive lists, sets and dictionaries.
- Added format version 4, which supports efficient representations of short strings.

버전 3.14에서 변경: Added format version 5, which allows marshalling slices.

모듈은 다음 함수를 정의합니다:

`marshal.`**`dump`**(*value, file, version=version, /, *, allow_code=True*)

열린 파일에 값을 기록합니다. value는 지원되는 형이어야 합니다. 파일은 쓰기 가능한 바이너리 파일이어야 합니다.

value가 지원되지 않는 형이면 (또는 지원되지 않는 형의 객체를 담고 있다면) `ValueError` 예외가 발생합니다 — 하지만, 찌꺼기 데이터도 파일에 기록됩니다. `load()`로 객체를 제대로 읽을 수 없습니다. 코드 객체는 *allow_code*가 참일 때만 지원됩니다.

version 인자는 `dump`가 사용해야 하는 데이터 형식을 나타냅니다 (아래를 참조하십시오).

Raises an *auditing event* `marshal.dumps` with arguments `value`, `version`.

버전 3.13에서 변경: Added the *allow_code* parameter.

`marshal.`**`load`**(*file, /, *, allow_code=True*)

열린 파일에서 하나의 값을 읽고 그것을 반환합니다. 유효한 값을 읽히지 않으면 (예를 들어, 데이터가 다른 파이썬 버전의 호환되지 않는 마샬 형식이라서) `EOFError`, `ValueError` 또는 `TypeError`를 발생시킵니다. 코드 객체는 *allow_code*가 참일 때만 지원됩니다. 파일은 읽을 수 있는 바이너리 파일이어야 합니다.

Raises an *auditing event* `marshal.load` with no arguments.

> **참고**
>
> 지원하지 않는 형을 포함하는 객체가 `dump()`로 마샬 되었으면, `load()`는 역 마샬이 불가능한 형을 `None`으로 치환합니다.

버전 3.10에서 변경: This call used to raise a `code.__new__` audit event for each code object. Now it raises a single `marshal.load` event for the entire load operation.

버전 3.13에서 변경: Added the *allow_code* parameter.

`marshal.`**`dumps`**(*value, version=version, /, *, allow_code=True*)

`dump(value, file)`에 의해 파일에 기록될 바이트열 객체를 반환합니다. value는 지원되는 형이어야 합니다. value가 지원되지 않는 형이면 (또는 지원되지 않는 형의 객체를 담고 있다면) `ValueError` 예외를 발생시킵니다. 코드 객체는 *allow_code*가 참일 때만 지원됩니다.

version 인자는 `dumps`가 사용해야 하는 데이터 형식을 나타냅니다 (아래를 참조하십시오).

Raises an *auditing event* `marshal.dumps` with arguments `value`, `version`.

버전 3.13에서 변경: Added the *allow_code* parameter.

`marshal.`**`loads`**(*bytes, /, *, allow_code=True*)

바이트열류 객체를 값으로 변환합니다. 유효한 값이 없으면 `EOFError`, `ValueError` 또는 `TypeError`를 발생시킵니다. 코드 객체는 *allow_code*가 참일 때만 지원됩니다. 입력의 여분의 바이트는 무시됩니다.

Raises an *auditing event* `marshal.loads` with argument `bytes`.

버전 3.10에서 변경: This call used to raise a `code.__new__` audit event for each code object. Now it raises a single `marshal.loads` event for the entire load operation.

버전 3.13에서 변경: Added the *allow_code* parameter.

또한, 다음 상수가 정의됩니다:

`marshal.`**`version`**

Indicates the format that the module uses. Version 0 is the historical first version; subsequent versions add new features. Generally, a new version becomes the default when it is introduced.

Version	Available since	New features
1	Python 2.4	Sharing interned strings
2	Python 2.5	Binary representation of floats
3	Python 3.4	Support for object instancing and recursion
4	Python 3.4	Efficient representation of short strings
5	Python 3.14	Support for `slice` objects

12.5 dbm — 유닉스 "데이터베이스" 인터페이스

소스 코드: Lib/dbm/__init__.py

---

*dbm*은 DBM 데이터베이스의 변형에 대한 일반 인터페이스입니다:

- *dbm.sqlite3*
- *dbm.gnu*
- *dbm.ndbm*

이러한 모듈이 설치되어 있지 않으면, *dbm.dumb* 모듈에 있는 느리지만 간단한 구현이 사용됩니다. 오라클 Berkeley DB에 대한 제삼자 인터페이스가 있습니다.

exception `dbm.`**`error`**

지원되는 각 모듈에 의해 발생할 수 있는 예외를 포함하는 튜플. 역시 *dbm.error*라고 이름 붙인 고유한 예외를 첫 번째 항목으로 갖고 있습니다 — *dbm.error*가 발생할 때 이것이 사용됩니다.

`dbm.`**`whichdb`**(*filename*)

이 함수는 사용 가능한 몇 가지 간단한 데이터베이스 모듈 — *dbm.sqlite3*, *dbm.gnu*, *dbm.ndbm* 또는 *dbm.dumb* — 중 어느 것을 사용하여 주어진 파일을 열어야 하는지 추측합니다.

다음 값 중 하나를 반환합니다:

- 읽을 수 없거나 존재하지 않아 파일을 열 수 없으면 `None`
- 파일 형식을 추측할 수 없으면 빈 문자열(`''`)
- 필요한 모듈 이름을 포함하는 문자열, 가령 `'dbm.ndbm'`이나 `'dbm.gnu'`

버전 3.11에서 변경: *filename* accepts a *path-like object*.

`dbm.`**`open`**(*file*, *flag='r'*, *mode=0o666*)

데이터베이스를 열고 해당 데이터베이스 객체를 반환합니다.

매개변수

- **file** (*path-like object*) – 열 데이터베이스 파일. 데이터베이스 파일이 이미 존재하면, *whichdb()* 함수를 사용하여 유형을 판별하고 적절한 모듈이 사용됩니다; 존재하지 않으면, 위에 나열된 것 중 임포트 할 수 있는 첫 번째 서브 모듈이 사용됩니다.

- **flag** (str) –
  - 'r' (기본값): Open existing database for reading only.
  - 'w': Open existing database for reading and writing.
  - 'c': Open database for reading and writing, creating it if it doesn' t exist.
  - 'n': Always create a new, empty database, open for reading and writing.
- **mode** (int) – The Unix file access mode of the file (default: octal 0o666), used only when the database has to be created.

버전 3.11에서 변경: *file* accepts a *path-like object*.

open() 이 반환한 객체는 *dict*와 같은 기본 기능을 지원합니다; 키와 해당 값을 저장, 조회 및 삭제할 수 있으며, get() 과 setdefault() 메서드뿐만 아니라 in 연산자와 keys() 메서드도 사용할 수 있습니다.

키와 값은 항상 *bytes*로 저장됩니다. 이는 문자열이 사용될 때 저장되기 전에 기본 인코딩으로 묵시적으로 변환됨을 의미합니다.

이 객체는 with 문에서도 사용되도록 지원해서, 완료될 때 자동으로 닫힙니다.

버전 3.2에서 변경: 이제 모든 *dbm* 백엔드에서 get() 과 setdefault() 메서드를 사용할 수 있습니다.

버전 3.4에서 변경: *open()* 이 반환한 객체에 컨텍스트 관리 프로토콜에 대한 기본 지원을 추가했습니다.

버전 3.8에서 변경: 읽기 전용 데이터베이스에서 키를 삭제하면 *KeyError* 대신 데이터베이스 모듈 특정 예외가 발생합니다.

다음 예제는 일부 호스트 명과 해당 제목을 기록한 다음, 데이터베이스의 내용을 인쇄합니다:

```
import dbm

# 데이터베이스를 엽니다, 필요하면 만듭니다.
with dbm.open('cache', 'c') as db:

    # 어떤 값을 저장합니다
    db[b'hello'] = b'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # 키는 이제 바이트열로 간주함에 유의하십시오.
    assert db[b'www.python.org'] == b'Python Website'
    # 값이 바이트열임에 주목하십시오.
    assert db['www.cnn.com'] == b'Cable News Network'

    # 종종 사용되는 dict 인터페이스의 메서드도 작동합니다.
    print(db.get('python.org', b'not present'))

    # 문자열이 아닌 키나 값을 저장하면 예외(대부분 TypeError)가
    # 발생합니다.
    db['www.yahoo.com'] = 4

# with 문을 떠날 때 db가 자동으로 닫힙니다.
```

더 보기

모듈 *shelve*
문자열이 아닌 데이터를 저장하는 지속성 모듈.

개별 서브 모듈은 다음 섹션에서 설명합니다.

12.5.1 dbm.sqlite3 — dbm 의 SQLite 백엔드

Added in version 3.13.

소스 코드: Lib/dbm/sqlite3.py

This module uses the standard library *sqlite3* module to provide an SQLite backend for the *dbm* module. The files created by *dbm.sqlite3* can thus be opened by *sqlite3*, or any other SQLite browser, including the SQLite CLI.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

dbm.sqlite3.**open**(*filename*, /, *flag='r'*, *mode=0o666*)

Open an SQLite database. The returned object behaves like a *mapping*, implements a `close()` method, and supports a "closing" context manager via the `with` keyword.

매개변수

- **filename** (*path-like object*) – The path to the database to be opened.
- **flag** (`str`) –
  - `'r'` (기본값): Open existing database for reading only.
  - `'w'`: Open existing database for reading and writing.
  - `'c'`: Open database for reading and writing, creating it if it doesn' t exist.
  - `'n'`: Always create a new, empty database, open for reading and writing.
- **mode** – 파일의 유닉스 파일 액세스 모드 (기본값: 8진수 `0o666`), 데이터베이스를 만들어야 할 때만 사용됩니다.

12.5.2 dbm.gnu — GNU 데이터베이스 관리자

소스 코드: Lib/dbm/gnu.py

The *dbm.gnu* module provides an interface to the GDBM (GNU dbm) library, similar to the *dbm.ndbm* module, but with additional functionality like crash tolerance.

> **참고**
>
> The file formats created by *dbm.gnu* and *dbm.ndbm* are incompatible and can not be used interchangeably.

가용성: not Android, not iOS, not WASI.

이 모듈은 모바일 플랫폼이나 웹어셈블리 플랫폼에서 지원되지 않습니다.

exception dbm.gnu.**error**

I/O 에러와 같은 *dbm.gnu* 특정 에러에서 발생합니다. 잘못된 키 지정과 같은 일반적인 매핑 에러에 대해서는 *KeyError*가 발생합니다.

dbm.gnu.**open**(*filename*, *flag='r'*, *mode=0o666*, /)

GDBM 데이터베이스를 열고 `gdbm` 객체를 반환합니다.

매개변수

- **filename** (*path-like object*) – 열 데이터베이스 파일.
- **flag** (`str`) –
  - `'r'` (기본값): Open existing database for reading only.

- `'w'`: Open existing database for reading and writing.
- `'c'`: Open database for reading and writing, creating it if it doesn' t exist.
- `'n'`: Always create a new, empty database, open for reading and writing.

데이터베이스를 여는 방법을 제어하기 위해 다음과 같은 추가 문자가 추가될 수 있습니다:

- `'f'`: 데이터베이스를 빠른 모드로 엽니다. 데이터베이스로의 쓰기는 동기화되지 않습니다.
- `'s'`: 동기화 모드. 데이터베이스 변경 사항이 파일에 즉시 기록됩니다.
- `'u'`: 데이터베이스를 잠그지 않습니다.

모든 플래그가 모든 버전의 GDBM에서 유효한 것은 아닙니다. 지원되는 플래그 문자의 목록은 `open_flags` 멤버를 참조하세요.

- **mode** (`int`) – The Unix file access mode of the file (default: octal `0o666`), used only when the database has to be created.

예외 발생

error – 유효하지 않은 *flag* 인자가 전달될 때.

버전 3.11에서 변경: *filename* accepts a *path-like object*.

`dbm.gnu.`**`open_flags`**

A string of characters the *flag* parameter of `open()` supports.

`gdbm` objects behave similar to *mappings*, but `items()` and `values()` methods are not supported. The following methods are also provided:

`gdbm.`**`firstkey`**`()`

이 메서드와 `nextkey()` 메서드를 사용하여 데이터베이스의 모든 키를 순회할 수 있습니다. 순회는 GDBM의 내부 해시값 순이며, 키의 값으로 정렬되지 않습니다. 이 메서드는 시작 키를 반환합니다.

`gdbm.`**`nextkey`**`(`*key*`)`

순회에서 *key* 뒤에 오는 키를 반환합니다. 다음 코드는 메모리에 모든 키를 포함하는 리스트를 만들지 않고, 데이터베이스 `db`의 모든 키를 인쇄합니다:

```
k = db.firstkey()
while k is not None:
    print(k)
    k = db.nextkey(k)
```

`gdbm.`**`reorganize`**`()`

많은 삭제를 수행한 후에, GDBM 파일이 사용하는 공간을 줄이려면, 이 루틴이 데이터베이스를 재구성합니다. `gdbm` 객체는 이 재구성을 사용할 때 외에는 데이터베이스 파일의 길이를 줄이지 않습니다; 그렇지 않으면, 삭제된 파일 공간이 유지되고 새 (키, 값) 쌍이 추가될 때 재사용됩니다.

`gdbm.`**`sync`**`()`

데이터베이스가 빠른 모드로 열렸을 때, 이 메서드를 사용하면 기록되지 않은 데이터가 디스크에 기록됩니다.

`gdbm.`**`close`**`()`

GDBM 데이터베이스를 닫습니다.

`gdbm.`**`clear`**`()`

GDBM 데이터베이스에서 모든 항목을 삭제합니다.

Added in version 3.13.

12.5.3 dbm.ndbm — 새 데이터베이스 관리자(New Database Manager)

소스 코드: Lib/dbm/ndbm.py

---

dbm.ndbm 모듈은 NDBM (New Database Manager) 라이브러리에 대한 인터페이스를 제공합니다. 이 모듈은 "고전적인" NDBM 인터페이스나 GDBM 호환 인터페이스와 함께 사용할 수 있습니다.

> **참고**
>
> The file formats created by *dbm.gnu* and *dbm.ndbm* are incompatible and can not be used interchangeably.

> **경고**
>
> The NDBM library shipped as part of macOS has an undocumented limitation on the size of values, which can result in corrupted database files when storing values larger than this limit. Reading such corrupted files can result in a hard crash (segmentation fault).

가용성: not Android, not iOS, not WASI.

이 모듈은 모바일 플랫폼이나 웹어셈블리 플랫폼에서 지원되지 않습니다.

exception dbm.ndbm.**error**

I/O 에러와 같은 *dbm.ndbm* 특정 에러에서 발생합니다. 잘못된 키 지정과 같은 일반적인 매핑 에러에 대해서는 *KeyError*가 발생합니다.

dbm.ndbm.**library**

사용된 NDBM 구현 라이브러리의 이름.

dbm.ndbm.**open**(*filename*, *flag='r'*, *mode=0o666*, /)

NDBM 데이터베이스를 열고 ndbm 객체를 반환합니다.

매개변수

- **filename** (*path-like object*) – 데이터베이스 파일의 (.dir이나 .pag 확장자가 제거된) 이름(basename) 입니다.
- **flag** (str) –
  - 'r' (기본값): Open existing database for reading only.
  - 'w': Open existing database for reading and writing.
  - 'c': Open database for reading and writing, creating it if it doesn' t exist.
  - 'n': Always create a new, empty database, open for reading and writing.
- **mode** (int) – The Unix file access mode of the file (default: octal 0o666), used only when the database has to be created.

ndbm objects behave similar to *mappings*, but items() and values() methods are not supported. The following methods are also provided:

버전 3.11에서 변경: Accepts *path-like object* for filename.

ndbm.**close**()

NDBM 데이터베이스를 닫습니다.

ndbm.**clear**()

NDBM 데이터베이스에서 모든 항목을 삭제합니다.

Added in version 3.13.

12.5.4 dbm.dumb — 이식성 있는 DBM 구현

소스 코드: Lib/dbm/dumb.py

> **참고**
>
> *dbm.dumb* 모듈은 더욱 강인한 모듈을 사용할 수 없을 때 *dbm* 모듈에 대한 최후의 대체 폴백으로 사용됩니다. *dbm.dumb* 모듈은 속도를 위해 작성되지 않았으며 다른 데이터베이스 모듈만큼 많이 사용되지는 않습니다.

---

dbm.dumb 모듈은 완전히 파이썬으로 작성된 지속적인(persistent) *dict*와 유사한 인터페이스를 제공합니다. *dbm.gnu*와 같은 다른 *dbm* 백엔드와 달리, 외부 라이브러리가 필요하지 않습니다.

dbm.dumb 모듈은 다음과 같은 것들을 정의합니다:

exception dbm.dumb.**error**

I/O 에러와 같은 *dbm.dumb* 특정 에러에서 발생합니다. 잘못된 키 지정과 같은 일반적인 매핑 에러에 대해서는 *KeyError*가 발생합니다.

dbm.dumb.**open**(*filename*, *flag='c'*, *mode=0o666*)

Open a dbm.dumb database. The returned database object behaves similar to a *mapping*, in addition to providing *sync()* and *close()* methods.

매개변수

- **filename** – The basename of the database file (without extensions). A new database creates the following files:
  - *filename*.dat
  - *filename*.dir
- **flag** (str) –
  - 'r': Open existing database for reading only.
  - 'w': Open existing database for reading and writing.
  - 'c' (기본값): Open database for reading and writing, creating it if it doesn' t exist.
  - 'n': Always create a new, empty database, open for reading and writing.
- **mode** (int) – The Unix file access mode of the file (default: octal 0o666), used only when the database has to be created.

> **경고**
>
> 파이썬 AST 컴파일러의 스택 깊이 제한으로 인해, 충분히 큰/복잡한 항목이 있는 데이터베이스를 로드할 때 파이썬 인터프리터가 충돌할 수 있습니다.

버전 3.5에서 변경: *flag*이 'n' 일 때, *open()*은 항상 새 데이터베이스를 만듭니다.

버전 3.8에서 변경: A database opened read-only if *flag* is 'r'. A database is not created if it does not exist if *flag* is 'r' or 'w'.

버전 3.11에서 변경: *filename* accepts a *path-like object*.

collections.abc.MutableMapping 클래스가 제공하는 메서드 외에도, 다음 메서드를 제공합니다:

dumbdbm.**sync**()

디스크 상의 디렉터리와 데이터 파일을 동기화합니다. 이 메서드는 *shelve.Shelf.sync()* 메서드에 의해 호출됩니다.

dumbdbm.**close**()

데이터베이스를 닫습니다.

12.6 sqlite3 — DB-API 2.0 interface for SQLite databases

소스 코드: Lib/sqlite3/ SQLite는 별도의 서버 프로세스가 필요 없고 SQL 질의 언어의 비표준 변형을 사용하여 데이터베이스에 액세스할 수 있는 경량 디스크 기반 데이터베이스를 제공하는 C 라이브러리입니다. 일부 응용 프로그램은 내부 데이터 저장을 위해 SQLite를 사용할 수 있습니다. SQLite를 사용하여 응용 프로그램을 프로토타입 한 다음 PostgreSQL 이나 Oracle과 같은 더 큰 데이터베이스로 코드를 이식할 수도 있습니다.

The `sqlite3` module was written by Gerhard Häring. It provides an SQL interface compliant with the DB-API 2.0 specification described by **PEP 249**, and requires SQLite 3.15.2 or newer.

This document includes four main sections:

- *Tutorial* teaches how to use the `sqlite3` module.
- *Reference* describes the classes and functions this module defines.
- *How-to guides* details how to handle specific tasks.
- *Explanation* provides in-depth background on transaction control.

더 보기

https://www.sqlite.org
SQLite 웹 페이지; 설명서는 지원되는 SQL 언어에 대한 문법과 사용 가능한 데이터형을 설명합니다.

https://www.w3schools.com/sql/
SQL 문법 학습을 위한 자습서, 레퍼런스 및 예제

PEP 249 - 데이터베이스 API 명세 2.0
Marc-André Lemburg가 작성한 PEP.

12.6.1 Tutorial

In this tutorial, you will create a database of Monty Python movies using basic `sqlite3` functionality. It assumes a fundamental understanding of database concepts, including cursors and transactions.

First, we need to create a new database and open a database connection to allow `sqlite3` to work with it. Call *sqlite3.connect()* to create a connection to the database `tutorial.db` in the current working directory, implicitly creating it if it does not exist:

```
import sqlite3
con = sqlite3.connect("tutorial.db")
```

The returned *Connection* object `con` represents the connection to the on-disk database.

In order to execute SQL statements and fetch results from SQL queries, we will need to use a database cursor. Call *con.cursor()* to create the *Cursor*:

```
cur = con.cursor()
```

Now that we' ve got a database connection and a cursor, we can create a database table `movie` with columns for title, release year, and review score. For simplicity, we can just use column names in the table declaration – thanks to the

flexible typing feature of SQLite, specifying the data types is optional. Execute the `CREATE TABLE` statement by calling *cur.execute(...)*:

```
cur.execute("CREATE TABLE movie(title, year, score)")
```

We can verify that the new table has been created by querying the `sqlite_master` table built-in to SQLite, which should now contain an entry for the `movie` table definition (see The Schema Table for details). Execute that query by calling *cur.execute(...)*, assign the result to `res`, and call *res.fetchone()* to fetch the resulting row:

```
>>> res = cur.execute("SELECT name FROM sqlite_master")
>>> res.fetchone()
('movie',)
```

We can see that the table has been created, as the query returns a *tuple* containing the table' s name. If we query `sqlite_master` for a non-existent table `spam`, `res.fetchone()` will return `None`:

```
>>> res = cur.execute("SELECT name FROM sqlite_master WHERE name='spam'")
>>> res.fetchone() is None
True
```

Now, add two rows of data supplied as SQL literals by executing an `INSERT` statement, once again by calling *cur.execute(...)*:

```
cur.execute("""
    INSERT INTO movie VALUES
        ('Monty Python and the Holy Grail', 1975, 8.2),
        ('And Now for Something Completely Different', 1971, 7.5)
""")
```

The `INSERT` statement implicitly opens a transaction, which needs to be committed before changes are saved in the database (see *Transaction control* for details). Call *con.commit()* on the connection object to commit the transaction:

```
con.commit()
```

We can verify that the data was inserted correctly by executing a `SELECT` query. Use the now-familiar *cur.execute(...)* to assign the result to `res`, and call *res.fetchall()* to return all resulting rows:

```
>>> res = cur.execute("SELECT score FROM movie")
>>> res.fetchall()
[(8.2,), (7.5,)]
```

The result is a *list* of two `tuple`s, one per row, each containing that row' s `score` value.

Now, insert three more rows by calling *cur.executemany(...)*:

```
data = [
    ("Monty Python Live at the Hollywood Bowl", 1982, 7.9),
    ("Monty Python's The Meaning of Life", 1983, 7.5),
    ("Monty Python's Life of Brian", 1979, 8.0),
]
cur.executemany("INSERT INTO movie VALUES(?, ?, ?)", data)
con.commit()  # Remember to commit the transaction after executing INSERT.
```

Notice that `?` placeholders are used to bind `data` to the query. Always use placeholders instead of string formatting to bind Python values to SQL statements, to avoid SQL injection attacks (see *How to use placeholders to bind values in SQL queries* for more details).

We can verify that the new rows were inserted by executing a `SELECT` query, this time iterating over the results of the query:

```
>>> for row in cur.execute("SELECT year, title FROM movie ORDER BY year"):
...     print(row)
(1971, 'And Now for Something Completely Different')
(1975, 'Monty Python and the Holy Grail')
(1979, "Monty Python's Life of Brian")
(1982, 'Monty Python Live at the Hollywood Bowl')
(1983, "Monty Python's The Meaning of Life")
```

Each row is a two-item *tuple* of `(year, title)`, matching the columns selected in the query.

Finally, verify that the database has been written to disk by calling *con.close()* to close the existing connection, opening a new one, creating a new cursor, then querying the database:

```
>>> con.close()
>>> new_con = sqlite3.connect("tutorial.db")
>>> new_cur = new_con.cursor()
>>> res = new_cur.execute("SELECT title, year FROM movie ORDER BY score DESC")
>>> title, year = res.fetchone()
>>> print(f'The highest scoring Monty Python movie is {title!r}, released in {year}
↪')
The highest scoring Monty Python movie is 'Monty Python and the Holy Grail',␣
↪released in 1975
>>> new_con.close()
```

You' ve now created an SQLite database using the `sqlite3` module, inserted data and retrieved values from it in multiple ways.

더 보기

- *How-to guides* for further reading:
  - *How to use placeholders to bind values in SQL queries*
  - *How to adapt custom Python types to SQLite values*
  - *How to convert SQLite values to custom Python types*
  - *How to use the connection context manager*
  - *How to create and use row factories*
- *Explanation* for in-depth background on transaction control.

12.6.2 Reference

Module functions

`sqlite3.`**`connect`**(*database*, *timeout=5.0*, *detect_types=0*, *isolation_level='DEFERRED'*, *check_same_thread=True*, *factory=sqlite3.Connection*, *cached_statements=128*, *uri=False*, ***, *autocommit=sqlite3.LEGACY_TRANSACTION_CONTROL*)

Open a connection to an SQLite database.

매개변수

- **`database`** (*path-like object*) – The path to the database file to be opened. You can pass `":memory:"` to create an SQLite database existing only in memory, and open a connection to it.
- **`timeout`** (`float`) – How many seconds the connection should wait before raising an *`OperationalError`* when a table is locked. If another connection opens a transaction to modify a table, that table will be locked until the transaction is committed. Default five seconds.

- **detect_types** (int) – Control whether and how data types not *natively supported by SQLite* are looked up to be converted to Python types, using the converters registered with `register_converter()`. Set it to any combination (using `|`, bitwise or) of `PARSE_DECLTYPES` and `PARSE_COLNAMES` to enable this. Column names takes precedence over declared types if both flags are set. By default (`0`), type detection is disabled.
- **isolation_level** (str | None) – Control legacy transaction handling behaviour. See `Connection.isolation_level` and *Transaction control via the isolation_level attribute* for more information. Can be `"DEFERRED"` (default), `"EXCLUSIVE"` or `"IMMEDIATE"`; or `None` to disable opening transactions implicitly. Has no effect unless `Connection.autocommit` is set to `LEGACY_TRANSACTION_CONTROL` (the default).
- **check_same_thread** (bool) – If `True` (default), `ProgrammingError` will be raised if the database connection is used by a thread other than the one that created it. If `False`, the connection may be accessed in multiple threads; write operations may need to be serialized by the user to avoid data corruption. See `threadsafety` for more information.
- **factory** (Connection) – A custom subclass of `Connection` to create the connection with, if not the default `Connection` class.
- **cached_statements** (int) – The number of statements that `sqlite3` should internally cache for this connection, to avoid parsing overhead. By default, 128 statements.
- **uri** (bool) – If set to `True`, *database* is interpreted as a URI (Uniform Resource Identifier) with a file path and an optional query string. The scheme part *must* be `"file:"`, and the path can be relative or absolute. The query string allows passing parameters to SQLite, enabling various *How to work with SQLite URIs*.
- **autocommit** (bool) – Control **PEP 249** transaction handling behaviour. See `Connection.autocommit` and *Transaction control via the autocommit attribute* for more information. *autocommit* currently defaults to `LEGACY_TRANSACTION_CONTROL`. The default will change to `False` in a future Python release.

반환 형식
Connection

인자 `database`로 감사 이벤트(*auditing event*) `sqlite3.connect`를 발생시킵니다.

Raises an *auditing event* `sqlite3.connect/handle` with argument `connection_handle`.

버전 3.4에서 변경: *uri* 매개 변수가 추가되었습니다.

버전 3.7에서 변경: *database*는 이제 문자열뿐만 아니라 경로류 객체 일 수도 있습니다.

버전 3.10에서 변경: Added the `sqlite3.connect/handle` auditing event.

버전 3.12에서 변경: Added the *autocommit* parameter.

버전 3.13에서 변경: Positional use of the parameters *timeout*, *detect_types*, *isolation_level*, *check_same_thread*, *factory*, *cached_statements*, and *uri* is deprecated. They will become keyword-only parameters in Python 3.15.

`sqlite3.`**complete_statement**(*statement*)

Return `True` if the string *statement* appears to contain one or more complete SQL statements. No syntactic verification or parsing of any kind is performed, other than checking that there are no unclosed string literals and the statement is terminated by a semicolon.

For example:

```
>>> sqlite3.complete_statement("SELECT foo FROM bar;")
True
>>> sqlite3.complete_statement("SELECT foo")
False
```

This function may be useful during command-line input to determine if the entered text seems to form a complete SQL statement, or if additional input is needed before calling `execute()`.

See `runsource()` in Lib/sqlite3/__main__.py for real-world use.

`sqlite3.`**`enable_callback_tracebacks`**(*flag*, /)

Enable or disable callback tracebacks. By default you will not get any tracebacks in user-defined functions, aggregates, converters, authorizer callbacks etc. If you want to debug them, you can call this function with *flag* set to `True`. Afterwards, you will get tracebacks from callbacks on `sys.stderr`. Use `False` to disable the feature again.

> **참고**
>
> Errors in user-defined function callbacks are logged as unraisable exceptions. Use an `unraisable hook handler` for introspection of the failed callback.

`sqlite3.`**`register_adapter`**(*type*, *adapter*, /)

Register an *adapter callable* to adapt the Python type *type* into an SQLite type. The adapter is called with a Python object of type *type* as its sole argument, and must return a value of a *type that SQLite natively understands*.

`sqlite3.`**`register_converter`**(*typename*, *converter*, /)

Register the *converter callable* to convert SQLite objects of type *typename* into a Python object of a specific type. The converter is invoked for all SQLite values of type *typename*; it is passed a `bytes` object and should return an object of the desired Python type. Consult the parameter *detect_types* of `connect()` for information regarding how type detection works.

Note: *typename* and the name of the type in your query are matched case-insensitively.

Module constants

`sqlite3.`**`LEGACY_TRANSACTION_CONTROL`**

Set `autocommit` to this constant to select old style (pre-Python 3.12) transaction control behaviour. See *Transaction control via the isolation_level attribute* for more information.

`sqlite3.`**`PARSE_DECLTYPES`**

Pass this flag value to the *detect_types* parameter of `connect()` to look up a converter function using the declared types for each column. The types are declared when the database table is created. `sqlite3` will look up a converter function using the first word of the declared type as the converter dictionary key. For example:

```
CREATE TABLE test(
   i integer primary key,  ! will look up a converter named "integer"
   p point,                ! will look up a converter named "point"
   n number(10)            ! will look up a converter named "number"
 )
```

This flag may be combined with `PARSE_COLNAMES` using the `|` (bitwise or) operator.

> **참고**
>
> Generated fields (for example `MAX(p)`) are returned as `str`. Use `PARSE_COLNAMES` to enforce types for such queries.

`sqlite3.`**`PARSE_COLNAMES`**

Pass this flag value to the *detect_types* parameter of `connect()` to look up a converter function by using the type name, parsed from the query column name, as the converter dictionary key. The query column name must be wrapped in double quotes (`"`) and the type name must be wrapped in square brackets (`[]`).

```
SELECT MAX(p) as "p [point]" FROM test;  ! will look up converter "point"
```

This flag may be combined with *PARSE_DECLTYPES* using the | (bitwise or) operator.

sqlite3.**SQLITE_OK**

sqlite3.**SQLITE_DENY**

sqlite3.**SQLITE_IGNORE**

Flags that should be returned by the *authorizer_callback* *callable* passed to *Connection.set_authorizer()*, to indicate whether:

- Access is allowed (SQLITE_OK),
- The SQL statement should be aborted with an error (SQLITE_DENY)
- The column should be treated as a NULL value (SQLITE_IGNORE)

sqlite3.**apilevel**

String constant stating the supported DB-API level. Required by the DB-API. Hard-coded to "2.0".

sqlite3.**paramstyle**

String constant stating the type of parameter marker formatting expected by the sqlite3 module. Required by the DB-API. Hard-coded to "qmark".

> **참고**
>
> The named DB-API parameter style is also supported.

sqlite3.**sqlite_version**

Version number of the runtime SQLite library as a *string*.

sqlite3.**sqlite_version_info**

Version number of the runtime SQLite library as a *tuple* of *integers*.

sqlite3.**threadsafety**

Integer constant required by the DB-API 2.0, stating the level of thread safety the sqlite3 module supports. This attribute is set based on the default threading mode the underlying SQLite library is compiled with. The SQLite threading modes are:

1. **Single-thread**: In this mode, all mutexes are disabled and SQLite is unsafe to use in more than a single thread at once.
2. **Multi-thread**: In this mode, SQLite can be safely used by multiple threads provided that no single database connection is used simultaneously in two or more threads.
3. **Serialized**: In serialized mode, SQLite can be safely used by multiple threads with no restriction.

The mappings from SQLite threading modes to DB-API 2.0 threadsafety levels are as follows:

SQLite threading mode	**thread-safety**	SQLITE_THREADS/	DB-API 2.0 meaning
single-thread	0	0	Threads may not share the module
multi-thread	1	2	Threads may share the module, but not connections
serialized	3	1	Threads may share the module, connections and cursors

버전 3.11에서 변경: Set *threadsafety* dynamically instead of hard-coding it to 1.

sqlite3.**SQLITE_DBCONFIG_DEFENSIVE**

sqlite3.**SQLITE_DBCONFIG_DQS_DDL**

sqlite3.**SQLITE_DBCONFIG_DQS_DML**

sqlite3.**SQLITE_DBCONFIG_ENABLE_FKEY**

sqlite3.**SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER**
sqlite3.**SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION**
sqlite3.**SQLITE_DBCONFIG_ENABLE_QPSG**
sqlite3.**SQLITE_DBCONFIG_ENABLE_TRIGGER**
sqlite3.**SQLITE_DBCONFIG_ENABLE_VIEW**
sqlite3.**SQLITE_DBCONFIG_LEGACY_ALTER_TABLE**
sqlite3.**SQLITE_DBCONFIG_LEGACY_FILE_FORMAT**
sqlite3.**SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE**
sqlite3.**SQLITE_DBCONFIG_RESET_DATABASE**
sqlite3.**SQLITE_DBCONFIG_TRIGGER_EQP**
sqlite3.**SQLITE_DBCONFIG_TRUSTED_SCHEMA**
sqlite3.**SQLITE_DBCONFIG_WRITABLE_SCHEMA**

These constants are used for the `Connection.setconfig()` and `getconfig()` methods.

The availability of these constants varies depending on the version of SQLite Python was compiled with.

Added in version 3.12.

> **더 보기**
>
> **https://www.sqlite.org/c3ref/c_dbconfig_defensive.html**
> SQLite docs: Database Connection Configuration Options

Deprecated since version 3.12, removed in version 3.14: The `version` and `version_info` constants.

Connection objects

class sqlite3.**Connection**

Each open SQLite database is represented by a `Connection` object, which is created using `sqlite3.connect()`. Their main purpose is creating `Cursor` objects, and *Transaction control*.

> **더 보기**
>
> - *How to use connection shortcut methods*
> - *How to use the connection context manager*

버전 3.13에서 변경: A `ResourceWarning` is emitted if `close()` is not called before a `Connection` object is deleted.

An SQLite database connection has the following attributes and methods:

cursor(*factory=Cursor*)

Create and return a `Cursor` object. The cursor method accepts a single optional parameter *factory*. If supplied, this must be a *callable* returning an instance of `Cursor` or its subclasses.

blobopen(*table*, *column*, *row*, */*, ***, *readonly=False*, *name='main'*)

Open a `Blob` handle to an existing BLOB (Binary Large OBject).

매개변수

- **table** (`str`) – The name of the table where the blob is located.
- **column** (`str`) – The name of the column where the blob is located.
- **row** (`str`) – The name of the row where the blob is located.

- **readonly** (bool) – Set to True if the blob should be opened without write permissions. Defaults to False.
- **name** (str) – The name of the database where the blob is located. Defaults to "main".

예외 발생

OperationalError – When trying to open a blob in a WITHOUT ROWID table.

반환 형식

Blob

> **참고**
>
> The blob size cannot be changed using the *Blob* class. Use the SQL function zeroblob to create a blob with a fixed size.

Added in version 3.11.

commit()

Commit any pending transaction to the database. If *autocommit* is True, or there is no open transaction, this method does nothing. If autocommit is False, a new transaction is implicitly opened if a pending transaction was committed by this method.

rollback()

Roll back to the start of any pending transaction. If *autocommit* is True, or there is no open transaction, this method does nothing. If autocommit is False, a new transaction is implicitly opened if a pending transaction was rolled back by this method.

close()

Close the database connection. If *autocommit* is False, any pending transaction is implicitly rolled back. If autocommit is True or *LEGACY_TRANSACTION_CONTROL*, no implicit transaction control is executed. Make sure to *commit()* before closing to avoid losing pending changes.

execute(*sql*, *parameters*=(), /)

Create a new *Cursor* object and call *execute()* on it with the given *sql* and *parameters*. Return the new cursor object.

executemany(*sql*, *parameters*, /)

Create a new *Cursor* object and call *executemany()* on it with the given *sql* and *parameters*. Return the new cursor object.

executescript(*sql_script*, /)

Create a new *Cursor* object and call *executescript()* on it with the given *sql_script*. Return the new cursor object.

create_function(*name*, *narg*, *func*, *, *deterministic=False*)

Create or remove a user-defined SQL function.

매개변수

- **name** (str) – The name of the SQL function.
- **narg** (int) – The number of arguments the SQL function can accept. If -1, it may take any number of arguments.
- **func** (*callback* | None) – A *callable* that is called when the SQL function is invoked. The callable must return *a type natively supported by SQLite*. Set to None to remove an existing SQL function.
- **deterministic** (bool) – If True, the created SQL function is marked as deterministic, which allows SQLite to perform additional optimizations.

버전 3.8에서 변경: Added the *deterministic* parameter.

예:

```
>>> import hashlib
>>> def md5sum(t):
...     return hashlib.md5(t).hexdigest()
>>> con = sqlite3.connect(":memory:")
>>> con.create_function("md5", 1, md5sum)
>>> for row in con.execute("SELECT md5(?)", (b"foo",)):
...     print(row)
('acbd18db4cc2f85cedef654fccc4a4d8',)
>>> con.close()
```

버전 3.13에서 변경: Passing *name*, *narg*, and *func* as keyword arguments is deprecated. These parameters will become positional-only in Python 3.15.

create_aggregate(*name*, *n_arg*, *aggregate_class*)

Create or remove a user-defined SQL aggregate function.

매개변수

- **name** (str) – The name of the SQL aggregate function.
- **n_arg** (int) – The number of arguments the SQL aggregate function can accept. If -1, it may take any number of arguments.
- **aggregate_class** (*class* | None) – A class must implement the following methods:
  - step(): Add a row to the aggregate.
  - finalize(): Return the final result of the aggregate as *a type natively supported by SQLite*.

  The number of arguments that the step() method must accept is controlled by *n_arg*.

  Set to None to remove an existing SQL aggregate function.

예:

```
class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.execute("CREATE TABLE test(i)")
cur.execute("INSERT INTO test(i) VALUES(1)")
cur.execute("INSERT INTO test(i) VALUES(2)")
cur.execute("SELECT mysum(i) FROM test")
print(cur.fetchone()[0])

con.close()
```

버전 3.13에서 변경: Passing *name*, *n_arg*, and *aggregate_class* as keyword arguments is deprecated. These parameters will become positional-only in Python 3.15.

create_window_function(*name*, *num_params*, *aggregate_class*, /)

Create or remove a user-defined aggregate window function.

매개변수

- **name** (str) – The name of the SQL aggregate window function to create or remove.
- **num_params** (int) – The number of arguments the SQL aggregate window function can accept. If -1, it may take any number of arguments.
- **aggregate_class** (*class* | None) – A class that must implement the following methods:
  - step(): Add a row to the current window.
  - value(): Return the current value of the aggregate.
  - inverse(): Remove a row from the current window.
  - finalize(): Return the final result of the aggregate as *a type natively supported by SQLite*.

  The number of arguments that the step() and value() methods must accept is controlled by *num_params*.

  Set to None to remove an existing SQL aggregate window function.

예외 발생

NotSupportedError – If used with a version of SQLite older than 3.25.0, which does not support aggregate window functions.

Added in version 3.11.

예:

```
# Example taken from https://www.sqlite.org/windowfunctions.html#udfwinfunc
class WindowSumInt:
    def __init__(self):
        self.count = 0

    def step(self, value):
        """Add a row to the current window."""
        self.count += value

    def value(self):
        """Return the current value of the aggregate."""
        return self.count

    def inverse(self, value):
        """Remove a row from the current window."""
        self.count -= value

    def finalize(self):
        """Return the final value of the aggregate.

        Any clean-up actions should be placed here.
        """
        return self.count


con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE test(x, y)")
values = [
    ("a", 4),
    ("b", 5),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    ("c", 3),
    ("d", 8),
    ("e", 1),
]
cur.executemany("INSERT INTO test VALUES(?, ?)", values)
con.create_window_function("sumint", 1, WindowSumInt)
cur.execute("""
    SELECT x, sumint(y) OVER (
        ORDER BY x ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) AS sum_y
    FROM test ORDER BY x
""")
print(cur.fetchall())
con.close()
```

create_collation(*name*, *callable*, /)

Create a collation named *name* using the collating function *callable*. *callable* is passed two `string` arguments, and it should return an `integer`:

- `1` if the first is ordered higher than the second
- `-1` if the first is ordered lower than the second
- `0` if they are ordered equal

The following example shows a reverse sorting collation:

```
def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
    else:
        return -1

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.execute("CREATE TABLE test(x)")
cur.executemany("INSERT INTO test(x) VALUES(?)", [("a",), ("b",)])
cur.execute("SELECT x FROM test ORDER BY x COLLATE reverse")
for row in cur:
    print(row)
con.close()
```

Remove a collation function by setting *callable* to `None`.

버전 3.11에서 변경: The collation name can contain any Unicode character. Earlier, only ASCII characters were allowed.

interrupt()

Call this method from a different thread to abort any queries that might be executing on the connection. Aborted queries will raise an `OperationalError`.

set_authorizer(*authorizer_callback*)

Register *callable* *authorizer_callback* to be invoked for each attempt to access a column of a table in the database. The callback should return one of `SQLITE_OK`, `SQLITE_DENY`, or `SQLITE_IGNORE` to signal how access to the column should be handled by the underlying SQLite library.

The first argument to the callback signifies what kind of operation is to be authorized. The second and third argument will be arguments or `None` depending on the first argument. The 4th argument is the name

of the database ("main", "temp", etc.) if applicable. The 5th argument is the name of the inner-most trigger or view that is responsible for the access attempt or `None` if this access attempt is directly from input SQL code.

Please consult the SQLite documentation about the possible values for the first argument and the meaning of the second and third argument depending on the first one. All necessary constants are available in the `sqlite3` module.

Passing `None` as *authorizer_callback* will disable the authorizer.

버전 3.11에서 변경: Added support for disabling the authorizer using `None`.

버전 3.13에서 변경: Passing *authorizer_callback* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

set_progress_handler(*progress_handler*, *n*)

Register *callable* *progress_handler* to be invoked for every *n* instructions of the SQLite virtual machine. This is useful if you want to get called from SQLite during long-running operations, for example to update a GUI.

If you want to clear any previously installed progress handler, call the method with `None` for *progress_handler*.

Returning a non-zero value from the handler function will terminate the currently executing query and cause it to raise a `DatabaseError` exception.

버전 3.13에서 변경: Passing *progress_handler* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

set_trace_callback(*trace_callback*)

Register *callable* *trace_callback* to be invoked for each SQL statement that is actually executed by the SQLite backend.

The only argument passed to the callback is the statement (as `str`) that is being executed. The return value of the callback is ignored. Note that the backend does not only run statements passed to the `Cursor.execute()` methods. Other sources include the *transaction management* of the `sqlite3` module and the execution of triggers defined in the current database.

Passing `None` as *trace_callback* will disable the trace callback.

> **참고**
>
> Exceptions raised in the trace callback are not propagated. As a development and debugging aid, use `enable_callback_tracebacks()` to enable printing tracebacks from exceptions raised in the trace callback.

Added in version 3.3.

버전 3.13에서 변경: Passing *trace_callback* as a keyword argument is deprecated. The parameter will become positional-only in Python 3.15.

enable_load_extension(*enabled*, /)

Enable the SQLite engine to load SQLite extensions from shared libraries if *enabled* is `True`; else, disallow loading SQLite extensions. SQLite extensions can define new functions, aggregates or whole new virtual table implementations. One well-known extension is the fulltext-search extension distributed with SQLite.

> **참고**
>
> The `sqlite3` module is not built with loadable extension support by default, because some platforms (notably macOS) have SQLite libraries which are compiled without this feature. To get load-

able extension support, you must pass the `--enable-loadable-sqlite-extensions` option to **configure**.

Raises an *auditing event* `sqlite3.enable_load_extension` with arguments `connection`, `enabled`.

Added in version 3.2.

버전 3.10에서 변경: Added the `sqlite3.enable_load_extension` auditing event.

```
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("CREATE VIRTUAL TABLE recipe USING fts3(name, ingredients)")
con.executescript("""
    INSERT INTO recipe (name, ingredients) VALUES('broccoli stew',
↪'broccoli peppers cheese tomatoes');
    INSERT INTO recipe (name, ingredients) VALUES('pumpkin stew', 'pumpkin␣
↪onions garlic celery');
    INSERT INTO recipe (name, ingredients) VALUES('broccoli pie',
↪'broccoli cheese onions flour');
    INSERT INTO recipe (name, ingredients) VALUES('pumpkin pie', 'pumpkin␣
↪sugar flour butter');
    """)
for row in con.execute("SELECT rowid, name, ingredients FROM recipe WHERE␣
↪name MATCH 'pie'"):
    print(row)
```

load_extension(*path*, /, *, *entrypoint=None*)

Load an SQLite extension from a shared library. Enable extension loading with `enable_load_extension()` before calling this method.

매개변수

- **path** (`str`) – The path to the SQLite extension.
- **entrypoint** (`str` | *None*) – Entry point name. If `None` (the default), SQLite will come up with an entry point name of its own; see the SQLite docs Loading an Extension for details.

Raises an *auditing event* `sqlite3.load_extension` with arguments `connection`, `path`.

Added in version 3.2.

버전 3.10에서 변경: Added the `sqlite3.load_extension` auditing event.

버전 3.12에서 변경: Added the *entrypoint* parameter.

iterdump(*, *filter=None*)

Return an *iterator* to dump the database as SQL source code. Useful when saving an in-memory database for later restoration. Similar to the `.dump` command in the **sqlite3** shell.

매개변수

filter (str | None) – An optional LIKE pattern for database objects to dump, e.g. prefix_%. If None (the default), all database objects will be included.

예:

```
# Convert file example.db to SQL dump file dump.sql
con = sqlite3.connect('example.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
con.close()
```

더 보기

How to handle non-UTF-8 text encodings

버전 3.13에서 변경: Added the *filter* parameter.

backup(*target*, *, *pages=-1*, *progress=None*, *name='main'*, *sleep=0.250*)

Create a backup of an SQLite database.

Works even if the database is being accessed by other clients or concurrently by the same connection.

매개변수

- **target** (Connection) – The database connection to save the backup to.
- **pages** (int) – The number of pages to copy at a time. If equal to or less than 0, the entire database is copied in a single step. Defaults to -1.
- **progress** (*callback* | None) – If set to a *callable*, it is invoked with three integer arguments for every backup iteration: the *status* of the last iteration, the *remaining* number of pages still to be copied, and the *total* number of pages. Defaults to None.
- **name** (str) – The name of the database to back up. Either "main" (the default) for the main database, "temp" for the temporary database, or the name of a custom database as attached using the ATTACH DATABASE SQL statement.
- **sleep** (float) – The number of seconds to sleep between successive attempts to back up remaining pages.

Example 1, copy an existing database into another:

```
def progress(status, remaining, total):
    print(f'Copied {total-remaining} of {total} pages...')

src = sqlite3.connect('example.db')
dst = sqlite3.connect('backup.db')
with dst:
    src.backup(dst, pages=1, progress=progress)
dst.close()
src.close()
```

Example 2, copy an existing database into a transient copy:

```
src = sqlite3.connect('example.db')
dst = sqlite3.connect(':memory:')
src.backup(dst)
dst.close()
src.close()
```

Added in version 3.7.

> **더 보기**
>
> *How to handle non-UTF-8 text encodings*

getlimit(*category*, /)

Get a connection runtime limit.

매개변수

category (int) – The SQLite limit category to be queried.

반환 형식

int

예외 발생

ProgrammingError – If *category* is not recognised by the underlying SQLite library.

Example, query the maximum length of an SQL statement for `Connection` `con` (the default is 1000000000):

```
>>> con.getlimit(sqlite3.SQLITE_LIMIT_SQL_LENGTH)
1000000000
```

Added in version 3.11.

setlimit(*category*, *limit*, /)

Set a connection runtime limit. Attempts to increase a limit above its hard upper bound are silently truncated to the hard upper bound. Regardless of whether or not the limit was changed, the prior value of the limit is returned.

매개변수

- **category** (int) – The SQLite limit category to be set.
- **limit** (int) – The value of the new limit. If negative, the current limit is unchanged.

반환 형식

int

예외 발생

ProgrammingError – If *category* is not recognised by the underlying SQLite library.

Example, limit the number of attached databases to 1 for `Connection` `con` (the default limit is 10):

```
>>> con.setlimit(sqlite3.SQLITE_LIMIT_ATTACHED, 1)
10
>>> con.getlimit(sqlite3.SQLITE_LIMIT_ATTACHED)
1
```

Added in version 3.11.

getconfig(*op*, /)

Query a boolean connection configuration option.

매개변수

op (int) – A *SQLITE_DBCONFIG code*.

반환 형식

bool

Added in version 3.12.

setconfig(*op*, *enable=True*, /)

Set a boolean connection configuration option.

매개변수

- **op** (`int`) – A *SQLITE_DBCONFIG code*.
- **enable** (`bool`) – `True` if the configuration option should be enabled (default); `False` if it should be disabled.

Added in version 3.12.

serialize(*, *name='main'*)

Serialize a database into a `bytes` object. For an ordinary on-disk database file, the serialization is just a copy of the disk file. For an in-memory database or a "temp" database, the serialization is the same sequence of bytes which would be written to disk if that database were backed up to disk.

매개변수

name (`str`) – The database name to be serialized. Defaults to `"main"`.

반환 형식

bytes

> **참고**
>
> This method is only available if the underlying SQLite library has the serialize API.

Added in version 3.11.

deserialize(*data*, /, *, *name='main'*)

Deserialize a `serialized` database into a `Connection`. This method causes the database connection to disconnect from database *name*, and reopen *name* as an in-memory database based on the serialization contained in *data*.

매개변수

- **data** (`bytes`) – A serialized database.
- **name** (`str`) – The database name to deserialize into. Defaults to `"main"`.

예외 발생

- ***OperationalError*** – If the database connection is currently involved in a read transaction or a backup operation.
- ***DatabaseError*** – If *data* does not contain a valid SQLite database.
- ***OverflowError*** – If `len(data)` is larger than `2**63 - 1`.

> **참고**
>
> This method is only available if the underlying SQLite library has the deserialize API.

Added in version 3.11.

autocommit

This attribute controls **PEP 249**-compliant transaction behaviour. `autocommit` has three allowed values:

- `False`: Select **PEP 249**-compliant transaction behaviour, implying that `sqlite3` ensures a transaction is always open. Use `commit()` and `rollback()` to close transactions.

  This is the recommended value of `autocommit`.
- `True`: Use SQLite' s autocommit mode. `commit()` and `rollback()` have no effect in this mode.

- *LEGACY_TRANSACTION_CONTROL*: Pre-Python 3.12 (non-**PEP 249**-compliant) transaction control. See *isolation_level* for more details.

  This is currently the default value of `autocommit`.

Changing `autocommit` to `False` will open a new transaction, and changing it to `True` will commit any pending transaction.

See *Transaction control via the autocommit attribute* for more details.

> **참고**
>
> The *isolation_level* attribute has no effect unless *autocommit* is *LEGACY_TRANSACTION_CONTROL*.

Added in version 3.12.

in_transaction

This read-only attribute corresponds to the low-level SQLite autocommit mode.

`True` if a transaction is active (there are uncommitted changes), `False` otherwise.

Added in version 3.2.

isolation_level

Controls the *legacy transaction handling mode* of `sqlite3`. If set to `None`, transactions are never implicitly opened. If set to one of `"DEFERRED"`, `"IMMEDIATE"`, or `"EXCLUSIVE"`, corresponding to the underlying SQLite transaction behaviour, *implicit transaction management* is performed.

If not overridden by the *isolation_level* parameter of *connect()*, the default is `""`, which is an alias for `"DEFERRED"`.

> **참고**
>
> Using *autocommit* to control transaction handling is recommended over using `isolation_level`. `isolation_level` has no effect unless *autocommit* is set to *LEGACY_TRANSACTION_CONTROL* (the default).

row_factory

The initial *row_factory* for *Cursor* objects created from this connection. Assigning to this attribute does not affect the `row_factory` of existing cursors belonging to this connection, only new ones. Is `None` by default, meaning each row is returned as a *tuple*.

See *How to create and use row factories* for more details.

text_factory

A *callable* that accepts a *bytes* parameter and returns a text representation of it. The callable is invoked for SQLite values with the `TEXT` data type. By default, this attribute is set to *str*.

See *How to handle non-UTF-8 text encodings* for more details.

total_changes

Return the total number of database rows that have been modified, inserted, or deleted since the database connection was opened.

Cursor objects

A `Cursor` object represents a database cursor which is used to execute SQL statements, and manage the context of a fetch operation. Cursors are created using *Connection.cursor()*, or by using any of the *connection shortcut methods*.

Cursor objects are *iterators*, meaning that if you `execute()` a `SELECT` query, you can simply iterate over the cursor to fetch the resulting rows:

```
for row in cur.execute("SELECT t FROM data"):
    print(row)
```

class sqlite3.**Cursor**

`Cursor` 인스턴스에는 다음과 같은 어트리뷰트와 메서드가 있습니다.

execute(*sql*, *parameters*=(), /)

Execute a single SQL statement, optionally binding Python values using *placeholders*.

매개변수

- **sql** (`str`) – A single SQL statement.
- **parameters** (`dict` | *sequence*) – Python values to bind to placeholders in *sql*. A `dict` if named placeholders are used. A sequence if unnamed placeholders are used. See *How to use placeholders to bind values in SQL queries*.

예외 발생

ProgrammingError – When *sql* contains more than one SQL statement. When *named placeholders* are used and *parameters* is a sequence instead of a `dict`.

If `autocommit` is `LEGACY_TRANSACTION_CONTROL`, `isolation_level` is not `None`, *sql* is an `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statement, and there is no open transaction, a transaction is implicitly opened before executing *sql*.

버전 3.14에서 변경: `ProgrammingError` is emitted if *named placeholders* are used and *parameters* is a sequence instead of a `dict`.

Use `executescript()` to execute multiple SQL statements.

executemany(*sql*, *parameters*, /)

For every item in *parameters*, repeatedly execute the *parameterized* DML (Data Manipulation Language) SQL statement *sql*.

Uses the same implicit transaction handling as `execute()`.

매개변수

- **sql** (`str`) – A single SQL DML statement.
- **parameters** (*iterable*) – An iterable of parameters to bind with the placeholders in *sql*. See *How to use placeholders to bind values in SQL queries*.

예외 발생

ProgrammingError – When *sql* contains more than one SQL statement or is not a DML statement, When *named placeholders* are used and the items in *parameters* are sequences instead of `dict`s.

예:

```
rows = [
    ("row1",),
    ("row2",),
]
# cur is an sqlite3.Cursor object
cur.executemany("INSERT INTO data VALUES(?)", rows)
```

> **참고**
>
> Any resulting rows are discarded, including DML statements with RETURNING clauses.

버전 3.14에서 변경: `ProgrammingError` is emitted if *named placeholders* are used and the items in *parameters* are sequences instead of `dict`s.

executescript(*sql_script*, /)

Execute the SQL statements in *sql_script*. If the `autocommit` is `LEGACY_TRANSACTION_CONTROL` and there is a pending transaction, an implicit `COMMIT` statement is executed first. No other implicit transaction control is performed; any transaction control must be added to *sql_script*.

sql_script must be a `string`.

예:

```
# cur is an sqlite3.Cursor object
cur.executescript("""
    BEGIN;
    CREATE TABLE person(firstname, lastname, age);
    CREATE TABLE book(title, author, published);
    CREATE TABLE publisher(name, address);
    COMMIT;
""")
```

fetchone()

If `row_factory` is `None`, return the next row query result set as a `tuple`. Else, pass it to the row factory and return its result. Return `None` if no more data is available.

fetchmany(*size=cursor.arraysize*)

Return the next set of rows of a query result as a `list`. Return an empty list if no more rows are available.

The number of rows to fetch per call is specified by the *size* parameter. If *size* is not given, `arraysize` determines the number of rows to be fetched. If fewer than *size* rows are available, as many rows as are available are returned.

size 매개 변수와 관련된 성능 고려 사항이 있습니다. 최적의 성능을 위해서, 일반적으로 arraysize 어트리뷰트를 사용하는 것이 가장 좋습니다. *size* 매개 변수가 사용되면, `fetchmany()` 호출마다 같은 값을 유지하는 것이 가장 좋습니다.

fetchall()

Return all (remaining) rows of a query result as a `list`. Return an empty list if no rows are available. Note that the `arraysize` attribute can affect the performance of this operation.

close()

(`__del__`이 호출 될 때가 아니라) 지금 커서를 닫습니다.

이 시점부터는 커서를 사용할 수 없습니다; 커서로 어떤 연산이건 시도하면 `ProgrammingError` 예외가 발생합니다.

setinputsizes(*sizes*, /)

Required by the DB-API. Does nothing in `sqlite3`.

setoutputsize(*size*, *column=None*, /)

Required by the DB-API. Does nothing in `sqlite3`.

arraysize

`fetchmany()`에 의해 반환되는 행의 수를 제어하는 읽기/쓰기 어트리뷰트. 기본값은 1입니다. 이는 호출 당 하나의 행을 가져오는 것을 뜻합니다.

connection

Read-only attribute that provides the SQLite database `Connection` belonging to the cursor. A `Cursor` object created by calling `con.cursor()` will have a `connection` attribute that refers to *con*:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
>>> con.close()
```

description

Read-only attribute that provides the column names of the last query. To remain compatible with the Python DB API, it returns a 7-tuple for each column where the last six items of each tuple are `None`.

일치하는 행이 없는 `SELECT` 문에도 설정됩니다.

lastrowid

Read-only attribute that provides the row id of the last inserted row. It is only updated after successful `INSERT` or `REPLACE` statements using the *execute()* method. For other statements, after *executemany()* or *executescript()*, or if the insertion failed, the value of `lastrowid` is left unchanged. The initial value of `lastrowid` is `None`.

> **참고**
>
> Inserts into `WITHOUT ROWID` tables are not recorded.

버전 3.6에서 변경: `REPLACE` 문에 대한 지원이 추가되었습니다.

rowcount

Read-only attribute that provides the number of modified rows for `INSERT`, `UPDATE`, `DELETE`, and `REPLACE` statements; is `-1` for other statements, including CTE (Common Table Expression) queries. It is only updated by the *execute()* and *executemany()* methods, after the statement has run to completion. This means that any resulting rows must be fetched in order for `rowcount` to be updated.

row_factory

Control how a row fetched from this `Cursor` is represented. If `None`, a row is represented as a *tuple*. Can be set to the included *sqlite3.Row*; or a *callable* that accepts two arguments, a *Cursor* object and the `tuple` of row values, and returns a custom object representing an SQLite row.

Defaults to what *Connection.row_factory* was set to when the `Cursor` was created. Assigning to this attribute does not affect *Connection.row_factory* of the parent connection.

See *How to create and use row factories* for more details.

Row objects

class sqlite3.**Row**

A `Row` instance serves as a highly optimized *row_factory* for *Connection* objects. It supports iteration, equality testing, *len()*, and *mapping* access by column name and index.

Two `Row` objects compare equal if they have identical column names and values.

See *How to create and use row factories* for more details.

keys()

Return a *list* of column names as *strings*. Immediately after a query, it is the first member of each tuple in *Cursor.description*.

버전 3.5에서 변경: 슬라이싱 지원이 추가되었습니다.

Blob objects

class sqlite3.**Blob**

Added in version 3.11.

A `Blob` instance is a *file-like object* that can read and write data in an SQLite BLOB. Call `len(blob)` to get the size (number of bytes) of the blob. Use indices and *slices* for direct access to the blob data.

Use the `Blob` as a *context manager* to ensure that the blob handle is closed after use.

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE test(blob_col blob)")
con.execute("INSERT INTO test(blob_col) VALUES(zeroblob(13))")

# Write to our blob, using two write operations:
with con.blobopen("test", "blob_col", 1) as blob:
    blob.write(b"hello, ")
    blob.write(b"world.")
    # Modify the first and last bytes of our blob
    blob[0] = ord("H")
    blob[-1] = ord("!")

# Read the contents of our blob
with con.blobopen("test", "blob_col", 1) as blob:
    greeting = blob.read()

print(greeting)  # outputs "b'Hello, world!'"
con.close()
```

close()

Close the blob.

The blob will be unusable from this point onward. An `Error` (or subclass) exception will be raised if any further operation is attempted with the blob.

read(*length=-1*, /)

Read *length* bytes of data from the blob at the current offset position. If the end of the blob is reached, the data up to EOF (End of File) will be returned. When *length* is not specified, or is negative, `read()` will read until the end of the blob.

write(*data*, /)

Write *data* to the blob at the current offset. This function cannot change the blob length. Writing beyond the end of the blob will raise `ValueError`.

tell()

Return the current access position of the blob.

seek(*offset*, *origin=os.SEEK_SET*, /)

Set the current access position of the blob to *offset*. The *origin* argument defaults to `os.SEEK_SET` (absolute blob positioning). Other values for *origin* are `os.SEEK_CUR` (seek relative to the current position) and `os.SEEK_END` (seek relative to the blob' s end).

PrepareProtocol objects

class sqlite3.**PrepareProtocol**

The PrepareProtocol type' s single purpose is to act as a **PEP 246** style adaption protocol for objects that can *adapt themselves* to *native SQLite types*.

예외

The exception hierarchy is defined by the DB-API 2.0 (**PEP 249**).

exception sqlite3.**Warning**

This exception is not currently raised by the `sqlite3` module, but may be raised by applications using `sqlite3`, for example if a user-defined function truncates data while inserting. `Warning` is a subclass of *`Exception`*.

exception sqlite3.**Error**

The base class of the other exceptions in this module. Use this to catch all errors with one single `except` statement. `Error` is a subclass of *`Exception`*.

If the exception originated from within the SQLite library, the following two attributes are added to the exception:

sqlite_errorcode

The numeric error code from the SQLite API

Added in version 3.11.

sqlite_errorname

The symbolic name of the numeric error code from the SQLite API

Added in version 3.11.

exception sqlite3.**InterfaceError**

Exception raised for misuse of the low-level SQLite C API. In other words, if this exception is raised, it probably indicates a bug in the `sqlite3` module. `InterfaceError` is a subclass of *`Error`*.

exception sqlite3.**DatabaseError**

Exception raised for errors that are related to the database. This serves as the base exception for several types of database errors. It is only raised implicitly through the specialised subclasses. `DatabaseError` is a subclass of *`Error`*.

exception sqlite3.**DataError**

Exception raised for errors caused by problems with the processed data, like numeric values out of range, and strings which are too long. `DataError` is a subclass of *`DatabaseError`*.

exception sqlite3.**OperationalError**

Exception raised for errors that are related to the database' s operation, and not necessarily under the control of the programmer. For example, the database path is not found, or a transaction could not be processed. `OperationalError` is a subclass of *`DatabaseError`*.

exception sqlite3.**IntegrityError**

데이터베이스의 관계형 무결성이 영향을 받을 때 발생하는 예외. 예를 들어, 외부 키 (foreign key) 검사가 실패할 때. *`DatabaseError`*의 서브 클래스입니다.

exception sqlite3.**InternalError**

Exception raised when SQLite encounters an internal error. If this is raised, it may indicate that there is a problem with the runtime SQLite library. `InternalError` is a subclass of *`DatabaseError`*.

exception sqlite3.**ProgrammingError**

Exception raised for `sqlite3` API programming errors, for example supplying the wrong number of bindings to a query, or trying to operate on a closed *`Connection`*. `ProgrammingError` is a subclass of *`DatabaseError`*.

exception sqlite3.**NotSupportedError**

Exception raised in case a method or database API is not supported by the underlying SQLite library. For example, setting *deterministic* to `True` in *`create_function()`*, if the underlying SQLite library does not support deterministic functions. `NotSupportedError` is a subclass of *`DatabaseError`*.

SQLite 와 파이썬 형

SQLite는 기본적으로 다음 형을 지원합니다: NULL, INTEGER, REAL, TEXT, BLOB.

따라서 다음과 같은 파이썬 형을 아무 문제 없이 SQLite로 보낼 수 있습니다:

파이썬 형	SQLite 형
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

이것은 SQLite 형이 기본적으로 파이썬 형으로 변환되는 방법입니다:

SQLite 형	파이썬 형
NULL	None
INTEGER	*int*
REAL	*float*
TEXT	*text_factory*에 따라 다릅니다, 기본적으로 *str*.
BLOB	*bytes*

The type system of the sqlite3 module is extensible in two ways: you can store additional Python types in an SQLite database via *object adapters*, and you can let the sqlite3 module convert SQLite types to Python types via *converters*.

Default adapters and converters (deprecated)

> **참고**
>
> The default adapters and converters are deprecated as of Python 3.12. Instead, use the *Adapter and converter recipes* and tailor them to your needs.

The deprecated default adapters and converters consist of:

- An adapter for *datetime.date* objects to *strings* in ISO 8601 format.
- An adapter for *datetime.datetime* objects to strings in ISO 8601 format.
- A converter for *declared* "date" types to *datetime.date* objects.
- A converter for declared "timestamp" types to *datetime.datetime* objects. Fractional parts will be truncated to 6 digits (microsecond precision).

> **참고**
>
> The default "timestamp" converter ignores UTC offsets in the database and always returns a naive *datetime.datetime* object. To preserve UTC offsets in timestamps, either leave converters disabled, or register an offset-aware converter with *register_converter()*.

버전 3.12부터 폐지됨.

Command-line interface

The `sqlite3` module can be invoked as a script, using the interpreter' s `-m` switch, in order to provide a simple SQLite shell. The argument signature is as follows:

```
python -m sqlite3 [-h] [-v] [filename] [sql]
```

Type `.quit` or CTRL-D to exit the shell.

-h, --help

Print CLI help.

-v, --version

Print underlying SQLite library version.

Added in version 3.12.

12.6.3 How-to guides

How to use placeholders to bind values in SQL queries

SQL operations usually need to use values from Python variables. However, beware of using Python' s string operations to assemble queries, as they are vulnerable to SQL injection attacks. For example, an attacker can simply close the single quote and inject `OR TRUE` to select all rows:

```
>>> # Never do this -- insecure!
>>> symbol = input()
' OR TRUE; --
>>> sql = "SELECT * FROM stocks WHERE symbol = '%s'" % symbol
>>> print(sql)
SELECT * FROM stocks WHERE symbol = '' OR TRUE; --'
>>> cur.execute(sql)
```

Instead, use the DB-API' s parameter substitution. To insert a variable into a query string, use a placeholder in the string, and substitute the actual values into the query by providing them as a `tuple` of values to the second argument of the cursor' s `execute()` method.

An SQL statement may use one of two kinds of placeholders: question marks (qmark style) or named placeholders (named style). For the qmark style, *parameters* must be a *sequence* whose length must match the number of placeholders, or a `ProgrammingError` is raised. For the named style, *parameters* must be an instance of a `dict` (or a subclass), which must contain keys for all named parameters; any extra items are ignored. Here' s an example of both styles:

```
con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE lang(name, first_appeared)")

# This is the named style used with executemany():
data = (
    {"name": "C", "year": 1972},
    {"name": "Fortran", "year": 1957},
    {"name": "Python", "year": 1991},
    {"name": "Go", "year": 2009},
)
cur.executemany("INSERT INTO lang VALUES(:name, :year)", data)

# This is the qmark style used in a SELECT query:
params = (1972,)
cur.execute("SELECT * FROM lang WHERE first_appeared = ?", params)
print(cur.fetchall())
con.close()
```

> **참고**
>
> **PEP 249** numeric placeholders are *not* supported. If used, they will be interpreted as named placeholders.

How to adapt custom Python types to SQLite values

SQLite supports only a limited set of data types natively. To store custom Python types in SQLite databases, *adapt* them to one of the *Python types SQLite natively understands*.

There are two ways to adapt Python objects to SQLite types: letting your object adapt itself, or using an *adapter callable*. The latter will take precedence above the former. For a library that exports a custom type, it may make sense to enable that type to adapt itself. As an application developer, it may make more sense to take direct control by registering custom adapter functions.

How to write adaptable objects

Suppose we have a `Point` class that represents a pair of coordinates, `x` and `y`, in a Cartesian coordinate system. The coordinate pair will be stored as a text string in the database, using a semicolon to separate the coordinates. This can be implemented by adding a `__conform__(self, protocol)` method which returns the adapted value. The object passed to *protocol* will be of type `PrepareProtocol`.

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return f"{self.x};{self.y}"

con = sqlite3.connect(":memory:")
cur = con.cursor()

cur.execute("SELECT ?", (Point(4.0, -3.2),))
print(cur.fetchone()[0])
con.close()
```

How to register adapter callables

The other possibility is to create a function that converts the Python object to an SQLite-compatible type. This function can then be registered using `register_adapter()`.

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return f"{point.x};{point.y}"

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

cur.execute("SELECT ?", (Point(1.0, 2.5),))
print(cur.fetchone()[0])
con.close()
```

How to convert SQLite values to custom Python types

Writing an adapter lets you convert *from* custom Python types *to* SQLite values. To be able to convert *from* SQLite values *to* custom Python types, we use *converters*.

Let' s go back to the `Point` class. We stored the x and y coordinates separated via semicolons as strings in SQLite.

First, we' ll define a converter function that accepts the string as a parameter and constructs a `Point` object from it.

> **참고**
>
> Converter functions are **always** passed a *bytes* object, no matter the underlying SQLite data type.

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

We now need to tell `sqlite3` when it should convert a given SQLite value. This is done when connecting to a database, using the *detect_types* parameter of *connect()*. There are three options:

- Implicit: set *detect_types* to *PARSE_DECLTYPES*
- Explicit: set *detect_types* to *PARSE_COLNAMES*
- Both: set *detect_types* to `sqlite3.PARSE_DECLTYPES | sqlite3.PARSE_COLNAMES`. Column names take precedence over declared types.

The following example illustrates the implicit and explicit approaches:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

def adapt_point(point):
    return f"{point.x};{point.y}"

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# Register the adapter and converter
sqlite3.register_adapter(Point, adapt_point)
sqlite3.register_converter("point", convert_point)

# 1) Parse using declared types
p = Point(4.0, -3.2)
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.execute("CREATE TABLE test(p point)")

cur.execute("INSERT INTO test(p) VALUES(?)", (p,))
cur.execute("SELECT p FROM test")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

# 2) Parse using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
cur = con.execute("CREATE TABLE test(p)")

cur.execute("INSERT INTO test(p) VALUES(?)", (p,))
cur.execute('SELECT p AS "p [point]" FROM test')
print("with column names:", cur.fetchone()[0])
cur.close()
con.close()
```

Adapter and converter recipes

This section shows recipes for common adapters and converters.

```
import datetime
import sqlite3

def adapt_date_iso(val):
    """Adapt datetime.date to ISO 8601 date."""
    return val.isoformat()

def adapt_datetime_iso(val):
    """Adapt datetime.datetime to timezone-naive ISO 8601 date."""
    return val.replace(tzinfo=None).isoformat()

def adapt_datetime_epoch(val):
    """Adapt datetime.datetime to Unix timestamp."""
    return int(val.timestamp())

sqlite3.register_adapter(datetime.date, adapt_date_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_iso)
sqlite3.register_adapter(datetime.datetime, adapt_datetime_epoch)

def convert_date(val):
    """Convert ISO 8601 date to datetime.date object."""
    return datetime.date.fromisoformat(val.decode())

def convert_datetime(val):
    """Convert ISO 8601 datetime to datetime.datetime object."""
    return datetime.datetime.fromisoformat(val.decode())

def convert_timestamp(val):
    """Convert Unix epoch timestamp to datetime.datetime object."""
    return datetime.datetime.fromtimestamp(int(val))

sqlite3.register_converter("date", convert_date)
sqlite3.register_converter("datetime", convert_datetime)
sqlite3.register_converter("timestamp", convert_timestamp)
```

How to use connection shortcut methods

Using the `execute()`, `executemany()`, and `executescript()` methods of the `Connection` class, your code can be written more concisely because you don' t have to create the (often superfluous) `Cursor` objects explicitly. Instead, the `Cursor` objects are created implicitly and these shortcut methods return the cursor objects. This way, you can execute a `SELECT` statement and iterate over it directly using only a single call on the `Connection` object.

```
# Create and fill the table.
con = sqlite3.connect(":memory:")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
con.execute("CREATE TABLE lang(name, first_appeared)")
data = [
    ("C++", 1985),
    ("Objective-C", 1984),
]
con.executemany("INSERT INTO lang(name, first_appeared) VALUES(?, ?)", data)

# Print the table contents
for row in con.execute("SELECT name, first_appeared FROM lang"):
    print(row)

print("I just deleted", con.execute("DELETE FROM lang").rowcount, "rows")

# close() is not a shortcut method and it's not called automatically;
# the connection object should be closed manually
con.close()
```

How to use the connection context manager

A *Connection* object can be used as a context manager that automatically commits or rolls back open transactions when leaving the body of the context manager. If the body of the `with` statement finishes without exceptions, the transaction is committed. If this commit fails, or if the body of the `with` statement raises an uncaught exception, the transaction is rolled back. If *autocommit* is `False`, a new transaction is implicitly opened after committing or rolling back.

If there is no open transaction upon leaving the body of the `with` statement, or if *autocommit* is `True`, the context manager does nothing.

> **참고**
>
> The context manager neither implicitly opens a new transaction nor closes the connection. If you need a closing context manager, consider using *contextlib.closing()*.

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(id INTEGER PRIMARY KEY, name VARCHAR UNIQUE)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))

# con.rollback() is called after the with block finishes with an exception,
# the exception is still raised and must be caught
try:
    with con:
        con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))
except sqlite3.IntegrityError:
    print("couldn't add Python twice")

# Connection object used as context manager only commits or rollbacks transactions,
# so the connection object should be closed manually
con.close()
```

How to work with SQLite URIs

Some useful URI tricks include:

- Open a database in read-only mode:

```
>>> con = sqlite3.connect("file:tutorial.db?mode=ro", uri=True)
>>> con.execute("CREATE TABLE readonly(data)")
Traceback (most recent call last):
OperationalError: attempt to write a readonly database
>>> con.close()
```

- Do not implicitly create a new database file if it does not already exist; will raise *OperationalError* if unable to create a new file:

```
>>> con = sqlite3.connect("file:nosuchdb.db?mode=rw", uri=True)
Traceback (most recent call last):
OperationalError: unable to open database file
```

- Create a shared named in-memory database:

```
db = "file:mem1?mode=memory&cache=shared"
con1 = sqlite3.connect(db, uri=True)
con2 = sqlite3.connect(db, uri=True)
with con1:
    con1.execute("CREATE TABLE shared(data)")
    con1.execute("INSERT INTO shared VALUES(28)")
res = con2.execute("SELECT data FROM shared")
assert res.fetchone() == (28,)

con1.close()
con2.close()
```

More information about this feature, including a list of parameters, can be found in the SQLite URI documentation.

How to create and use row factories

By default, `sqlite3` represents each row as a *tuple*. If a `tuple` does not suit your needs, you can use the *sqlite3.Row* class or a custom *row_factory*.

While `row_factory` exists as an attribute both on the *Cursor* and the *Connection*, it is recommended to set *Connection.row_factory*, so all cursors created from the connection will use the same row factory.

`Row` provides indexed and case-insensitive named access to columns, with minimal memory overhead and performance impact over a `tuple`. To use `Row` as a row factory, assign it to the `row_factory` attribute:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = sqlite3.Row
```

Queries now return `Row` objects:

```
>>> res = con.execute("SELECT 'Earth' AS name, 6378 AS radius")
>>> row = res.fetchone()
>>> row.keys()
['name', 'radius']
>>> row[0]         # Access by index.
'Earth'
>>> row["name"]    # Access by name.
'Earth'
>>> row["RADIUS"]  # Column names are case-insensitive.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
6378
>>> con.close()
```

> **참고**
>
> The `FROM` clause can be omitted in the `SELECT` statement, as in the above example. In such cases, SQLite returns a single row with columns defined by expressions, e.g. literals, with the given aliases `expr AS alias`.

You can create a custom *row_factory* that returns each row as a *dict*, with column names mapped to values:

```
def dict_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    return {key: value for key, value in zip(fields, row)}
```

Using it, queries now return a `dict` instead of a `tuple`:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = dict_factory
>>> for row in con.execute("SELECT 1 AS a, 2 AS b"):
...     print(row)
{'a': 1, 'b': 2}
>>> con.close()
```

The following row factory returns a *named tuple*:

```
from collections import namedtuple

def namedtuple_factory(cursor, row):
    fields = [column[0] for column in cursor.description]
    cls = namedtuple("Row", fields)
    return cls._make(row)
```

`namedtuple_factory()` can be used as follows:

```
>>> con = sqlite3.connect(":memory:")
>>> con.row_factory = namedtuple_factory
>>> cur = con.execute("SELECT 1 AS a, 2 AS b")
>>> row = cur.fetchone()
>>> row
Row(a=1, b=2)
>>> row[0]  # Indexed access.
1
>>> row.b   # Attribute access.
2
>>> con.close()
```

With some adjustments, the above recipe can be adapted to use a *dataclass*, or any other custom class, instead of a *namedtuple*.

How to handle non-UTF-8 text encodings

By default, `sqlite3` uses *str* to adapt SQLite values with the `TEXT` data type. This works well for UTF-8 encoded text, but it might fail for other encodings and invalid UTF-8. You can use a custom *text_factory* to handle such cases.

Because of SQLite' s flexible typing, it is not uncommon to encounter table columns with the `TEXT` data type containing non-UTF-8 encodings, or even arbitrary data. To demonstrate, let' s assume we have a database with

ISO-8859-2 (Latin-2) encoded text, for example a table of Czech-English dictionary entries. Assuming we now have a *Connection* instance `con` connected to this database, we can decode the Latin-2 encoded text using this *text_factory*:

```
con.text_factory = lambda data: str(data, encoding="latin2")
```

For invalid UTF-8 or arbitrary data in stored in `TEXT` table columns, you can use the following technique, borrowed from the unicode-howto:

```
con.text_factory = lambda data: str(data, errors="surrogateescape")
```

> **참고**
>
> The `sqlite3` module API does not support strings containing surrogates.

> **더 보기**
>
> unicode-howto

12.6.4 Explanation

Transaction control

`sqlite3` offers multiple methods of controlling whether, when and how database transactions are opened and closed. *Transaction control via the autocommit attribute* is recommended, while *Transaction control via the isolation_level attribute* retains the pre-Python 3.12 behaviour.

Transaction control via the `autocommit` attribute

The recommended way of controlling transaction behaviour is through the *Connection.autocommit* attribute, which should preferably be set using the *autocommit* parameter of *connect()*.

It is suggested to set *autocommit* to `False`, which implies **PEP 249**-compliant transaction control. This means:

- `sqlite3` ensures that a transaction is always open, so *connect()*, *Connection.commit()*, and *Connection.rollback()* will implicitly open a new transaction (immediately after closing the pending one, for the latter two). `sqlite3` uses `BEGIN DEFERRED` statements when opening transactions.
- Transactions should be committed explicitly using `commit()`.
- Transactions should be rolled back explicitly using `rollback()`.
- An implicit rollback is performed if the database is *close()*-ed with pending changes.

Set *autocommit* to `True` to enable SQLite' s autocommit mode. In this mode, *Connection.commit()* and *Connection.rollback()* have no effect. Note that SQLite' s autocommit mode is distinct from the **PEP 249**-compliant *Connection.autocommit* attribute; use *Connection.in_transaction* to query the low-level SQLite autocommit mode.

Set *autocommit* to *LEGACY_TRANSACTION_CONTROL* to leave transaction control behaviour to the *Connection.isolation_level* attribute. See *Transaction control via the isolation_level attribute* for more information.

Transaction control via the `isolation_level` attribute

> **참고**
>
> The recommended way of controlling transactions is via the *autocommit* attribute. See *Transaction control via the autocommit attribute*.

If *Connection.autocommit* is set to *LEGACY_TRANSACTION_CONTROL* (the default), transaction behaviour is controlled using the *Connection.isolation_level* attribute. Otherwise, `isolation_level` has no effect.

If the connection attribute *isolation_level* is not `None`, new transactions are implicitly opened before *execute()* and *executemany()* executes `INSERT`, `UPDATE`, `DELETE`, or `REPLACE` statements; for other statements, no implicit transaction handling is performed. Use the *commit()* and *rollback()* methods to respectively commit and roll back pending transactions. You can choose the underlying SQLite transaction behaviour — that is, whether and what type of `BEGIN` statements `sqlite3` implicitly executes – via the *isolation_level* attribute.

If *isolation_level* is set to `None`, no transactions are implicitly opened at all. This leaves the underlying SQLite library in autocommit mode, but also allows the user to perform their own transaction handling using explicit SQL statements. The underlying SQLite library autocommit mode can be queried using the *in_transaction* attribute.

The *executescript()* method implicitly commits any pending transaction before execution of the given SQL script, regardless of the value of *isolation_level*.

버전 3.6에서 변경: `sqlite3` used to implicitly commit an open transaction before DDL statements. This is no longer the case.

버전 3.12에서 변경: The recommended way of controlling transactions is now via the *autocommit* attribute.

CHAPTER 13

데이터 압축 및 보관

The modules described in this chapter support data compression with the zlib, gzip, bzip2, lzma, and zstd algorithms, and the creation of ZIP- and tar-format archives. See also 아카이브 연산 provided by the *shutil* module.

13.1 The compression package

Added in version 3.14.

The compression package contains the canonical compression modules containing interfaces to several different compression algorithms. Some of these modules have historically been available as separate modules; those will continue to be available under their original names for compatibility reasons, and will not be removed without a deprecation cycle. The use of modules in compression is encouraged where practical.

- compression.bz2 – Re-exports *bz2*
- compression.gzip – Re-exports *gzip*
- compression.lzma – Re-exports *lzma*
- compression.zlib – Re-exports *zlib*
- *compression.zstd* – Wrapper for the Zstandard compression library

13.2 compression.zstd — Compression compatible with the Zstandard format

Added in version 3.14.

Source code: Lib/compression/zstd/__init__.py

This module provides classes and functions for compressing and decompressing data using the Zstandard (or *zstd*) compression algorithm. The zstd manual describes Zstandard as "a fast lossless compression algorithm, targeting real-time compression scenarios at zlib-level and better compression ratios." Also included is a file interface that supports reading and writing the contents of .zst files created by the **zstd** utility, as well as raw zstd compressed streams.

The compression.zstd module contains:

- The *open()* function and *ZstdFile* class for reading and writing compressed files.

- The `ZstdCompressor` and `ZstdDecompressor` classes for incremental (de)compression.
- The `compress()` and `decompress()` functions for one-shot (de)compression.
- The `train_dict()` and `finalize_dict()` functions and the `ZstdDict` class to train and manage Zstandard dictionaries.
- The `CompressionParameter`, `DecompressionParameter`, and `Strategy` classes for setting advanced (de)compression parameters.

13.2.1 Exceptions

exception compression.zstd.**ZstdError**

This exception is raised when an error occurs during compression or decompression, or while initializing the (de)compressor state.

13.2.2 Reading and writing compressed files

compression.zstd.**open**(*file*, */*, *mode='rb'*, ***, *level=None*, *options=None*, *zstd_dict=None*, *encoding=None*, *errors=None*, *newline=None*)

Open a Zstandard-compressed file in binary or text mode, returning a *file object*.

The *file* argument can be either a file name (given as a `str`, `bytes` or *path-like* object), in which case the named file is opened, or it can be an existing file object to read from or write to.

The mode argument can be either `'rb'` for reading (default), `'wb'` for overwriting, `'ab'` for appending, or `'xb'` for exclusive creation. These can equivalently be given as `'r'`, `'w'`, `'a'`, and `'x'` respectively. You may also open in text mode with `'rt'`, `'wt'`, `'at'`, and `'xt'` respectively.

When reading, the *options* argument can be a dictionary providing advanced decompression parameters; see `DecompressionParameter` for detailed information about supported parameters. The *zstd_dict* argument is a `ZstdDict` instance to be used during decompression. When reading, if the *level* argument is not None, a `TypeError` will be raised.

When writing, the *options* argument can be a dictionary providing advanced decompression parameters; see `CompressionParameter` for detailed information about supported parameters. The *level* argument is the compression level to use when writing compressed data. Only one of *level* or *options* may be non-None. The *zstd_dict* argument is a `ZstdDict` instance to be used during compression.

In binary mode, this function is equivalent to the `ZstdFile` constructor: `ZstdFile(file, mode, ...)`. In this case, the *encoding*, *errors*, and *newline* parameters must not be provided.

In text mode, a `ZstdFile` object is created, and wrapped in an `io.TextIOWrapper` instance with the specified encoding, error handling behavior, and line endings.

class compression.zstd.**ZstdFile**(*file*, */*, *mode='rb'*, ***, *level=None*, *options=None*, *zstd_dict=None*)

Open a Zstandard-compressed file in binary mode.

A `ZstdFile` can wrap an already-open *file object*, or operate directly on a named file. The *file* argument specifies either the file object to wrap, or the name of the file to open (as a `str`, `bytes` or *path-like* object). If wrapping an existing file object, the wrapped file will not be closed when the `ZstdFile` is closed.

The *mode* argument can be either `'rb'` for reading (default), `'wb'` for overwriting, `'xb'` for exclusive creation, or `'ab'` for appending. These can equivalently be given as `'r'`, `'w'`, `'x'` and `'a'` respectively.

If *file* is a file object (rather than an actual file name), a mode of `'w'` does not truncate the file, and is instead equivalent to `'a'`.

When reading, the *options* argument can be a dictionary providing advanced decompression parameters; see `DecompressionParameter` for detailed information about supported parameters. The *zstd_dict* argument is a `ZstdDict` instance to be used during decompression. When reading, if the *level* argument is not None, a `TypeError` will be raised.

When writing, the *options* argument can be a dictionary providing advanced decompression parameters; see `CompressionParameter` for detailed information about supported parameters. The *level* argument is the

compression level to use when writing compressed data. Only one of *level* or *options* may be passed. The *zstd_dict* argument is a `ZstdDict` instance to be used during compression.

`ZstdFile` supports all the members specified by `io.BufferedIOBase`, except for `detach()` and `truncate()`. Iteration and the `with` statement are supported.

The following method and attributes are also provided:

peek(*size=-1*)

Return buffered data without advancing the file position. At least one byte of data will be returned, unless EOF has been reached. The exact number of bytes returned is unspecified (the *size* argument is ignored).

> **참고**
>
> While calling `peek()` does not change the file position of the `ZstdFile`, it may change the position of the underlying file object (for example, if the `ZstdFile` was constructed by passing a file object for *file*).

mode

`'rb'` for reading and `'wb'` for writing.

name

The name of the Zstandard file. Equivalent to the `name` attribute of the underlying *file object*.

13.2.3 Compressing and decompressing data in memory

`compression.zstd.`**compress**(*data*, *level=None*, *options=None*, *zstd_dict=None*)

Compress *data* (a *bytes-like object*), returning the compressed data as a `bytes` object.

The *level* argument is an integer controlling the level of compression. *level* is an alternative to setting `CompressionParameter.compression_level` in *options*. Use `bounds()` on `compression_level` to get the values that can be passed for *level*. If advanced compression options are needed, the *level* argument must be omitted and in the *options* dictionary the `CompressionParameter.compression_level` parameter should be set.

The *options* argument is a Python dictionary containing advanced compression parameters. The valid keys and values for compression parameters are documented as part of the `CompressionParameter` documentation.

The *zstd_dict* argument is an instance of `ZstdDict` containing trained data to improve compression efficiency. The function `train_dict()` can be used to generate a Zstandard dictionary.

`compression.zstd.`**decompress**(*data*, *zstd_dict=None*, *options=None*)

Decompress *data* (a *bytes-like object*), returning the uncompressed data as a `bytes` object.

The *options* argument is a Python dictionary containing advanced decompression parameters. The valid keys and values for compression parameters are documented as part of the `DecompressionParameter` documentation.

The *zstd_dict* argument is an instance of `ZstdDict` containing trained data used during compression. This must be the same Zstandard dictionary used during compression.

If *data* is the concatenation of multiple distinct compressed frames, decompress all of these frames, and return the concatenation of the results.

class `compression.zstd.`**ZstdCompressor**(*level=None*, *options=None*, *zstd_dict=None*)

Create a compressor object, which can be used to compress data incrementally.

For a more convenient way of compressing a single chunk of data, see the module-level function `compress()`.

The *level* argument is an integer controlling the level of compression. *level* is an alternative to setting `CompressionParameter.compression_level` in *options*. Use `bounds()` on `compression_level`

to get the values that can be passed for *level*. If advanced compression options are needed, the *level* argument must be omitted and in the *options* dictionary the `CompressionParameter.compression_level` parameter should be set.

The *options* argument is a Python dictionary containing advanced compression parameters. The valid keys and values for compression parameters are documented as part of the `CompressionParameter` documentation.

The *zstd_dict* argument is an optional instance of `ZstdDict` containing trained data to improve compression efficiency. The function `train_dict()` can be used to generate a Zstandard dictionary.

compress(*data*, *mode=ZstdCompressor.CONTINUE*)

Compress *data* (a *bytes-like object*), returning a `bytes` object with compressed data if possible, or otherwise an empty `bytes` object. Some of *data* may be buffered internally, for use in later calls to `compress()` and `flush()`. The returned data should be concatenated with the output of any previous calls to `compress()`.

The *mode* argument is a `ZstdCompressor` attribute, either `CONTINUE`, `FLUSH_BLOCK`, or `FLUSH_FRAME`.

When all data has been provided to the compressor, call the `flush()` method to finish the compression process. If `compress()` is called with *mode* set to `FLUSH_FRAME`, `flush()` should not be called, as it would write out a new empty frame.

flush(*mode=ZstdCompressor.FLUSH_FRAME*)

Finish the compression process, returning a `bytes` object containing any data stored in the compressor's internal buffers.

The *mode* argument is a `ZstdCompressor` attribute, either `FLUSH_BLOCK`, or `FLUSH_FRAME`.

set_pledged_input_size(*size*)

Specify the amount of uncompressed data *size* that will be provided for the next frame. *size* will be written into the frame header of the next frame unless `CompressionParameter.content_size_flag` is `False` or `0`. A size of `0` means that the frame is empty. If *size* is `None`, the frame header will omit the frame size. Frames that include the uncompressed data size require less memory to decompress, especially at higher compression levels.

If `last_mode` is not `FLUSH_FRAME`, a `ValueError` is raised as the compressor is not at the start of a frame. If the pledged size does not match the actual size of data provided to `compress()`, future calls to `compress()` or `flush()` may raise `ZstdError` and the last chunk of data may be lost.

After `flush()` or `compress()` are called with mode `FLUSH_FRAME`, the next frame will not include the frame size into the header unless `set_pledged_input_size()` is called again.

CONTINUE

Collect more data for compression, which may or may not generate output immediately. This mode optimizes the compression ratio by maximizing the amount of data per block and frame.

FLUSH_BLOCK

Complete and write a block to the data stream. The data returned so far can be immediately decompressed. Past data can still be referenced in future blocks generated by calls to `compress()`, improving compression.

FLUSH_FRAME

Complete and write out a frame. Future data provided to `compress()` will be written into a new frame and *cannot* reference past data.

last_mode

The last mode passed to either `compress()` or `flush()`. The value can be one of `CONTINUE`, `FLUSH_BLOCK`, or `FLUSH_FRAME`. The initial value is `FLUSH_FRAME`, signifying that the compressor is at the start of a new frame.

class compression.zstd.**ZstdDecompressor**(*zstd_dict=None*, *options=None*)

Create a decompressor object, which can be used to decompress data incrementally.

For a more convenient way of decompressing an entire compressed stream at once, see the module-level function `decompress()`.

The *options* argument is a Python dictionary containing advanced decompression parameters. The valid keys and values for compression parameters are documented as part of the `DecompressionParameter` documentation.

The *zstd_dict* argument is an instance of `ZstdDict` containing trained data used during compression. This must be the same Zstandard dictionary used during compression.

> **참고**
>
> This class does not transparently handle inputs containing multiple compressed frames, unlike the `decompress()` function and `ZstdFile` class. To decompress a multi-frame input, you should use `decompress()`, `ZstdFile` if working with a *file object*, or multiple `ZstdDecompressor` instances.

decompress(*data*, *max_length=-1*)

Decompress *data* (a *bytes-like object*), returning uncompressed data as bytes. Some of *data* may be buffered internally, for use in later calls to `decompress()`. The returned data should be concatenated with the output of any previous calls to `decompress()`.

If *max_length* is non-negative, the method returns at most *max_length* bytes of decompressed data. If this limit is reached and further output can be produced, the `needs_input` attribute will be set to `False`. In this case, the next call to `decompress()` may provide *data* as `b''` to obtain more of the output.

If all of the input data was decompressed and returned (either because this was less than *max_length* bytes, or because *max_length* was negative), the `needs_input` attribute will be set to `True`.

Attempting to decompress data after the end of a frame will raise a `ZstdError`. Any data found after the end of the frame is ignored and saved in the `unused_data` attribute.

eof

`True` if the end-of-stream marker has been reached.

unused_data

Data found after the end of the compressed stream.

Before the end of the stream is reached, this will be `b''`.

needs_input

`False` if the `decompress()` method can provide more decompressed data before requiring new compressed input.

13.2.4 Zstandard dictionaries

`compression.zstd.`**train_dict**(*samples*, *dict_size*)

Train a Zstandard dictionary, returning a `ZstdDict` instance. Zstandard dictionaries enable more efficient compression of smaller sizes of data, which is traditionally difficult to compress due to less repetition. If you are compressing multiple similar groups of data (such as similar files), Zstandard dictionaries can improve compression ratios and speed significantly.

The *samples* argument (an iterable of `bytes` objects), is the population of samples used to train the Zstandard dictionary.

The *dict_size* argument, an integer, is the maximum size (in bytes) the Zstandard dictionary should be. The Zstandard documentation suggests an absolute maximum of no more than 100 KB, but the maximum can often be smaller depending on the data. Larger dictionaries generally slow down compression, but improve compression ratios. Smaller dictionaries lead to faster compression, but reduce the compression ratio.

compression.zstd.**finalize_dict**(*zstd_dict*, /, *samples*, *dict_size*, *level*)

An advanced function for converting a "raw content" Zstandard dictionary into a regular Zstandard dictionary. "Raw content" dictionaries are a sequence of bytes that do not need to follow the structure of a normal Zstandard dictionary.

The *zstd_dict* argument is a `ZstdDict` instance with the `dict_content` containing the raw dictionary contents.

The *samples* argument (an iterable of `bytes` objects), contains sample data for generating the Zstandard dictionary.

The *dict_size* argument, an integer, is the maximum size (in bytes) the Zstandard dictionary should be. See `train_dict()` for suggestions on the maximum dictionary size.

The *level* argument (an integer) is the compression level expected to be passed to the compressors using this dictionary. The dictionary information varies for each compression level, so tuning for the proper compression level can make compression more efficient.

class compression.zstd.**ZstdDict**(*dict_content*, /, *, *is_raw=False*)

A wrapper around Zstandard dictionaries. Dictionaries can be used to improve the compression of many small chunks of data. Use `train_dict()` if you need to train a new dictionary from sample data.

The *dict_content* argument (a *bytes-like object*), is the already trained dictionary information.

The *is_raw* argument, a boolean, is an advanced parameter controlling the meaning of *dict_content*. `True` means *dict_content* is a "raw content" dictionary, without any format restrictions. `False` means *dict_content* is an ordinary Zstandard dictionary, created from Zstandard functions, for example, `train_dict()` or the external **zstd** CLI.

When passing a `ZstdDict` to a function, the `as_digested_dict` and `as_undigested_dict` attributes can control how the dictionary is loaded by passing them as the `zstd_dict` argument, for example, `compress(data, zstd_dict=zd.as_digested_dict)`. Digesting a dictionary is a costly operation that occurs when loading a Zstandard dictionary. When making multiple calls to compression or decompression, passing a digested dictionary will reduce the overhead of loading the dictionary.

표 1: Difference for compression

	Digested dictionary	Undigested dictionary
Advanced parameters of the compressor which may be overridden by the dictionary' s parameters	`window_log`, `hash_log`, `chain_log`, `search_log`, `min_match`, `target_length`, `strategy`, `enable_long_distance_matchin` `ldm_hash_log`, `ldm_min_match`, `ldm_bucket_size_log`, `ldm_hash_rate_log`, and some non-public parameters.	None
`ZstdDict` internally caches the dictionary	Yes. It' s faster when loading a digested dictionary again with the same compression level.	No. If you wish to load an undigested dictionary multiple times, consider reusing a compressor object.

If passing a `ZstdDict` without any attribute, an undigested dictionary is passed by default when compressing and a digested dictionary is generated if necessary and passed by default when decompressing.

dict_content

The content of the Zstandard dictionary, a `bytes` object. It' s the same as the *dict_content* argument in the `__init__` method. It can be used with other programs, such as the `zstd` CLI program.

dict_id

Identifier of the Zstandard dictionary, a non-negative int value.

Non-zero means the dictionary is ordinary, created by Zstandard functions and following the Zstandard format.

`0` means a "raw content" dictionary, free of any format restriction, used for advanced users.

> **참고**
>
> The meaning of `0` for `ZstdDict.dict_id` is different from the `dictionary_id` attribute to the *get_frame_info()* function.

as_digested_dict

Load as a digested dictionary.

as_undigested_dict

Load as an undigested dictionary.

13.2.5 Advanced parameter control

class compression.zstd.**CompressionParameter**

An *IntEnum* containing the advanced compression parameter keys that can be used when compressing data.

The *bounds()* method can be used on any attribute to get the valid values for that parameter.

Parameters are optional; any omitted parameter will have it' s value selected automatically.

Example getting the lower and upper bound of *compression_level*:

```
lower, upper = CompressionParameter.compression_level.bounds()
```

Example setting the *window_log* to the maximum size:

```
_lower, upper = CompressionParameter.window_log.bounds()
options = {CompressionParameter.window_log: upper}
compress(b'venezuelan beaver cheese', options=options)
```

bounds()

Return the tuple of int bounds, `(lower, upper)`, of a compression parameter. This method should be called on the attribute you wish to retrieve the bounds of. For example, to get the valid values for *compression_level*, one may check the result of `CompressionParameter.compression_level.bounds()`.

Both the lower and upper bounds are inclusive.

compression_level

A high-level means of setting other compression parameters that affect the speed and ratio of compressing data.

Regular compression levels are greater than `0`. Values greater than `20` are considered "ultra" compression and require more memory than other levels. Negative values can be used to trade off faster compression for worse compression ratios.

Setting the level to zero uses *COMPRESSION_LEVEL_DEFAULT*.

window_log

Maximum allowed back-reference distance the compressor can use when compressing data, expressed as power of two, `1 << window_log` bytes. This parameter greatly influences the memory usage of compression. Higher values require more memory but gain better compression values.

A value of zero causes the value to be selected automatically.

hash_log

Size of the initial probe table, as a power of two. The resulting memory usage is `1 << (hash_log+2)` bytes. Larger tables improve compression ratio of strategies <= *dfast*, and improve compression speed of strategies > *dfast*.

A value of zero causes the value to be selected automatically.

chain_log

Size of the multi-probe search table, as a power of two. The resulting memory usage is `1 << (chain_log+2)` bytes. Larger tables result in better and slower compression. This parameter has no effect for the *fast* strategy. It' s still useful when using *dfast* strategy, in which case it defines a secondary probe table.

A value of zero causes the value to be selected automatically.

search_log

Number of search attempts, as a power of two. More attempts result in better and slower compression. This parameter is useless for *fast* and *dfast* strategies.

A value of zero causes the value to be selected automatically.

min_match

Minimum size of searched matches. Larger values increase compression and decompression speed, but decrease ratio. Note that Zstandard can still find matches of smaller size, it just tweaks its search algorithm to look for this size and larger. For all strategies < *btopt*, the effective minimum is `4`; for all strategies > *fast*, the effective maximum is `6`.

A value of zero causes the value to be selected automatically.

target_length

The impact of this field depends on the selected *Strategy*.

For strategies *btopt*, *btultra* and *btultra2*, the value is the length of a match considered "good enough" to stop searching. Larger values make compression ratios better, but compresses slower.

For strategy *fast*, it is the distance between match sampling. Larger values make compression faster, but with a worse compression ratio.

A value of zero causes the value to be selected automatically.

strategy

The higher the value of selected strategy, the more complex the compression technique used by zstd, resulting in higher compression ratios but slower compression.

> **더 보기**
>
> *Strategy*

enable_long_distance_matching

Long distance matching can be used to improve compression for large inputs by finding large matches at greater distances. It increases memory usage and window size.

`True` or `1` enable long distance matching while `False` or `0` disable it.

Enabling this parameter increases default *window_log* to 128 MiB except when expressly set to a different value. This setting is enabled by default if `window_log` >= 128 MiB and the compression strategy >= *btopt* (compression level 16+).

ldm_hash_log

Size of the table for long distance matching, as a power of two. Larger values increase memory usage and compression ratio, but decrease compression speed.

A value of zero causes the value to be selected automatically.

ldm_min_match

Minimum match size for long distance matcher. Larger or too small values can often decrease the compression ratio.

A value of zero causes the value to be selected automatically.

ldm_bucket_size_log

Log size of each bucket in the long distance matcher hash table for collision resolution. Larger values improve collision resolution but decrease compression speed.

A value of zero causes the value to be selected automatically.

ldm_hash_rate_log

Frequency of inserting/looking up entries into the long distance matcher hash table. Larger values improve compression speed. Deviating far from the default value will likely result in a compression ratio decrease.

A value of zero causes the value to be selected automatically.

content_size_flag

Write the size of the data to be compressed into the Zstandard frame header when known prior to compressing.

This flag only takes effect under the following scenarios:

- Calling *compress()* for one-shot compression
- Providing all of the data to be compressed in the frame in a single *ZstdCompressor.compress()* call, with the *ZstdCompressor.FLUSH_FRAME* mode.
- Calling *ZstdCompressor.set_pledged_input_size()* with the exact amount of data that will be provided to the compressor prior to any calls to *ZstdCompressor.compress()* for the current frame. `ZstdCompressor.set_pledged_input_size()` must be called for each new frame.

All other compression calls may not write the size information into the frame header.

`True` or `1` enable the content size flag while `False` or `0` disable it.

checksum_flag

A four-byte checksum using XXHash64 of the uncompressed content is written at the end of each frame. Zstandard' s decompression code verifies the checksum. If there is a mismatch a *ZstdError* exception is raised.

`True` or `1` enable checksum generation while `False` or `0` disable it.

dict_id_flag

When compressing with a *ZstdDict*, the dictionary' s ID is written into the frame header.

`True` or `1` enable storing the dictionary ID while `False` or `0` disable it.

nb_workers

Select how many threads will be spawned to compress in parallel. When `nb_workers` > 0, enables multi-threaded compression, a value of `1` means "one-thread multi-threaded mode". More workers improve speed, but also increase memory usage and slightly reduce compression ratio.

A value of zero disables multi-threading.

job_size

Size of a compression job, in bytes. This value is enforced only when *nb_workers* >= 1. Each compression job is completed in parallel, so this value can indirectly impact the number of active threads.

A value of zero causes the value to be selected automatically.

overlap_log

Sets how much data is reloaded from previous jobs (threads) for new jobs to be used by the look behind window during compression. This value is only used when *nb_workers* >= 1. Acceptable values vary from 0 to 9.

- 0 means dynamically set the overlap amount
- 1 means no overlap
- 9 means use a full window size from the previous job

Each increment halves/doubles the overlap size. "8" means an overlap of `window_size/2`, "7" means an overlap of `window_size/4`, etc.

class compression.zstd.**DecompressionParameter**

An *IntEnum* containing the advanced decompression parameter keys that can be used when decompressing data. Parameters are optional; any omitted parameter will have it' s value selected automatically.

The *bounds()* method can be used on any attribute to get the valid values for that parameter.

Example setting the *window_log_max* to the maximum size:

```
data = compress(b'Some very long buffer of bytes...')

_lower, upper = DecompressionParameter.window_log_max.bounds()

options = {DecompressionParameter.window_log_max: upper}
decompress(data, options=options)
```

bounds()

Return the tuple of int bounds, `(lower, upper)`, of a decompression parameter. This method should be called on the attribute you wish to retrieve the bounds of.

Both the lower and upper bounds are inclusive.

window_log_max

The base-two logarithm of the maximum size of the window used during decompression. This can be useful to limit the amount of memory used when decompressing data. A larger maximum window size leads to faster decompression.

A value of zero causes the value to be selected automatically.

class compression.zstd.**Strategy**

An *IntEnum* containing strategies for compression. Higher-numbered strategies correspond to more complex and slower compression.

> **참고**
>
> The values of attributes of `Strategy` are not necessarily stable across zstd versions. Only the ordering of the attributes may be relied upon. The attributes are listed below in order.

The following strategies are available:

fast

dfast

greedy

lazy

lazy2

btlazy2

btopt

btultra

btultra2

13.2.6 Miscellaneous

compression.zstd.**get_frame_info**(*frame_buffer*)

Retrieve a `FrameInfo` object containing metadata about a Zstandard frame. Frames contain metadata related to the compressed data they hold.

class compression.zstd.**FrameInfo**

Metadata related to a Zstandard frame.

decompressed_size

The size of the decompressed contents of the frame.

dictionary_id

An integer representing the Zstandard dictionary ID needed for decompressing the frame. `0` means the dictionary ID was not recorded in the frame header. This may mean that a Zstandard dictionary is not needed, or that the ID of a required dictionary was not recorded.

compression.zstd.**COMPRESSION_LEVEL_DEFAULT**

The default compression level for Zstandard: `3`.

compression.zstd.**zstd_version_info**

Version number of the runtime zstd library as a tuple of integers (major, minor, release).

13.2.7 Examples

Reading in a compressed file:

```
from compression import zstd

with zstd.open("file.zst") as f:
    file_content = f.read()
```

Creating a compressed file:

```
from compression import zstd

data = b"Insert Data Here"
with zstd.open("file.zst", "w") as f:
    f.write(data)
```

Compressing data in memory:

```
from compression import zstd

data_in = b"Insert Data Here"
data_out = zstd.compress(data_in)
```

Incremental compression:

```
from compression import zstd

comp = zstd.ZstdCompressor()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
out1 = comp.compress(b"Some data\n")
out2 = comp.compress(b"Another piece of data\n")
out3 = comp.compress(b"Even more data\n")
out4 = comp.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

Writing compressed data to an already-open file:

```
from compression import zstd

with open("myfile", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with zstd.open(f, "w") as zstf:
        zstf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

Creating a compressed file using compression parameters:

```
from compression import zstd

options = {
   zstd.CompressionParameter.checksum_flag: 1
}
with zstd.open("file.zst", "w", options=options) as f:
    f.write(b"Mind if I squeeze in?")
```

13.3 zlib — gzip 과 호환되는 압축

데이터 압축이 필요한 응용 프로그램의 경우, 이 모듈의 함수는 zlib 라이브러리를 사용하여 압축과 압축 해제를 하도록 합니다. zlib 라이브러리는 https://www.zlib.net 에 자체 홈페이지가 있습니다. 파이썬 모듈과 1.1.3 이전의 zlib 라이브러리 버전 간에는 호환성 문제가 알려져 있습니다; 1.1.3에는 보안 취약점이 있기 때문에, 1.1.4 이상을 사용하는 것이 좋습니다.

zlib의 함수에는 많은 옵션이 있으며 종종 특정 순서로 사용해야 합니다. 이 설명서는 모든 순열을 다루려고 시도하지는 않습니다; 권위 있는 정보는 zlib 매뉴얼(http://www.zlib.net/manual.html)을 참조하십시오.

`.gz` 파일 읽기와 쓰기에 대해서는 *gzip* 모듈을 참조하십시오.

이 모듈에서 사용 가능한 예외와 함수는 다음과 같습니다:

exception zlib.**error**

압축과 압축 해제 에러에서 발생하는 예외.

zlib.**adler32**(*data*[, *value*])

*data*의 Adler-32 체크섬을 계산합니다. (Adler-32 체크섬은 CRC32만큼 신뢰성 있지만, 훨씬 빠르게 계산할 수 있습니다.) 결과는 부호 없는 32비트 정수입니다. *value*가 있으면, 체크섬의 시작 값으로 사용됩니다; 그렇지 않으면, 기본값 1이 사용됩니다. *value*를 전달하면 여러 입력을 이어붙인 것에 대한 잇따른(running) 체크섬을 계산할 수 있습니다. 알고리즘은 암호학적으로 강력하지 않아서, 인증이나 디지털 서명에 사용해서는 안 됩니다. 알고리즘은 체크섬 알고리즘으로 사용하도록 설계되었으므로, 일반적인 해시 알고리즘으로 사용하기에 적합하지 않습니다.

버전 3.0에서 변경: 결과는 항상 부호 없는 값입니다.

zlib.**compress**(*data*, /, *level=-1*, *wbits=MAX_WBITS*)

*data*에 있는 바이트열을 압축하여, 압축된 데이터를 포함하는 바이트열 객체를 반환합니다. *level*은 압축 수준을 제어하는 `0`에서 `9` 또는 `-1`인 정수입니다; `1`(Z_BEST_SPEED)은 가장 빠르

고 압축률이 가장 낮습니다, 9(Z_BEST_COMPRESSION)는 가장 느리고 최대 압축을 생성합니다. 0(Z_NO_COMPRESSION)은 압축하지 않습니다. 기본값은 -1(Z_DEFAULT_COMPRESSION)입니다. Z_DEFAULT_COMPRESSION은 속도와 압축률 사이의 기본 절충(현재 수준 6과 동등합니다)을 나타냅니다.

wbits 인자는 데이터를 압축할 때 사용되는 히스토리 버퍼의 크기(또는 "창 크기(window size)")와, 출력에 헤더와 트레일러가 포함되는지를 제어합니다. 여러 범위의 값을 취할 수 있으며, 기본값은 15(MAX_WBITS)입니다:

- +9 에서 +15: 창 크기의 밑이 2인 로그, 그래서 창 크기는 512에서 32768 사이의 범위입니다. 값이 클수록 메모리 사용량이 증가하면서 압축률이 높아집니다. 결과 출력에는 zlib 특정 헤더와 트레일러가 포함됩니다.
- −9 에서 −15: *wbits*의 절댓값을 창 크기의 로그로 사용하면서, 헤더나 후행 체크섬 없이 원시 출력 스트림을 생성합니다.
- +25 에서 +31 = 16 + (9 에서 15): 하위 4비트를 창 크기의 로그로 사용하면서, 출력에 기본 **gzip** 헤더와 후행 체크섬을 포함합니다.

에러가 발생하면 `error` 예외를 발생시킵니다.

버전 3.6에서 변경: *level*은 이제 키워드 매개 변수로 사용될 수 있습니다.

버전 3.11에서 변경: The *wbits* parameter is now available to set window bits and compression type.

zlib.**compressobj**(*level=-1*, *method=DEFLATED*, *wbits=MAX_WBITS*, *memLevel=DEF_MEM_LEVEL*, *strategy=Z_DEFAULT_STRATEGY*[, *zdict*])

메모리에 한 번에 맞지 않는 데이터 스트림을 압축하는 데 사용되는 압축 객체를 반환합니다.

*level*은 압축 수준입니다 – 0에서 9 또는 -1인 정수입니다. 1(Z_BEST_SPEED) 값은 가장 빠르고 압축률이 가장 낮지만, 9(Z_BEST_COMPRESSION) 값은 가장 느리고 최대 압축을 생성합니다. 0(Z_NO_COMPRESSION)은 압축하지 않습니다. 기본값은 -1(Z_DEFAULT_COMPRESSION)입니다. Z_DEFAULT_COMPRESSION은 속도와 압축률 사이의 기본 절충(현재 수준 6과 동등합니다)을 나타냅니다.

*method*는 압축 알고리즘입니다. 현재, 유일하게 지원되는 값은 `DEFLATED`입니다.

wbits 매개 변수는 히스토리 버퍼의 크기(또는 "창 크기(window size)")와, 어떤 헤더와 트레일러 형식을 사용할지를 제어합니다. *compress()에서 설명된 것*과 같은 의미입니다.

memLevel 인자는 내부 압축 상태에 사용되는 메모리양을 제어합니다. 유효한 값의 범위는 1에서 9입니다. 값이 클수록 더 많은 메모리를 사용하지만, 더 빠르고 더 작은 출력을 생성합니다.

*strategy*는 압축 알고리즘을 조정하는 데 사용됩니다. 가능한 값은 `Z_DEFAULT_STRATEGY`, `Z_FILTERED`, `Z_HUFFMAN_ONLY`, `Z_RLE`(zlib 1.2.0.1) 및 `Z_FIXED`(zlib 1.2.2.2)입니다.

*zdict*는 사전 정의된 압축 딕셔너리입니다. 이것은 압축될 데이터에서 자주 나타날 것으로 예상되는 서브 시퀀스를 포함하는 일련의 바이트 시퀀스(가령 `bytes` 객체)입니다. 가장 흔할 것으로 예상되는 서브 시퀀스는 딕셔너리 끝에 와야 합니다.

버전 3.3에서 변경: *zdict* 매개 변수와 키워드 인자 지원이 추가되었습니다.

zlib.**crc32**(*data*[, *value*])

*data*의 CRC (Cyclic Redundancy Check; 순환 중복 검사) 체크섬을 계산합니다. 결과는 부호 없는 32비트 정수입니다. *value*가 있으면, 체크섬의 시작 값으로 사용됩니다; 그렇지 않으면, 기본값 1이 사용됩니다. *value*를 전달하면 여러 입력을 이어붙인 것에 대한 잇따른(running) 체크섬을 계산할 수 있습니다. 알고리즘은 암호학적으로 강력하지 않아서, 인증이나 디지털 서명에 사용해서는 안 됩니다. 알고리즘은 체크섬 알고리즘으로 사용하도록 설계되었으므로, 일반적인 해시 알고리즘으로 사용하기에 적합하지 않습니다.

버전 3.0에서 변경: 결과는 항상 부호 없는 값입니다.

zlib.**decompress**(*data*, */*, *wbits=MAX_WBITS*, *bufsize=DEF_BUF_SIZE*)

*data*에 있는 바이트열을 압축 해제하여, 압축되지 않은 데이터를 포함하는 바이트열 객체를 반환합니다. *wbits* 매개 변수는 *data*의 형식에 따라 다르며, 아래에서 자세히 설명합니다. *bufsize*가 제공되면, 출력 버퍼의 초기 크기로 사용됩니다. 에러가 발생하면 `error` 예외를 발생시킵니다.

wbits 인자는 히스토리 버퍼의 크기(또는 "창 크기(window size)")와, 어떤 헤더와 트레일러 형식을 기대하는지를 제어합니다. `compressobj()`의 매개 변수와 유사하지만, 더 많은 범위의 값을 받아들입니다:

- +8 에서 +15: 창 크기의 밑이 2인 로그. 입력은 zlib 헤더와 트레일러를 포함해야 합니다.
- 0: zlib 헤더에서 창 크기를 자동으로 결정합니다. zlib 1.2.3.5부터 지원됩니다.
- −8 에서 −15: *wbits*의 절댓값을 창 크기의 로그로 사용합니다. 입력은 헤더나 트레일러가 없는 원시 스트림이어야 합니다.
- +24 에서 +31 = 16 + (8 에서 15): 값의 하위 4비트를 창 크기의 로그로 사용합니다. 입력은 gzip 헤더와 트레일러를 포함해야 합니다.
- +40 에서 +47 = 32 + (8 에서 15): 값의 하위 4비트를 창 크기의 로그로 사용하고, zlib나 gzip 형식을 자동으로 받아들입니다.

스트림을 압축 해제할 때, 창 크기는 스트림을 압축하는 데 원래 사용된 크기보다 작아서는 안 됩니다; 너무 작은 값을 사용하면 `error` 예외가 발생할 수 있습니다. 기본 *wbits* 값은 가장 큰 창 크기에 해당하며 zlib 헤더와 트레일러가 포함될 것을 요구합니다.

*bufsize*는 압축 해제된 데이터를 담는 데 사용되는 버퍼의 초기 크기입니다. 더 많은 공간이 필요하면, 필요에 따라 버퍼 크기가 증가하므로, 이 값을 정확하게 얻을 필요는 없습니다; 조정하면 `malloc()`에 대한 몇 번의 호출만 절약됩니다.

버전 3.6에서 변경: *wbits*와 *bufsize*는 키워드 인자로 사용할 수 있습니다.

`zlib.`**`decompressobj`**(*wbits=MAX_WBITS*[, *zdict*])

메모리에 한 번에 맞지 않는 데이터 스트림을 압축 해제하는 데 사용되는 압축 해제 객체를 반환합니다.

wbits 인자는 히스토리 버퍼의 크기(또는 "창 크기(window size)")와, 어떤 헤더와 트레일러 형식을 기대하는지를 제어합니다. *decompress()에서 설명된 것*과 같은 의미입니다.

zdict 매개 변수는 사전 정의된 압축 딕셔너리를 지정합니다. 제공되면□ 압축 해제할 데이터를 생성한 압축기에서 사용한 것과 같은 딕셔너리이어야 합니다.

> **참고**
>
> *zdict*가 가변 객체(가령 `bytearray`)이면, `decompressobj()`에 대한 호출과 압축 해제기의 `decompress()` 메서드에 대한 첫 번째 호출 사이에 내용을 수정해서는 안 됩니다.

버전 3.3에서 변경: *zdict* 매개 변수를 추가했습니다.

압축 객체는 다음 메서드를 지원합니다:

`Compress.`**`compress`**(*data*)

*data*를 압축하여, *data*의 데이터 중 적어도 일부에 대한 압축된 데이터가 포함된 바이트열 객체를 반환합니다. 이 데이터는 `compress()` 메서드에 대한 이전 호출에서 생성된 출력에 이어붙여야 합니다. 일부 입력은 나중에 처리하기 위해 내부 버퍼에 보관될 수 있습니다.

`Compress.`**`flush`**([*mode*])

계류 중인 모든 입력이 처리되고, 나머지 압축 출력을 포함하는 바이트열 객체가 반환됩니다. *mode*는 상수 `Z_NO_FLUSH`, `Z_PARTIAL_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH`, `Z_BLOCK`(zlib 1.2.3.4) 또는 `Z_FINISH`에서 선택할 수 있으며, 기본값은 `Z_FINISH`입니다. `Z_FINISH`를 제외한 모든 상수는 추가 바이트열 데이터를 압축하도록 허락하는 반면, `Z_FINISH`는 압축된 스트림을 완료하고 추가 데이터의 압축을 방지합니다. *mode*를 `Z_FINISH`로 설정하고 `flush()`를 호출한 후, `compress()` 메서드를 다시 호출할 수 없습니다; 유일한 현실적인 조치는 객체를 삭제하는 것입니다.

`Compress.`**`copy`**()

압축 객체의 복사본을 반환합니다. 공통 초기 접두사를 공유하는 데이터 집합을 효율적으로 압축하는 데 사용할 수 있습니다.

버전 3.8에서 변경: 압축 객체에 `copy.copy()`와 `copy.deepcopy()` 지원이 추가되었습니다.

압축 해제 객체는 다음과 같은 메서드와 어트리뷰트를 지원합니다:

`Decompress.`**`unused_data`**

압축된 데이터가 끝난 뒤의 바이트를 포함하는 바이트열 객체. 즉, 압축 데이터가 들어 있는 마지막 바이트를 사용할 수 있을 때까지 `b""`로 남습니다. 전체 바이트열이 압축된 데이터를 포함하는 것으로 판명되면, 이것은 빈 바이트열 객체인 `b""`입니다.

`Decompress.`**`unconsumed_tail`**

압축되지 않은 데이터 버퍼의 한계를 초과하기 때문에 마지막 `decompress()` 호출에 의해 소비되지 않은 모든 데이터를 포함하는 바이트열 객체. 이 데이터는 아직 zlib 장치가 볼 수 없었기 때문에 올바른 출력을 얻으려면 후속 `decompress()` 메서드 호출에 다시 공급해야 합니다 (아마도 추가 데이터를 이것에 이어붙여서).

`Decompress.`**`eof`**

압축된 데이터 스트림의 끝에 도달했는지를 나타내는 불리언.

올바르게 구성된 압축 스트림과 불완전하거나 잘린 스트림을 구별할 수 있도록 합니다.

Added in version 3.3.

`Decompress.`**`decompress`**(*data*, *max_length=0*)

*data*를 압축 해제하여, *string*의 데이터 중 적어도 일부에 해당하는 압축되지 않은 데이터를 포함하는 바이트열 객체를 반환합니다. 이 데이터는 `decompress()` 메서드에 대한 이전 호출에서 생성된 출력에 이어붙여야 합니다. 입력 데이터 중 일부는 나중에 처리하기 위해 내부 버퍼에 보존될 수 있습니다.

선택적 매개 변수 *max_length*가 0이 아니면 반환 값은 *max_length*보다 길지 않습니다. 이는 모든 압축 입력을 처리할 수 없음을 뜻합니다; 소비되지 않은 데이터는 `unconsumed_tail` 어트리뷰트에 저장됩니다. 압축 해제를 계속하려면 이 바이트열을 `decompress()`에 대한 후속 호출로 전달해야 합니다. *max_length*가 0이면 전체 입력이 압축 해제되고, `unconsumed_tail`은 비어 있습니다.

버전 3.6에서 변경: *max_length*는 키워드 인자로 사용할 수 있습니다.

`Decompress.`**`flush`**([*length*])

계류 중인 모든 입력이 처리되고, 나머지 압축되지 않은 출력을 포함하는 바이트열 객체가 반환됩니다. `flush()`를 호출한 후, `decompress()` 메서드를 다시 호출할 수 없습니다; 유일한 현실적인 조치는 객체를 삭제하는 것입니다.

선택적 매개 변수 *length*는 출력 버퍼의 초기 크기를 설정합니다.

`Decompress.`**`copy`**()

압축 해제 객체의 복사본을 반환합니다. 이것은 미래 시점에 스트림으로의 임의 탐색(random seek) 속도를 높이기 위해 데이터 스트림의 중간 지점에서 압축 해제기의 상태를 저장하는 데 사용될 수 있습니다.

버전 3.8에서 변경: 압축 해제 객체에 `copy.copy()`와 `copy.deepcopy()` 지원이 추가되었습니다.

사용 중인 zlib 라이브러리 버전에 대한 정보는 다음 상수를 통해 사용할 수 있습니다:

`zlib.`**`ZLIB_VERSION`**

모듈을 빌드하는 데 사용된 zlib 라이브러리의 버전 문자열. `ZLIB_RUNTIME_VERSION`으로 사용 가능한, 실행 시간에 실제로 사용되는 zlib 라이브러리와 다를 수 있습니다.

`zlib.`**`ZLIB_RUNTIME_VERSION`**

인터프리터가 실제로 로드한 zlib 라이브러리의 버전 문자열.

Added in version 3.3.

`zlib.`**`ZLIBNG_VERSION`**

The version string of the zlib-ng library that was used for building the module if zlib-ng was used. When present, the `ZLIB_VERSION` and `ZLIB_RUNTIME_VERSION` constants reflect the version of the zlib API provided by zlib-ng.

If zlib-ng was not used to build the module, this constant will be absent.

Added in version 3.14.

더 보기

모듈 *gzip*
gzip 형식 파일 읽기와 쓰기

http://www.zlib.net
zlib 라이브러리 홈페이지.

http://www.zlib.net/manual.html
zlib 매뉴얼은 라이브러리의 많은 함수의 의미와 사용법을 설명합니다.

In case gzip (de)compression is a bottleneck, the python-isal package speeds up (de)compression with a mostly compatible API.

13.4 gzip — gzip 파일 지원

소스 코드: Lib/gzip.py

이 모듈은 GNU 프로그램 **gzip**과 **gunzip**처럼 파일을 압축하고 압축을 푸는 간단한 인터페이스를 제공합니다.

데이터 압축은 *zlib* 모듈에 의해 제공됩니다.

gzip 모듈은 *open()*, *compress()* 및 *decompress()* 편리 함수뿐만 아니라 *GzipFile* 클래스도 제공합니다. *GzipFile* 클래스는 **gzip**-형식 파일을 읽고 쓰는데, 자동으로 데이터를 압축하거나 압축을 풀어서 일반적인 파일 객체처럼 보이게 합니다.

compress와 **pack** 프로그램에서 생성된 것과 같은, **gzip**과 **gunzip** 프로그램으로 압축을 풀 수 있는 추가 파일 형식은 이 모듈에서 지원하지 않습니다.

이 모듈은 다음 항목을 정의합니다:

gzip.**open**(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)

바이너리나 텍스트 모드로 gzip으로 압축된 파일을 열고, 파일 객체를 반환합니다.

filename 인자는 실제 파일명(*str*이나 *bytes* 객체)이나, 읽거나 쓸 기존 파일 객체가 될 수 있습니다.

mode 인자는 바이너리 모드의 경우 `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'` 또는 `'xb'`, 또는 텍스트 모드의 경우 `'rt'`, `'at'`, `'wt'` 또는 `'xt'` 중 하나일 수 있습니다. 기본값은 `'rb'`입니다.

compresslevel 인자는 *GzipFile* 생성자와 마찬가지로 0에서 9 사이의 정수입니다.

바이너리 모드의 경우, 이 함수는 *GzipFile* 생성자 `GzipFile(filename, mode, compresslevel)`와 동등합니다. 이 경우, *encoding*, *errors* 및 *newline* 인자를 제공하면 안 됩니다.

텍스트 모드의 경우, *GzipFile* 객체가 만들어지고, 지정된 인코딩, 에러 처리 동작 및 줄 종료를 갖는 *io.TextIOWrapper* 인스턴스로 감싸집니다.

버전 3.3에서 변경: 파일 객체인 *filename* 지원, 텍스트 모드 지원 및 *encoding*, *errors* 및 *newline* 인자가 추가되었습니다.

버전 3.4에서 변경: `'x'`, `'xb'` 및 `'xt'` 모드에 대한 지원이 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

exception gzip.**BadGzipFile**

유효하지 않은 gzip 파일에 대한 예외. *OSError*를 상속합니다. *EOFError*와 *zlib.error*도 유효하지 않은 gzip 파일에 대해서 발생할 수 있습니다.

Added in version 3.8.

class gzip.**GzipFile**(*filename=None, mode=None, compresslevel=9, fileobj=None, mtime=None*)

`truncate()` 메서드를 제외하고, 대부분 파일 객체 메서드를 흉내 내는 `GzipFile` 클래스의 생성자입니다. *fileobj*와 *filename* 중 적어도 하나는 의미 있는 값을 부여해야 합니다.

새 클래스 인스턴스는 *fileobj*를 기반으로 하는데, 일반 파일, `io.BytesIO` 객체 또는 파일을 흉내 내는 다른 객체가 될 수 있습니다. 기본값은 `None`이며, 이 경우 파일 객체를 제공하기 위해 *filename*이 열립니다.

*fileobj*가 `None`이 아닐 때, *filename* 인자는 **gzip** 파일 헤더에 포함되는 데만 사용되며, 이 헤더에는 압축되지 않은 파일의 원래 파일명이 포함될 수 있습니다. 보고 알 수 있다면, *fileobj*의 파일명을 기본값으로 사용합니다; 그렇지 않으면, 기본값은 빈 문자열이며, 이 경우 원래 파일명은 헤더에 포함되지 않습니다.

mode 인자는 파일을 읽을지 쓸지에 따라 `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'` 또는 `'xb'` 중 하나일 수 있습니다. 보고 알 수 있다면, 기본값은 *fileobj*의 모드입니다; 그렇지 않으면, 기본값은 `'rb'`입니다. 향후 파이썬 릴리스에서는 *fileobj*의 모드가 사용되지 않습니다. 항상 쓰기를 위해서는 *mode*를 지정하는 것이 좋습니다.

파일이 항상 바이너리 모드로 열림에 유의하십시오. 텍스트 모드로 압축 파일을 열려면, `open()`을 사용하십시오 (또는 `GzipFile`을 `io.TextIOWrapper`로 감싸십시오).

compresslevel 인자는 압축 수준을 제어하는 `0`에서 `9`까지의 정수입니다; `1`은 가장 빠르고 압축률이 가장 낮으며, `9`는 가장 느리고 압축률이 가장 높습니다. `0`은 압축하지 않습니다. 기본값은 `9`입니다.

The optional *mtime* argument is the timestamp requested by gzip. The time is in Unix format, i.e., seconds since 00:00:00 UTC, January 1, 1970. If *mtime* is omitted or `None`, the current time is used. Use *mtime* = 0 to generate a compressed stream that does not depend on creation time.

See below for the `mtime` attribute that is set when decompressing.

`GzipFile` 객체의 `close()` 메서드를 호출해도 *fileobj*를 닫지 않습니다, 압축된 데이터 뒤에 뭔가 추가하기를 원할 수 있기 때문입니다. 또한, 이는 *fileobj*로 쓰기 위해 열린 `io.BytesIO` 객체를 전달하고, `io.BytesIO` 객체의 `getvalue()` 메서드를 사용하여 결과 메모리 버퍼를 얻을 수 있도록 합니다.

`GzipFile`은 이터레이션과 `with` 문을 포함하여 `io.BufferedIOBase` 인터페이스를 지원합니다. `truncate()` 메서드 만 구현되지 않습니다.

`GzipFile`은 다음 메서드와 어트리뷰트도 제공합니다:

peek(*n*)

파일 위치를 전진시키지 않고 압축되지 않은 *n* 바이트를 읽습니다. 반환된 바이트 수는 요청한 것보다 많거나 적을 수 있습니다.

> **참고**
>
> `peek()`를 호출할 때 `GzipFile`의 파일 위치가 변경되지는 않지만, 하부 파일 객체의 위치는 변경될 수 있습니다 (예를 들어, `GzipFile`이 *fileobj* 매개 변수로 생성된 경우).

Added in version 3.2.

mode

`'rb'` for reading and `'wb'` for writing.

버전 3.13에서 변경: In previous versions it was an integer `1` or `2`.

mtime

압축을 풀 때, 이 어트리뷰트는 가장 최근에 읽은 헤더의 마지막 타임스탬프로 설정됩니다. 유닉스 에포크 (1970-01-01 00:00:00 UTC) 이후의 시간(초)를 나타내는 정수입니다. 헤더를 읽기 전의 초깃값은 `None`입니다.

name

The path to the gzip file on disk, as a `str` or `bytes`. Equivalent to the output of `os.fspath()` on the original input path, with no other normalization, resolution or expansion.

버전 3.1에서 변경: *mtime* 생성자 인자와 `mtime` 어트리뷰트와 함께 `with` 문에 대한 지원이 추가되었습니다.

버전 3.2에서 변경: 제로 패딩(zero-padded)된 파일과 위치 변경할 수 없는(unseekable) 파일에 대한 지원이 추가되었습니다.

버전 3.3에서 변경: `io.BufferedIOBase.read1()` 메서드가 이제 구현됩니다.

버전 3.4에서 변경: `'x'` 및 `'xb'` 모드에 대한 지원이 추가되었습니다.

버전 3.5에서 변경: 임의의 바이트열류 객체를 쓰는 지원이 추가되었습니다. 이제 `read()` 메서드는 `None` 인자를 받아들입니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.9부터 폐지됨: *mode* 인자를 지정하지 않고 쓰기 위해 `GzipFile`을 여는 것은 폐지되었습니다.

버전 3.12에서 변경: Remove the `filename` attribute, use the `name` attribute instead.

`gzip.`**`compress`**(*data, compresslevel=9, *, mtime=0*)

Compress the *data*, returning a `bytes` object containing the compressed data. *compresslevel* and *mtime* have the same meaning as in the `GzipFile` constructor above, but *mtime* defaults to 0 for reproducible output.

Added in version 3.2.

버전 3.8에서 변경: 재현성 있는 출력을 위한 *mtime* 매개 변수가 추가되었습니다.

버전 3.11에서 변경: Speed is improved by compressing all data at once instead of in a streamed fashion. Calls with *mtime* set to `0` are delegated to `zlib.compress()` for better speed. In this situation the output may contain a gzip header "OS" byte value other than 255 "unknown" as supplied by the underlying zlib implementation.

버전 3.13에서 변경: The gzip header OS byte is guaranteed to be set to 255 when this function is used as was the case in 3.10 and earlier.

버전 3.14에서 변경: The *mtime* parameter now defaults to 0 for reproducible output. For the previous behaviour of using the current time, pass `None` to *mtime*.

`gzip.`**`decompress`**(*data*)

Decompress the *data*, returning a `bytes` object containing the uncompressed data. This function is capable of decompressing multi-member gzip data (multiple gzip blocks concatenated together). When the data is certain to contain only one member the `zlib.decompress()` function with *wbits* set to 31 is faster.

Added in version 3.2.

버전 3.11에서 변경: Speed is improved by decompressing members at once in memory instead of in a streamed fashion.

13.4.1 사용 예

압축된 파일을 읽는 방법의 예:

```
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

압축된 GZIP 파일을 만드는 방법의 예:

```
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

기존 파일을 GZIP 압축하는 방법의 예:

```
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

바이너리 문자열을 GZIP 압축하는 방법의 예:

```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

더 보기

모듈 *zlib*
gzip 파일 형식을 지원하는 데 필요한 기본 데이터 압축 모듈.

In case gzip (de)compression is a bottleneck, the python-isal package speeds up (de)compression with a mostly compatible API.

13.4.2 명령 줄 인터페이스

gzip 모듈은 파일을 압축하거나 압축 해제하는 간단한 명령 줄 인터페이스를 제공합니다.

일단 실행되면 *gzip* 모듈은 입력 파일을 유지합니다.

버전 3.8에서 변경: 새로운 명령 중 인터페이스를 사용법과 함께 추가합니다. 기본적으로, CLI를 실행할 때, 기본 압축 수준은 6입니다.

명령 줄 옵션

file
*file*이 지정되지 않으면, *sys.stdin*에서 읽습니다.

--fast
가장 빠른 압축 방법(압축을 덜 함)을 나타냅니다.

--best
가장 느린 압축 방법(최상의 압축)을 나타냅니다.

-d, --decompress
주어진 파일의 압축을 풉니다.

-h, --help
도움말 메시지를 표시합니다.

13.5 bz2 — bzip2 압축 지원

소스 코드: Lib/bz2.py

이 모듈은 bzip2 압축 알고리즘을 사용하여 데이터 압축과 압축 해제를 위한 포괄적인 인터페이스를 제공합니다.

bz2 모듈에는 다음이 포함됩니다:

- 압축된 파일을 읽고 쓰기 위한 *open()* 함수와 *BZ2File* 클래스.
- 증분 압축(해제)을 위한 *BZ2Compressor*와 *BZ2Decompressor* 클래스.
- 일괄 압축(해제)을 위한 *compress()*와 *decompress()* 함수.

13.5.1 파일 압축(해제)

bz2.**open**(*filename, mode='rb', compresslevel=9, encoding=None, errors=None, newline=None*)

바이너리나 텍스트 모드로 bzip2 압축된 파일을 열고, 파일 객체를 반환합니다.

*BZ2File*의 생성자와 마찬가지로, *filename* 인자는 실제 파일명(*str*이나 *bytes* 객체)이거나, 읽거나 쓸 기존 파일 객체가 될 수 있습니다.

mode 인자는 바이너리 모드의 경우 `'r'`, `'rb'`, `'w'`, `'wb'`, `'x'`, `'xb'`, `'a'` 또는 `'ab'`, 또는 텍스트 모드의 경우 `'rt'`, `'wt'`, `'xt'` 또는 `'at'` 중 하나일 수 있습니다. 기본값은 `'rb'`입니다.

compresslevel 인자는 *BZ2File* 생성자와 마찬가지로 1에서 9 사이의 정수입니다.

바이너리 모드의 경우, 이 함수는 *BZ2File* 생성자 `BZ2File(filename, mode, compresslevel=compresslevel)`와 동등합니다. 이 경우, *encoding*, *errors* 및 *newline* 인자를 제공하면 안 됩니다.

텍스트 모드의 경우, *BZ2File* 객체가 만들어지고, 지정된 인코딩, 에러 처리 동작 및 줄 종료를 갖는 *io.TextIOWrapper* 인스턴스로 감싸집니다.

Added in version 3.3.

버전 3.4에서 변경: `'x'` (배타적 생성) 모드가 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

class bz2.**BZ2File**(*filename, mode='r', *, compresslevel=9*)

바이너리 모드로 bzip2 압축된 파일을 엽니다.

*filename*이 *str*이나 *bytes* 객체면, 명명된 파일을 직접 엽니다. 그렇지 않으면, *filename*은 파일 객체여야 하며, 압축된 데이터를 읽거나 쓰는 데 사용됩니다.

mode 인자는 읽기를 위한 `'r'` (기본값), 덮어쓰기를 위한 `'w'`, 배타적 생성을 위한 `'x'` 또는 덧붙이기를 위한 `'a'` 중 하나일 수 있습니다. 이들은 각각 `'rb'`, `'wb'`, `'xb'` 및 `'ab'`로 주어지는 것과 동등합니다.

*filename*이 (실제 파일 이름 대신) 파일 객체면, `'w'` 모드는 파일을 자르지 않으며, 대신 `'a'`와 동등합니다.

*mode*가 `'w'`나 `'a'`이면, *compresslevel*은 압축 수준을 지정하는 `1`과 `9` 사이의 정수일 수 있습니다: `1`은 압축률이 가장 낮고, `9`(기본값)는 압축률이 가장 높습니다.

*mode*가 `'r'`이면, 입력 파일은 여러 개의 압축된 스트림을 이어 붙인 것일 수 있습니다.

*BZ2File*은 *detach()*와 *truncate()*를 제외하고, *io.BufferedIOBase*가 지정하는 모든 멤버를 제공합니다. 이터레이션과 `with` 문이 지원됩니다.

*BZ2File*는 다음 메서드와 어트리뷰트도 제공합니다:

peek([*n*])

파일 위치를 전진시키지 않고 버퍼링 된 데이터를 반환합니다. (EOF에 있지 않은 한) 적어도 1 바이트의 데이터가 반환됩니다. 반환되는 정확한 바이트 수는 지정되지 않습니다.

> **참고**
>
> *peek()*를 호출할 때 *BZ2File*의 파일 위치가 변경되지는 않지만, 하부 파일 객체의 위치는 변경될 수 있습니다 (예를 들어, *BZ2File*이 *filename*에 파일 객체를 전달하여 생성된 경우).

Added in version 3.3.

fileno()

Return the file descriptor for the underlying file.

Added in version 3.3.

readable()

Return whether the file was opened for reading.

Added in version 3.3.

seekable()

Return whether the file supports seeking.

Added in version 3.3.

writable()

Return whether the file was opened for writing.

Added in version 3.3.

read1(*size=-1*)

Read up to *size* uncompressed bytes, while trying to avoid making multiple reads from the underlying stream. Reads up to a buffer' s worth of data if size is negative.

Returns `b''` if the file is at EOF.

Added in version 3.3.

readinto(*b*)

Read bytes into *b*.

Returns the number of bytes read (0 for EOF).

Added in version 3.3.

mode

`'rb'` for reading and `'wb'` for writing.

Added in version 3.13.

name

The bzip2 file name. Equivalent to the `name` attribute of the underlying *file object*.

Added in version 3.13.

버전 3.1에서 변경: `with` 문에 대한 지원이 추가되었습니다.

버전 3.3에서 변경: 실제 파일명 대신 파일 객체 인 *filename*에 대한 지원이 추가되었습니다.

다중 스트림 파일 읽기 지원과 함께, `'a'` (덧붙이기) 모드가 추가되었습니다.

버전 3.4에서 변경: `'x'` (배타적 생성) 모드가 추가되었습니다.

버전 3.5에서 변경: `read()` 메서드는 이제 `None` 인자를 받아들입니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.9에서 변경: *buffering* 매개 변수가 제거되었습니다. 파이썬 3.0부터 무시되고 폐지되었습니다. 파일을 여는 방법을 제어하려면 열린 파일 객체를 전달하십시오.

compresslevel 매개 변수가 키워드 전용이 되었습니다.

버전 3.10에서 변경: This class is thread unsafe in the face of multiple simultaneous readers or writers, just like its equivalent classes in `gzip` and `lzma` have always been.

13.5.2 증분 압축(해제)

class bz2.**BZ2Compressor**(*compresslevel=9*)

새로운 압축기 객체를 만듭니다. 이 객체는 증분 적으로(incrementally) 데이터를 압축하는 데 사용될 수 있습니다. 일괄(one-shot) 압축에는, 대신 `compress()` 함수를 사용하십시오.

주어진다면, *compresslevel*은 `1`과 `9` 사이의 정수여야 합니다. 기본값은 `9`입니다.

compress(*data*)

압축기 객체에 데이터를 제공합니다. 가능하면 압축된 데이터 청크를 반환하고, 그렇지 않으면 빈 바이트열을 반환합니다.

압축기에 데이터 제공이 끝나면, `flush()` 메서드를 호출하여 압축 공정을 마무리하십시오.

flush()

압축 공정을 마칩니다. 내부 버퍼에 남아있는 압축된 데이터를 반환합니다.

이 메서드가 호출된 후에는, 압축기 객체를 사용할 수 없습니다.

class bz2.**BZ2Decompressor**

새로운 압축 해제기 객체를 만듭니다. 이 객체는 데이터를 증분 적으로 압축 해제하는 데 사용될 수 있습니다. 일괄 압축에는, 대신 `decompress()` 함수를 사용하십시오.

> **참고**
>
> 이 클래스는 `decompress()`와 `BZ2File`과 달리 다중 압축 스트림을 포함하는 입력을 투명하게 처리하지 않습니다. `BZ2Decompressor`로 다중 스트림 입력의 압축을 풀어야 한다면, 각 스트림마다 새 압축 해제기를 사용해야 합니다.

decompress(*data*, *max_length=-1*)

data(바이트열류 객체)의 압축을 풀고, 압축되지 않은 데이터를 바이트열로 반환합니다. 일부 *data*는 나중에 `decompress()`를 호출할 때 사용할 수 있도록 내부에 버퍼링 됩니다. 반환된 데이터는 `decompress()`에 대한 이전 호출의 출력에 이어붙여야 합니다.

*max_length*가 음수가 아니면, 최대 *max_length* 바이트의 압축 해제된 데이터를 반환합니다. 이 제한에 도달했고, 추가 출력을 생성할 수 없으면, `needs_input` 어트리뷰트는 `False`로 설정됩니다. 이때, `decompress()`에 대한 다음 호출은 출력을 더 얻기 위해 *data*를 `b''`로 제공할 수 있습니다.

입력 데이터가 모두 압축 해제되고 반환되었으면 (*max_length* 바이트 미만이거나 *max_length*가 음수라서), `needs_input` 어트리뷰트가 `True`로 설정됩니다.

스트림의 끝에 도달한 이후에 데이터의 압축을 풀려고 하면, `EOFError`가 발생합니다. 스트림의 끝 이후에 발견된 모든 데이터는 무시되고, `unused_data` 어트리뷰트에 저장됩니다.

버전 3.5에서 변경: *max_length* 매개 변수가 추가되었습니다.

eof

스트림의 끝(end-of-stream) 마커에 도달했으면 `True`.

Added in version 3.3.

unused_data

압축된 스트림의 끝 이후에 발견된 데이터.

스트림의 끝에 도달하기 전에 이 어트리뷰트를 액세스하면, 값은 `b''`가 됩니다.

needs_input

`decompress()` 메서드가 새로운 압축된 입력을 요구하기 전에 압축 해제된 데이터를 더 제공할 수 있으면 `False`.

Added in version 3.5.

13.5.3 일괄 압축(해제)

bz2.**compress**(*data*, *compresslevel=9*)

바이트열류 객체, *data*를 압축합니다.

주어진다면, *compresslevel*은 `1`과 `9` 사이의 정수여야 합니다. 기본값은 `9`입니다.

증분 압축에는, 대신 `BZ2Compressor`를 사용하십시오.

bz2.**decompress**(*data*)

비이트열류 객체, *data*를 압축 해제합니다.

*data*가 다중 압축 스트림을 이어붙인 것이면, 모든 스트림의 압축을 풉니다.

증분 압축 해제에는, 대신 *BZ2Decompressor*를 사용하십시오.

버전 3.3에서 변경: 다중 스트림 입력에 대한 지원이 추가되었습니다.

13.5.4 사용 예

다음은 *bz2* 모듈의 일반적인 사용법에 대한 몇 가지 예입니다.

왕복 압축을 시연하기 위해 *compress()*와 *decompress()* 사용하기:

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> c = bz2.compress(data)
>>> len(data) / len(c)  # Data compression ratio
1.513595166163142
>>> d = bz2.decompress(c)
>>> data == d  # Check equality to original object after round-trip
True
```

증분 압축을 위한 *BZ2Compressor* 사용하기:

```
>>> import bz2
>>> def gen_data(chunks=10, chunksize=1000):
...     """Yield incremental blocks of chunksize bytes."""
...     for _ in range(chunks):
...         yield b"z" * chunksize
...
>>> comp = bz2.BZ2Compressor()
>>> out = b""
>>> for chunk in gen_data():
...     # Provide data to the compressor object
...     out = out + comp.compress(chunk)
...
>>> # Finish the compression process.  Call this once you have
>>> # finished providing data to the compressor.
>>> out = out + comp.flush()
```

위의 예제는 매우 "무작위적이지 않은" 데이터 스트림(`b"z"` 청크의 스트림)을 사용합니다. 무작위 데이터는 압축이 잘되지 않지만, 반복적인 데이터는 일반적으로 높은 압축률을 산출합니다.

바이너리 모드로 bzip2 압축된 파일을 쓰고 읽기:

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> with bz2.open("myfile.bz2", "wb") as f:
...     # Write compressed data to file
...     unused = f.write(data)
...
>>> with bz2.open("myfile.bz2", "rb") as f:
...     # Decompress data from file
...     content = f.read()
...
>>> content == data  # Check equality to original object after round-trip
True
```

13.6 lzma — LZMA 알고리즘을 사용한 압축

Added in version 3.3.

소스 코드: Lib/lzma.py

이 모듈은 LZMA 압축 알고리즘을 사용하여 데이터를 압축 및 압축 해제하기 위한 클래스와 편의 함수를 제공합니다. 또한 **xz** 유틸리티에서 사용되는 `.xz`와 레거시 `.lzma` 파일 형식뿐만 아니라 원시 압축 스트림을 지원하는 파일 인터페이스도 포함되어 있습니다.

이 모듈에서 제공하는 인터페이스는 *bz2* 모듈의 인터페이스와 매우 유사합니다. *LZMAFile*과 *bz2.BZ2File*은 스레드 안전하지 않아서, 여러 스레드에서 단일 *LZMAFile* 인스턴스를 사용해야 하면 록으로 보호해야 합니다.

exception lzma.**LZMAError**

이 예외는 압축이나 압축 해제 중, 또는 압축기/압축 해제기 상태를 초기화하는 동안 에러가 발생할 때 발생합니다.

13.6.1 압축 파일 읽기와 쓰기

lzma.**open**(*filename, mode='rb', *, format=None, check=-1, preset=None, filters=None, encoding=None, errors=None, newline=None*)

바이너리나 텍스트 모드에서 LZMA 압축 파일을 열고, 파일 객체를 반환합니다.

filename 인자는 실제 파일 이름(*str*, *bytes* 또는 경로류 객체로 제공됩니다)일 수 있고, 이때는 명명된 파일이 열립니다. 또는 읽거나 쓸 기존 파일 객체일 수 있습니다.

mode 인자는 바이너리 모드의 경우 `"r"`, `"rb"`, `"w"`, `"wb"`, `"x"`, `"xb"`, `"a"` 또는 `"ab"`이거나, 텍스트 모드의 경우 `"rt"`, `"wt"`, `"xt"` 또는 `"at"` 일 수 있습니다. 기본값은 `"rb"`입니다.

파일을 읽기 위해 열 때, *format*과 *filters* 인자는 *LZMADecompressor*와 같은 의미입니다. 이 경우, *check*과 *preset* 인자를 사용하지 않아야 합니다.

파일을 쓰기 위해 열 때, *format*, *check*, *preset* 및 *filters* 인자는 *LZMACompressor*와 같은 의미입니다.

바이너리 모드의 경우, 이 함수는 *LZMAFile* 생성자와 동등합니다: `LZMAFile(filename, mode, ...)`. 이 경우, *encoding*, *errors* 및 *newline* 인자는 제공하지 않아야 합니다.

텍스트 모드의 경우, *LZMAFile* 객체가 만들어지고, 지정된 인코딩, 에러 처리 동작 및 줄 종료로 *io.TextIOWrapper* 인스턴스로 감쌉니다.

버전 3.4에서 변경: `"x"`, `"xb"` 및 `"xt"` 모드에 대한 지원이 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 허용합니다.

class lzma.**LZMAFile**(*filename=None, mode='r', *, format=None, check=-1, preset=None, filters=None*)

바이너리 모드로 LZMA 압축 파일을 엽니다.

*LZMAFile*은 이미 열려있는 파일 객체를 래핑하거나, 명명된 파일에 직접 작용할 수 있습니다. *filename* 인자는 래핑할 파일 객체나 열 파일의 이름(*str*, *bytes* 또는 경로류 객체로)을 지정합니다. 기존 파일 객체를 래핑할 때, 래핑 된 파일은 *LZMAFile*이 닫힐 때 닫히지 않습니다.

mode 인자는 읽기 위한 "r" (기본값), 덮어쓰기 위한 "w", 배타적 생성을 위한 "x" 또는 덧붙이기를 위한 "a" 일 수 있습니다. 이들은 각각 "rb", "wb", "xb" 및 "ab"로 동등하게 제공될 수 있습니다.

*filename*이 (실제 파일 이름이 아닌) 파일 객체이면, "w" 모드는 파일을 자르지 않으며, 대신 "a"와 동등합니다.

읽기 위해 파일을 열 때, 입력 파일은 여러 개의 개별 압축 스트림을 연결한 것일 수 있습니다. 이들은 단일 논리 스트림으로 투명하게 디코딩됩니다.

파일을 읽기 위해 열 때, *format*과 *filters* 인자는 *LZMADecompressor*와 같은 의미입니다. 이 경우, *check*과 *preset* 인자를 사용하지 않아야 합니다.

파일을 쓰기 위해 열 때, *format*, *check*, *preset* 및 *filters* 인자는 *LZMACompressor*와 같은 의미입니다.

*LZMAFile*은 *detach()*와 *truncate()*를 제외하고, *io.BufferedIOBase* 가 지정하는 모든 멤버를 지원합니다. 이터레이션과 with 문이 지원됩니다.

다음과 같은 메서드와 어트리뷰트도 제공됩니다:

peek(*size=-1*)

파일 위치를 진행하지 않고 버퍼링 된 데이터를 반환합니다. EOF에 도달하지 않았으면, 최소 1 바이트의 데이터가 반환됩니다. 반환되는 정확한 바이트 수는 지정되지 않습니다 (*size* 인자는 무시됩니다).

> **참고**
>
> *peek()*를 호출해도 *LZMAFile*의 파일 위치는 변경되지 않지만, 하부 파일 객체의 위치는 변경될 수 있습니다 (예를 들어 *LZMAFile*이 *filename*으로 파일 객체를 전달하여 생성되었을 때).

mode

'rb' for reading and 'wb' for writing.

Added in version 3.13.

name

The lzma file name. Equivalent to the *name* attribute of the underlying *file object*.

Added in version 3.13.

버전 3.4에서 변경: "x"와 "xb" 모드에 대한 지원이 추가되었습니다.

버전 3.5에서 변경: *read()* 메서드는 이제 None 인자를 허용합니다.

버전 3.6에서 변경: 경로류 객체를 허용합니다.

13.6.2 메모리에서의 데이터 압축과 압축 해제

class lzma.**LZMACompressor**(*format=FORMAT_XZ, check=-1, preset=None, filters=None*)

데이터를 증분 압축하는 데 사용할 수 있는 압축기 객체를 만듭니다.

단일 데이터 청크를 압축하는 더 편리한 방법은, *compress()*를 참조하십시오.

format 인자는 사용해야 할 컨테이너 형식을 지정합니다. 가능한 값은 다음과 같습니다:

- **FORMAT_XZ: .xz 컨테이너 형식.**
  이것이 기본 형식입니다.

- **FORMAT_ALONE: 레거시 .lzma 컨테이너 형식.**
  이 형식은 .xz보다 제한적입니다 – 무결성 검사나 다중 필터를 지원하지 않습니다.
- **FORMAT_RAW: 컨테이너 형식을 사용하지 않는 원시 데이터 스트림.**
  이 형식 지정자는 무결성 검사를 지원하지 않으며, 항상 사용자 지정 필터 체인(압축과 압축 해제 모두를 위한)을 지정해야 합니다. 또한, 이 방식으로 압축된 데이터는 FORMAT_AUTO를 사용하여 압축 해제할 수 없습니다 (*LZMADecompressor*를 참조하십시오).

check 인자는 압축된 데이터에 포함할 무결성 검사 유형을 지정합니다. 이 검사는 압축을 풀 때 데이터가 손상되지 않았는지 확인하는 데 사용됩니다. 가능한 값은 다음과 같습니다:

- CHECK_NONE: 무결성 검사가 없습니다. 이것은 FORMAT_ALONE과 FORMAT_RAW에 대한 기본값(그리고 유일하게 허용된 값)입니다.
- CHECK_CRC32: 32비트 순환 중복 검사(Cyclic Redundancy Check).
- CHECK_CRC64: 64비트 순환 중복 검사(Cyclic Redundancy Check). 이것이 FORMAT_XZ의 기본값입니다.
- CHECK_SHA256: 256비트 보안 해시 알고리즘(Secure Hash Algorithm).

지정된 검사가 지원되지 않으면, *LZMAError*가 발생합니다.

압축 설정은 사전 설정 압축 수준(*preset* 인자 사용), 또는 사용자 정의 필터 체인(*filters* 인자 사용)으로 지정할 수 있습니다.

preset 인자(제공된 경우)는 0rhk 9 사이의 (경계 포함) 정수여야 하며, 선택적으로 상수 PRESET_EXTREME과 OR 할 수 있습니다. *preset**과 **filters*가 모두 제공되지 않으면, 기본 동작은 PRESET_DEFAULT(사전 설정 수준 6)를 사용하는 것입니다. 사전 설정이 높을수록 출력은 작아 지지만, 압축 과정은 느려집니다.

> **참고**
>
> CPU를 많이 사용하는 것 외에도, 사전 설정이 높은 압축은 훨씬 더 많은 메모리를 요구합니다(그리고 압축을 풀기 위해 더 많은 메모리를 요구하는 출력을 생성합니다). 예를 들어 사전 설정 9를 사용하면, *LZMACompressor* 객체의 오버헤드가 800 MiB에 이를 수 있습니다. 이런 이유로, 일반적으로 기본 사전 설정을 사용하는 것이 가장 좋습니다.

filters 인자(제공된 경우)는 필터 체인 지정자여야 합니다. 자세한 내용은 사용자 정의 필터 체인 지정을 참조하십시오.

compress(*data*)

data(*bytes* 객체)를 압축하여, 적어도 입력의 일부에 대한 압축 데이터가 포함된 *bytes* 객체를 반환합니다. *data*의 일부는 나중에 *compress()*와 *flush()*에 대한 호출에 사용하기 위해 내부적으로 버퍼링 될 수 있습니다. 반환된 데이터는 *compress()*에 대한 이전 호출의 출력에 이어 붙여야 합니다.

flush()

압축 과정을 마치고, 압축기의 내부 버퍼에 저장된 모든 데이터가 포함된 *bytes* 객체를 반환합니다.

이 메서드를 호출한 후에는 압축기를 사용할 수 없습니다.

class lzma.**LZMADecompressor**(*format=FORMAT_AUTO, memlimit=None, filters=None*)

데이터를 점진적으로 압축 해제하는 데 사용할 수 있는 압축 해제기 객체를 만듭니다.

전체 압축 스트림을 한 번에 압축 해제하는 더 편리한 방법은 *decompress()*를 참조하십시오.

format 인자는 사용해야 하는 컨테이너 형식을 지정합니다. 기본값은 FORMAT_AUTO이며, .xz와 .lzma 파일을 모두 압축 해제할 수 있습니다. 다른 가능한 값은 FORMAT_XZ, FORMAT_ALONE 및 FORMAT_RAW입니다.

memlimit 인자는 압축 해제기가 사용할 수 있는 메모리양의 한계(바이트)를 지정합니다. 이 인자를 사용할 때, 주어진 메모리 한계 내에서 입력을 압축 해제할 수 없으면 *LZMAError*로 압축 해제에 실패합니다.

filters 인자는 압축 해제 중인 스트림을 만드는 데 사용된 필터 체인을 지정합니다. *format*이 `FORMAT_RAW`이면 이 인자가 필요하지만, 다른 형식에는 사용하지 않아야 합니다. 필터 체인에 대한 자세한 내용은 사용자 정의 필터 체인 지정을 참조하십시오.

> **참고**
>
> 이 클래스는 `decompress()`와 `LZMAFile`과 달리, 여러 압축 스트림을 포함하는 입력을 투명하게 처리하지 않습니다. `LZMADecompressor`로 다중 스트림 입력을 압축 해제하려면 각 스트림에 대해 새로운 압축 해제기를 만들어야 합니다.

decompress(*data*, *max_length=-1*)

data(바이트열류 객체)를 압축 해제하고, 압축되지 않은 데이터를 바이트열로 반환합니다. *data*의 일부는 나중에 `decompress()`를 호출할 때 사용하기 위해 내부적으로 버퍼링 될 수 있습니다. 반환된 데이터는 `decompress()`에 대한 이전 호출의 출력에 이어 붙여야 합니다.

*max_length*가 음수가 아니면, 최대 *max_length* 바이트의 압축 해제된 데이터를 반환합니다. 이 한계에 도달하고 추가 출력을 생성할 수 있으면, `needs_input` 어트리뷰트가 `False`로 설정됩니다. 이 경우, 다음 `decompress()` 호출은 *data*를 `b''`로 제공하여 더 많은 출력을 얻을 수 있습니다.

모든 입력 데이터가 압축 해제되어 반환되면 (이것이 *max_length* 바이트 미만이거나 *max_length*가 음수이기 때문에), `needs_input` 어트리뷰트는 `True`로 설정됩니다.

스트림 끝에 도달한 후 데이터 압축 해제를 시도하면 `EOFError`가 발생합니다. 스트림 끝 이후에 발견되는 모든 데이터는 무시되고 `unused_data` 어트리뷰트에 저장됩니다.

버전 3.5에서 변경: *max_length* 매개 변수를 추가했습니다.

check

입력 스트림이 사용하는 무결성 검사의 ID. 사용되는 무결성 검사를 결정하기 위해 충분한 입력이 디코딩될 때까지 `CHECK_UNKNOWN`일 수 있습니다.

eof

스트림 끝 마커에 도달하면 `True`.

unused_data

압축된 스트림이 끝난 후 발견된 데이터.

스트림의 끝에 도달하기 전에, 이것은 `b""`입니다.

needs_input

`decompress()` 메서드가 새로운 압축 입력을 요구하기 전에 더 많은 압축 해제된 데이터를 제공할 수 있으면 `False`.

Added in version 3.5.

`lzma`.**compress**(*data*, *format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

data(`bytes` 객체)를 압축하여, 압축된 데이터를 `bytes` 객체로 반환합니다.

format, *check*, *preset* 및 *filters* 인자에 대한 설명은 위의 `LZMACompressor`를 참조하십시오.

`lzma`.**decompress**(*data*, *format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

data(`bytes` 객체)를 압축 해제하여, 압축되지 않은 데이터를 `bytes` 객체로 반환합니다.

*data*가 여러 개의 개별 압축 스트림의 연결이면, 이러한 스트림들을 모두 압축 해제하고 결과를 이어 붙여 반환합니다.

format, *memlimit* 및 *filters* 인자에 대한 설명은 위의 `LZMADecompressor`를 참조하십시오.

13.6.3 기타

`lzma.is_check_supported(check)`

주어진 무결성 검사가 이 시스템에서 지원되면 `True`를 반환합니다.

`CHECK_NONE`과 `CHECK_CRC32`는 항상 지원됩니다. 제한된 기능 집합으로 컴파일된 **liblzma** 버전을 사용하는 경우 `CHECK_CRC64`와 `CHECK_SHA256`을 사용하지 못할 수 있습니다.

13.6.4 사용자 정의 필터 체인 지정

필터 체인 지정자는 딕셔너리의 시퀀스로, 각 딕셔너리에는 단일 필터의 ID와 옵션이 포함됩니다. 각 딕셔너리는 키 `"id"`를 포함해야 하며, 필터 종속 옵션을 지정하기 위해 추가 키를 포함할 수 있습니다. 유효한 필터 ID는 다음과 같습니다:

- 압축 필터:
  - `FILTER_LZMA1` (`FORMAT_ALONE`과 함께 사용)
  - `FILTER_LZMA2` (`FORMAT_XZ` 및 `FORMAT_RAW`와 함께 사용)
- 델타 필터:
  - `FILTER_DELTA`
- Branch-Call-Jump (BCJ) 필터:
  - `FILTER_X86`
  - `FILTER_IA64`
  - `FILTER_ARM`
  - `FILTER_ARMTHUMB`
  - `FILTER_POWERPC`
  - `FILTER_SPARC`

필터 체인은 최대 4개의 필터로 구성될 수 있으며, 비워 둘 수 없습니다. 체인의 마지막 필터는 압축 필터여야 하고, 다른 필터는 델타나 BCJ 필터여야 합니다.

압축 필터는 다음 옵션을 지원합니다 (필터를 나타내는 딕셔너리에 추가 항목으로 지정됩니다):

- `preset`: 명시적으로 지정되지 않은 옵션의 기본값 소스로 사용할 압축 사전 설정.
- `dict_size`: 바이트로 표현한 딕셔너리 크기. 4 KiB와 1.5 GiB 사이여야 합니다 (경계 포함).
- `lc`: 리터럴 컨텍스트 비트 수.
- `lp`: 리터럴 위치 비트 수. 합계 `lc + lp`는 최대 4여야 합니다.
- `pb`: 위치 비트 수; 최대 4여야 합니다.
- `mode`: `MODE_FAST`나 `MODE_NORMAL`.
- `nice_len`: 매치에서 "좋은 길이"로 간주하는 것. 273 이하여야 합니다.
- `mf`: 사용할 매치 파인더 – `MF_HC3`, `MF_HC4`, `MF_BT2`, `MF_BT3` 또는 `MF_BT4`.
- `depth`: 매치 파인더가 사용하는 최대 검색 깊이. 0(기본값)은 다른 필터 옵션을 기반으로 자동 선택함을 의미합니다.

델타 필터는 바이트 간 차이를 저장하여, 특정 상황에서 압축기에 대해 더 반복적인 입력을 생성합니다. 한 가지 옵션을 지원합니다, `dist`. 이것은 빼야 할 바이트 간의 거리를 나타냅니다. 기본값은 1입니다. 즉, 인접 바이트 간 차이를 취합니다.

BCJ 필터는 기계 코드에 적용하려는 것입니다. 이들은 압축기가 이용할 수 있는 중복성을 높이기 위해 코드에서 상대 분기, 호출 및 점프를 절대 주소 지정을 사용하도록 변환합니다. 이 필터는 한 가지 옵션을 지원합니다, `start_offset`. 이것은 입력 데이터의 시작 부분으로 매핑되어야 하는 주소를 지정합니다. 기본값은 0입니다.

13.6.5 예

압축 파일 읽기:

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

압축 파일 만들기:

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

메모리에서 데이터 압축하기:

```
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

증분 압축:

```
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
out2 = lzc.compress(b"Another piece of data\n")
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
# 모든 부분 결과를 이어 붙입니다:
result = b"".join([out1, out2, out3, out4])
```

이미 열린 파일에 압축된 데이터 쓰기:

```
import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
        lzf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

사용자 정의 필터 체인을 사용하여 압축 파일 만들기:

```
import lzma
my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME},
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")
```

13.7 zipfile — ZIP 아카이브 작업

소스 코드: Lib/zipfile/

ZIP 파일 형식은 흔히 쓰이는 아카이브와 압축 표준입니다. 이 모듈은 ZIP 파일을 만들고, 읽고, 쓰고, 추가하고, 나열하는 도구를 제공합니다. 이 모듈의 고급 사용을 위해서는 PKZIP Application Note에 정의된 형식의 이해가 필요합니다.

이 모듈은 현재 다중 디스크 ZIP 파일을 처리하지 않습니다. ZIP64 확장을 사용하는 ZIP 파일(즉, 크기가 4GiB 이상인 ZIP 파일)을 처리할 수 있습니다. ZIP 아카이브에 있는 암호화된 파일의 암호 해독을 지원하지만, 현재 암호화된 파일을 만들 수는 없습니다. C가 아닌 네이티브 파이썬으로 구현되므로 해독 속도가 매우 느립니다.

이 모듈은 다음 항목을 정의합니다:

exception zipfile.**BadZipFile**

잘못된 ZIP 파일로 인해 발생하는 에러.

Added in version 3.2.

exception zipfile.**BadZipfile**

이전 파이썬 버전과의 호환성을 위한, `BadZipFile`의 별칭.

버전 3.2부터 폐지됨.

exception zipfile.**LargeZipFile**

ZIP 파일에 ZIP64 기능이 필요하지만 활성화되지 않았을 때 발생하는 에러.

class zipfile.**ZipFile**

ZIP 파일을 읽고 쓰는 클래스. 생성자 세부 사항은 *ZipFile 객체* 섹션을 참조하십시오.

class zipfile.**Path**

Class that implements a subset of the interface provided by `pathlib.Path`, including the full `importlib.resources.abc.Traversable` interface.

Added in version 3.8.

class zipfile.**PyZipFile**

파이썬 라이브러리를 포함하는 ZIP 아카이브를 만들기 위한 클래스.

class zipfile.**ZipInfo**(*filename='NoName', date_time=(1980, 1, 1, 0, 0, 0)*)

아카이브 멤버에 관한 정보를 나타내는 데 사용되는 클래스. 이 클래스의 인스턴스는 `ZipFile` 객체의 `getinfo()`와 `infolist()` 메서드에 의해 반환됩니다. `zipfile` 모듈의 대부분 사용자는 이것들을 만들 필요는 없고, 이 모듈에서 만든 것들을 사용하기만 합니다. *filename*은 아카이브 멤버의 전체 이름이어야 하고, *date_time*은 파일을 마지막으로 수정한 시간을 기술하는 6개의 필드를 포함하는 튜플이어야 합니다; 필드는 *ZipInfo 객체* 섹션에 설명되어 있습니다.

버전 3.13에서 변경: A public `compress_level` attribute has been added to expose the formerly protected `_compresslevel`. The older protected name continues to work as a property for backwards compatibility.

_for_archive(*archive*)

Resolve the date_time, compression attributes, and external attributes to suitable defaults as used by `ZipFile.writestr()`.

Returns self for chaining.

Added in version 3.14.

zipfile.**is_zipfile**(*filename*)

*filename*이 매직 번호에 기반하여 유효한 ZIP 파일이면 `True`를, 그렇지 않으면 `False`를 반환합니다. *filename*은 파일이거나 파일류 객체일 수도 있습니다.

버전 3.1에서 변경: 파일과 파일류 객체를 지원합니다.

zipfile.**ZIP_STORED**

압축되지 않은 아카이브 멤버를 위한 숫자 상수.

zipfile.**ZIP_DEFLATED**

일반적인 ZIP 압축 방법을 위한 숫자 상수. `zlib` 모듈이 필요합니다.

zipfile.**ZIP_BZIP2**

BZIP2 압축 방법을 위한 숫자 상수. `bz2` 모듈이 필요합니다.

Added in version 3.3.

zipfile.**ZIP_LZMA**

LZMA 압축 방법을 위한 숫자 상수. *lzma* 모듈이 필요합니다.

Added in version 3.3.

zipfile.**ZIP_ZSTANDARD**

The numeric constant for Zstandard compression. This requires the *compression.zstd* module.

> **참고**
>
> In APPNOTE 6.3.7, the method ID `20` was assigned to Zstandard compression. This was changed in APPNOTE 6.3.8 to method ID `93` to avoid conflicts, with method ID `20` being deprecated. For compatibility, the `zipfile` module reads both method IDs but will only write data with method ID `93`.

Added in version 3.14.

> **참고**
>
> The ZIP file format specification has included support for bzip2 compression since 2001, for LZMA compression since 2006, and Zstandard compression since 2020. However, some tools (including older Python releases) do not support these compression methods, and may either refuse to process the ZIP file altogether, or fail to extract individual files.

> **더 보기**
>
> **PKZIP Application Note**
> 사용된 형식과 알고리즘의 저자인 Phil Katz의 ZIP 파일 형식에 대한 설명서.
>
> **Info-ZIP 홈페이지**
> Info-ZIP 프로젝트의 ZIP 아카이브 프로그램과 개발 라이브러리에 관한 정보.

13.7.1 ZipFile 객체

class zipfile.**ZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *compresslevel=None*, **, strict_timestamps=True*, *metadata_encoding=None*)

ZIP 파일을 엽니다, 여기서 *file*은 파일에 대한 경로 (문자열), 파일류 객체 또는 경로류 객체일 수 있습니다.

mode 매개 변수는 기존 파일을 읽으려면 `'r'`, 새 파일을 자르고 쓰려면 `'w'`, 기존 파일에 추가하려면 `'a'`, 새 파일을 독점적으로 작성하고 쓰려면 `'x'` 이어야 합니다. *mode*가 `'x'`이고 *file*이 기존 파일을 참조하면, `FileExistsError` 가 발생합니다. *mode*가 `'a'`이고 *file*이 기존 ZIP 파일을 참조하면, 추가 파일이 이곳으로 추가됩니다. *file*이 ZIP 파일을 참조하지 않으면, 새 ZIP 아카이브를 파일에 덧붙입니다(append). 이는 ZIP 아카이브를 다른 파일(가령 `python.exe`)에 추가하기 위한 것입니다. *mode*가 `'a'`이고 파일이 아예 존재하지 않으면, 파일이 만들어집니다. *mode*가 `'r'`이나 `'a'`이면, 파일은 탐색 가능(seekable)해야 합니다.

compression is the ZIP compression method to use when writing the archive, and should be `ZIP_STORED`, `ZIP_DEFLATED`, `ZIP_BZIP2`, `ZIP_LZMA`, or `ZIP_ZSTANDARD`; unrecognized values will cause `NotImplementedError` to be raised. If `ZIP_DEFLATED`, `ZIP_BZIP2`, `ZIP_LZMA`, or `ZIP_ZSTANDARD` is specified but the corresponding module (`zlib`, `bz2`, `lzma`, or `compression.zstd`) is not available, `RuntimeError` is raised. The default is `ZIP_STORED`.

*allowZip64*가 `True`(기본값)이면 zipfile은 ZIP 파일이 4GiB보다 클 때 ZIP64 확장을 사용하는 ZIP 파일을 만듭니다. `false`이면 ZIP 파일에 ZIP64 확장자가 필요할 때 *zipfile*은 예외를 발생시킵니다.

The *compresslevel* parameter controls the compression level to use when writing files to the archive. When using `ZIP_STORED` or `ZIP_LZMA` it has no effect. When using `ZIP_DEFLATED` integers `0` through `9` are accepted

(see *zlib* for more information). When using *ZIP_BZIP2* integers 1 through 9 are accepted (see *bz2* for more information). When using *ZIP_ZSTANDARD* integers -131072 through 22 are commonly accepted (see *CompressionParameter.compression_level* for more on retrieving valid values and their meaning).

strict_timestamps 인자를 False로 설정하면, 1980-01-01 이전의 zip 파일을 허용하는 대신 타임 스탬프를 1980-01-01로 설정합니다. 2107-12-31 이후의 파일에 대해서도 비슷한 동작이 발생하며, 타임 스탬프는 역시 한곗값으로 설정됩니다.

When mode is 'r', *metadata_encoding* may be set to the name of a codec, which will be used to decode metadata such as the names of members and ZIP comments.

파일이 'w', 'x' 또는 'a' 모드로 만들어졌고 아카이브에 아무런 파일도 추가하지 않고 닫히면, 비어있는 아카이브에 적합한 ZIP 구조가 파일에 기록됩니다.

ZipFile은 또한 컨텍스트 관리자이므로 with 문을 지원합니다. 이 예에서, *myzip*은 with 문 스위트가 완료된 후에 닫힙니다 – 예외가 발생할 때조차 그렇습니다:

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

> **참고**
>
> *metadata_encoding* is an instance-wide setting for the ZipFile. It is not currently possible to set this on a per-member basis.
>
> This attribute is a workaround for legacy implementations which produce archives with names in the current locale encoding or code page (mostly on Windows). According to the .ZIP standard, the encoding of metadata may be specified to be either IBM code page (default) or UTF-8 by a flag in the archive header. That flag takes precedence over *metadata_encoding*, which is a Python-specific extension.

버전 3.2에서 변경: *ZipFile*을 컨텍스트 관리자로 사용하는 기능이 추가되었습니다.

버전 3.3에서 변경: *bzip2*와 *lzma* 압축에 대한 지원이 추가되었습니다.

버전 3.4에서 변경: ZIP64 확장은 기본적으로 활성화됩니다.

버전 3.5에서 변경: 탐색할 수 없는(unseekable) 스트림으로의 쓰기 지원을 추가했습니다. 'x' 모드에 대한 지원이 추가되었습니다.

버전 3.6에서 변경: 이전에는, 인식할 수 없는 compression 값에 대해 평범한 *RuntimeError*가 발생했습니다.

버전 3.6.2에서 변경: *file* 매개 변수는 경로류 객체를 받아들입니다.

버전 3.7에서 변경: *compresslevel* 매개 변수를 추가했습니다.

버전 3.8에서 변경: *strict_timestamps* 키워드 전용 매개 변수

버전 3.11에서 변경: Added support for specifying member name encoding for reading metadata in the zipfile's directory and file headers.

ZipFile.**close**()

아카이브 파일을 닫습니다. 프로그램을 종료하기 전에 *close()*를 호출해야 합니다. 그렇지 않으면 필수 레코드가 기록되지 않습니다.

ZipFile.**getinfo**(*name*)

아카이브 멤버 *name*에 관한 정보가 있는 *ZipInfo* 객체를 반환합니다. 현재 아카이브에 포함되지 않은 이름에 대해 *getinfo()*를 호출하면 *KeyError*가 발생합니다.

ZipFile.**infolist**()

아카이브의 각 멤버에 대한 *ZipInfo* 객체를 포함하는 리스트를 반환합니다. 기존 아카이브가 열린 경우 객체는 디스크의 실제 ZIP 파일에 있는 항목과 순서가 같습니다.

ZipFile.**namelist**()

아카이브 멤버의 리스트를 이름으로 반환합니다.

ZipFile.**open**(*name, mode='r', pwd=None, *, force_zip64=False*)

아카이브 멤버를 바이너리 파일류 객체로 액세스합니다. *name*은 아카이브 내의 파일 이름이거나 *ZipInfo* 객체일 수 있습니다. 포함될 때 *mode* 매개 변수는 'r'(기본값)이거나 'w' 이어야 합니다. *pwd*는 암호화된 ZIP 파일을 해독하는 데 사용되는 *bytes* 객체인 비밀번호입니다.

*open()*은 컨텍스트 관리자이기도 하므로 with 문을 지원합니다:

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

mode 'r'에서 파일류 객체(ZipExtFile)는 읽기 전용이며 다음 메서드를 제공합니다: *read()*, *readline()*, *readlines()*, *seek()*, *tell()*, *__iter__()*, *__next__()*. 이러한 객체는 ZipFile과 독립적으로 작동할 수 있습니다.

mode='w'에서, *write()* 메서드를 지원하는 쓰기 가능한 파일 핸들이 반환됩니다. 쓰기 가능한 파일 핸들이 열려있는 동안, ZIP 파일에서 다른 파일을 읽거나 쓰려고 시도하면 *ValueError*가 발생합니다.

In both cases the file-like object has also attributes name, which is equivalent to the name of a file within the archive, and mode, which is 'rb' or 'wb' depending on the input mode.

파일을 기록할 때, 파일 크기를 미리 알 수 없지만 2GiB를 초과할 수 있으면, 헤더 형식이 큰 파일을 지원할 수 있도록 force_zip64=True를 전달하십시오. 파일 크기가 미리 알려졌으면, *file_size*가 설정된 *ZipInfo* 객체를 구성하고, 이를 *name* 매개 변수로 사용하십시오.

> **참고**
>
> *open()*, *read()* 및 *extract()* 메서드는 파일명이나 *ZipInfo* 객체를 취할 수 있습니다. 중복 이름을 가진 멤버가 포함된 ZIP 파일을 읽으려고 할 때 이 점에 감사할 것입니다.

버전 3.6에서 변경: mode='U' 지원이 제거되었습니다. 유니버설 줄 넘김 모드로 압축된 텍스트 파일을 읽으려면 *io.TextIOWrapper*를 사용하십시오.

버전 3.6에서 변경: 이제 *ZipFile.open()*은 이제 mode='w' 옵션으로 파일을 아카이브에 기록하는 데 사용될 수 있습니다.

버전 3.6에서 변경: 닫힌 ZipFile에 *open()*을 호출하면 *ValueError*가 발생합니다. 이전에는, *RuntimeError*가 발생했습니다.

버전 3.13에서 변경: Added attributes name and mode for the writeable file-like object. The value of the mode attribute for the readable file-like object was changed from 'r' to 'rb'.

ZipFile.**extract**(*member, path=None, pwd=None*)

아카이브에서 현재 작업 디렉터리로 멤버를 추출합니다. *member*는 전체 이름이거나 *ZipInfo* 객체여야 합니다. 파일 정보는 최대한 정확하게 추출됩니다. *path*는 추출할 다른 디렉터리를 지정합니다. *member*는 파일 이름이나 *ZipInfo* 객체일 수 있습니다. *pwd*는 암호화된 파일에 사용되는 *bytes* 객체인 비밀번호입니다.

만들어진 정규화된 경로(디렉터리나 새 파일)를 반환합니다.

> **참고**
>
> 멤버 파일명이 절대 경로이면, 드라이브/UNC 공유 지점(sharepoint)과 선행 (역) 슬래시가 제거됩니다, 예를 들어: ///foo/bar는 유닉스에서 foo/bar가 되고, 윈도우에서 C:\foo\bar는 foo\bar가 됩니다. 그리고 멤버 파일명의 모든 ".." 구성 요소가 제거됩니다, 예를 들어: ../../foo../../ba..r은 foo../ba..r이 됩니다. 윈도우에서 잘못된 문자(:, <, >, |, ", ? 및 *)는 밑줄(_)로 대체됩니다.

버전 3.6에서 변경: 닫힌 ZipFile에서 `extract()`를 호출하면 `ValueError`가 발생합니다. 이전에는 `RuntimeError`가 발생했습니다.

버전 3.6.2에서 변경: *path* 매개 변수는 경로류 객체를 받아들입니다.

`ZipFile.`**`extractall`**(*path=None, members=None, pwd=None*)

아카이브에서 현재 작업 디렉터리로 모든 멤버를 추출합니다. *path*는 추출할 다른 디렉터리를 지정합니다. *members*는 선택적이며 `namelist()`가 반환한 리스트의 부분 집합이어야 합니다. *pwd*는 암호화된 파일에 사용되는 `bytes` 객체인 비밀번호입니다.

> **경고**
>
> 사전 검사 없이 신뢰할 수 없는 출처의 아카이브를 추출하지 마십시오. 파일이 *path* 밖에 만들어질 수 있습니다, 예를 들어 `"/"`로 시작하는 절대 파일명을 가진 멤버나 두 점 `".."`을 포함하는 파일명. 이 모듈은 이를 방지하려고 시도합니다. `extract()` 참고를 참조하십시오.

버전 3.6에서 변경: 닫힌 ZipFile에서 `extractall()`을 호출하면 `ValueError`가 발생합니다. 이전에는 `RuntimeError`가 발생했습니다.

버전 3.6.2에서 변경: *path* 매개 변수는 경로류 객체를 받아들입니다.

`ZipFile.`**`printdir`**()

아카이브의 목차를 `sys.stdout`으로 인쇄합니다.

`ZipFile.`**`setpassword`**(*pwd*)

암호화된 파일을 추출하기 위해 *pwd*(`bytes` 객체)를 기본 비밀번호로 설정합니다.

`ZipFile.`**`read`**(*name, pwd=None*)

Return the bytes of the file *name* in the archive. *name* is the name of the file in the archive, or a `ZipInfo` object. The archive must be open for read or append. *pwd* is the password used for encrypted files as a `bytes` object and, if specified, overrides the default password set with `setpassword()`. Calling `read()` on a ZipFile that uses a compression method other than `ZIP_STORED`, `ZIP_DEFLATED`, `ZIP_BZIP2`, `ZIP_LZMA`, or `ZIP_ZSTANDARD` will raise a `NotImplementedError`. An error will also be raised if the corresponding compression module is not available.

버전 3.6에서 변경: 닫힌 ZipFile에서 `read()`를 호출하면 `ValueError`가 발생합니다. 이전에는 `RuntimeError`가 발생했습니다.

`ZipFile.`**`testzip`**()

아카이브의 모든 파일을 읽고 CRC와 파일 헤더를 확인합니다. 첫 번째 불량 파일의 이름을 반환하거나, `None`을 반환합니다.

버전 3.6에서 변경: 닫힌 ZipFile에서 `testzip()`을 호출하면 `ValueError`가 발생합니다. 이전에는 `RuntimeError`가 발생했습니다.

`ZipFile.`**`write`**(*filename, arcname=None, compress_type=None, compresslevel=None*)

*filename*이라는 파일을 아카이브에 기록하고, 아카이브 이름으로 *arcname*을 지정합니다 (기본적으로, *filename*과 같지만, 드라이브 문자가 없고 선행 경로 구분 기호가 제거됩니다). 주어지면, *compress_type*은 새 항목에 대해 생성자의 *compression* 매개 변수에 제공된 값을 대체합니다. 마찬가지로, *compresslevel*은 주어지면 생성자를 대체합니다. 아카이브는 `'w'`, `'x'` 또는 `'a'` 모드로 열려 있어야 합니다.

> **참고**
>
> The ZIP file standard historically did not specify a metadata encoding, but strongly recommended CP437 (the original IBM PC encoding) for interoperability. Recent versions allow use of UTF-8 (only). In this module, UTF-8 will automatically be used to write the member names if they contain any non-ASCII characters. It is not possible to write member names in any encoding other than ASCII or UTF-8.

> **참고**
>
> 아카이브 이름은 아카이브 루트에 상대적이어야 합니다. 즉, 경로 구분 기호로 시작해서는 안 됩니다.

> **참고**
>
> `arcname`(또는 `arcname`이 제공되지 않으면 `filename`)에 널 바이트가 포함되어 있으면, 아카이브의 파일 이름이 널 바이트에서 잘립니다.

> **참고**
>
> A leading slash in the filename may lead to the archive being impossible to open in some zip programs on Windows systems.

버전 3.6에서 변경: `'r'` 모드로 만들어진 ZipFile이나 닫힌 ZipFile에서 `write()`를 호출하면 `ValueError`가 발생합니다. 이전에는 `RuntimeError`가 발생했습니다.

`ZipFile.`**`writestr`**(*zinfo_or_arcname*, *data*, *compress_type=None*, *compresslevel=None*)

파일을 아카이브에 기록합니다. 내용은 *data*이며, `str`이나 `bytes` 인스턴스일 수 있습니다; `str`이면 먼저 UTF-8로 인코딩됩니다. *zinfo_or_arcname*은 아카이브에 제공될 파일 이름이거나 `ZipInfo` 인스턴스입니다. 인스턴스이면 최소한 파일명, 날짜 및 시간을 지정해야 합니다. 이름이면, 날짜와 시간이 현재 날짜와 시간으로 설정됩니다. 아카이브는 `'w'`, `'x'` 또는 `'a'` 모드로 열려 있어야 합니다.

주어지면, *compress_type*은 새 항목에 대해 생성자의 *compression* 매개 변수에 제공되거나 *zinfo_or_arcname*(`ZipInfo` 인스턴스인 경우)의 값을 대체합니다. 마찬가지로, *compresslevel*은 주어지면 생성자를 대체합니다.

> **참고**
>
> `ZipInfo` 인스턴스를 *zinfo_or_arcname* 매개 변수로 전달할 때, 사용되는 압축 방법은 주어진 `ZipInfo` 인스턴스의 *compress_type* 멤버에 지정된 압축 방법입니다. 기본적으로, `ZipInfo` 생성자는 이 멤버를 `ZIP_STORED`로 설정합니다.

버전 3.2에서 변경: *compress_type* 인자.

버전 3.6에서 변경: `'r'` 모드로 만들어진 ZipFile이나 닫힌 ZipFile에서 `writestr()`을 호출하면, `ValueError`가 발생합니다. 이전에는 `RuntimeError`가 발생했습니다.

`ZipFile.`**`mkdir`**(*zinfo_or_directory*, *mode=511*)

Create a directory inside the archive. If *zinfo_or_directory* is a string, a directory is created inside the archive with the mode that is specified in the *mode* argument. If, however, *zinfo_or_directory* is a `ZipInfo` instance then the *mode* argument is ignored.

The archive must be opened with mode `'w'`, `'x'` or `'a'`.

Added in version 3.11.

다음과 같은 데이터 어트리뷰트도 사용할 수 있습니다:

`ZipFile.`**`filename`**

ZIP 파일의 이름.

`ZipFile.`**`debug`**

사용할 디버그 출력 수준. 이것은 `0`(기본값, 출력 없음)에서 `3`(가장 많은 출력)으로 설정될 수 있습니다. 디버깅 정보는 `sys.stdout`에 기록됩니다.

ZipFile.**comment**

ZIP 파일에 연관되는 주석은 *bytes* 객체입니다. 'w', 'x' 또는 'a' 모드로 만들어진 *ZipFile* 인스턴스에 주석을 대입하면, 65535바이트를 넘지 않아야 합니다. 이보다 긴 주석은 잘립니다.

13.7.2 Path 객체

class zipfile.**Path**(*root*, *at*='')

root zip 파일(*ZipFile* 생성자에 전달하기에 적합한 *ZipFile* 인스턴스나 file일 수 있습니다)에서 Path 객체를 생성합니다.

at은 zip 파일 내에서 이 Path의 위치를 지정합니다, 예를 들어 'dir/file.txt', 'dir/' 또는 ''. 기본값은 빈 문자열이며, 루트를 나타냅니다.

> **참고**
>
> The *Path* class does not sanitize filenames within the ZIP archive. Unlike the *ZipFile.extract()* and *ZipFile.extractall()* methods, it is the caller's responsibility to validate or sanitize filenames to prevent path traversal vulnerabilities (e.g., filenames containing "..." or absolute paths). When handling untrusted archives, consider resolving filenames using *os.path.abspath()* and checking against the target directory with *os.path.commonpath()*.

Path 객체는 *pathlib.Path* 객체의 다음 기능을 노출합니다:

/ 연산자나 joinpath를 사용하여 Path 객체를 순회할 수 있습니다.

Path.**name**

최종 경로 구성 요소.

Path.**open**(*mode*='r', *, *pwd*, **)

현재 경로에서 *ZipFile.open()*을 호출합니다. 지원되는 모드를 통해 읽기 또는 쓰기, 텍스트 또는 바이너리로 여는 것을 허락합니다: 'r', 'w', 'rb', 'wb'. 위치와 키워드 인자는 텍스트로 열 때 *io.TextIOWrapper*로 전달되고 그렇지 않으면 무시됩니다. pwd는 *ZipFile.open()*에 대한 pwd 매개변수입니다.

버전 3.9에서 변경: open에 텍스트와 바이너리 모드에 대한 지원이 추가되었습니다. 기본 모드는 이제 텍스트입니다.

버전 3.11.2에서 변경: The encoding parameter can be supplied as a positional argument without causing a *TypeError*. As it could in 3.9. Code needing to be compatible with unpatched 3.10 and 3.11 versions must pass all *io.TextIOWrapper* arguments, encoding included, as keywords.

Path.**iterdir**()

현재 디렉터리의 자식을 열거합니다.

Path.**is_dir**()

현재 컨텍스트가 디렉터리를 참조하면 True를 반환합니다.

Path.**is_file**()

현재 컨텍스트가 파일을 참조하면 True를 반환합니다.

Path.**is_symlink**()

현재 컨텍스트가 심볼릭 링크를 참조하면 True를 반환합니다.

Added in version 3.12.

버전 3.13에서 변경: Previously, is_symlink would unconditionally return False.

Path.**exists**()

현재 컨텍스트가 zip 파일에 있는 파일이나 디렉터리를 참조하면 True를 반환합니다.

Path.**suffix**

The last dot-separated portion of the final component, if any. This is commonly called the file extension.

Added in version 3.11: Added `Path.suffix` property.

Path.**stem**

접미사(suffix)가 없는 최종 경로 구성 요소.

Added in version 3.11: Added `Path.stem` property.

Path.**suffixes**

A list of the path' s suffixes, commonly called file extensions.

Added in version 3.11: Added `Path.suffixes` property.

Path.**read_text**(*, **)

현재 파일을 유니코드 텍스트로 읽습니다. 위치와 키워드 인자는 `io.TextIOWrapper`로 전달됩니다 (컨텍스트에 의해 암시되는 `buffer` 제외).

버전 3.11.2에서 변경: The `encoding` parameter can be supplied as a positional argument without causing a `TypeError`. As it could in 3.9. Code needing to be compatible with unpatched 3.10 and 3.11 versions must pass all `io.TextIOWrapper` arguments, `encoding` included, as keywords.

Path.**read_bytes**()

현재 파일을 바이트열로 읽습니다.

Path.**joinpath**(*other)

Return a new Path object with each of the *other* arguments joined. The following are equivalent:

```
>>> Path(...).joinpath('child').joinpath('grandchild')
>>> Path(...).joinpath('child', 'grandchild')
>>> Path(...) / 'child' / 'grandchild'
```

버전 3.10에서 변경: Prior to 3.10, `joinpath` was undocumented and accepted exactly one parameter.

The zipp project provides backports of the latest path object functionality to older Pythons. Use `zipp.Path` in place of `zipfile.Path` for early access to changes.

13.7.3 PyZipFile 객체

`PyZipFile` 생성자는 `ZipFile` 생성자와 같은 매개 변수와 하나의 추가 매개 변수 *optimize*를 취합니다.

class zipfile.**PyZipFile**(*file, mode='r', compression=ZIP_STORED, allowZip64=True, optimize=-1*)

버전 3.2에서 변경: *optimize* 매개 변수를 추가했습니다.

버전 3.4에서 변경: ZIP64 확장은 기본적으로 활성화됩니다.

인스턴스에는 `ZipFile` 객체의 메서드들 외에 한 가지 추가 메서드가 있습니다:

writepy(*pathname, basename='', filterfunc=None*)

파일 `*.py`를 검색하고 해당 파일을 아카이브에 추가합니다.

`PyZipFile`에 대한 *optimize* 매개 변수가 제공되지 않았거나 `-1`이면, 해당 파일은 `*.pyc` 파일이며, 필요하면 컴파일합니다.

`PyZipFile`에 대한 *optimize* 매개 변수가 `0`, `1` 또는 `2`이면, 해당 최적화 수준(`compile()`을 참조하십시오)의 파일 만 아카이브에 추가되며, 필요하면 컴파일합니다.

*pathname*이 파일이면, 파일 이름은 `.py`로 끝나야하며, 단지 그 (해당 `*.pyc`) 파일 만 최상위 수준에 추가됩니다 (경로 정보 없음). *pathname*이 `.py`로 끝나지 않는 파일이면, `RuntimeError`가 발생합니다. 디렉터리이고, 디렉터리가 패키지 디렉터리가 아니면, 모든 파일 `*.pyc`가 최상위 수준에 추가됩니다. 디렉터리가 패키지 디렉터리이면, 모든 `*.pyc`가 패키지 이름의 파일 경로 아래에 추가되고, 서브 디렉터리가 패키지 디렉터리이면, 이들 모두도 재귀적으로 정렬된 순서로 추가됩니다.

*basename*은 내부 전용입니다.

*filterfunc*가 주어지면, 단일 문자열 인자를 취하는 함수여야 합니다. 아카이브에 추가되기 전에 각 경로(개별 전체 파일 경로를 포함합니다)를 전달합니다. *filterfunc*가 거짓 값을 반환하면, 경로가 추가되지 않으며, 디렉터리이면 내용이 무시됩니다. 예를 들어, 테스트 파일이 모두 `test` 디렉터리에 있거나 문자열 `test_`로 시작하면, *filterfunc*를 사용하여 해당 파일들을 제외할 수 있습니다:

```
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
...     return (not (fn == 'test' or fn.startswith('test_')))
...
>>> zf.writepy('myprog', filterfunc=notests)
```

`writepy()` 메서드는 다음과 같은 파일 이름으로 아카이브를 만듭니다:

```
string.pyc                    # 최상위 이름
test/__init__.pyc             # 패키지 디렉터리
test/testall.pyc              # 모듈 test.testall
test/bogus/__init__.pyc       # 서브 패키지 디렉터리
test/bogus/myfile.pyc         # 서브 모듈 test.bogus.myfile
```

버전 3.4에서 변경: *filterfunc* 매개 변수를 추가했습니다.

버전 3.6.2에서 변경: *pathname* 매개 변수는 경로류 객체를 받아들입니다.

버전 3.7에서 변경: 재귀는 디렉터리 항목을 정렬합니다.

13.7.4 ZipInfo 객체

`ZipInfo` 클래스의 인스턴스는 `ZipFile` 객체의 `getinfo()`와 `infolist()` 메서드가 반환합니다. 각 객체는 ZIP 아카이브의 단일 멤버에 대한 정보를 저장합니다.

파일 시스템 파일의 `ZipInfo` 인스턴스를 만드는 클래스 메서드가 하나 있습니다:

classmethod `ZipInfo.`**`from_file`**(*filename, arcname=None, *, strict_timestamps=True*)

zip 파일에 파일을 추가할 수 있도록, 파일 시스템의 파일에 대한 `ZipInfo` 인스턴스를 생성합니다.

*filename*은 파일 시스템에서 파일이나 디렉터리의 경로여야 합니다.

*arcname*이 지정되면, 아카이브 내에서의 이름으로 사용됩니다. *arcname*을 지정하지 않으면, 이름은 *filename*과 같지만, 드라이브 문자와 선행 경로 구분 기호가 제거됩니다.

strict_timestamps 인자를 `False`로 설정하면, 1980-01-01 이전의 zip 파일을 허용하는 대신 타임 스탬프를 1980-01-01로 설정합니다. 2107-12-31 이후의 파일에 대해서도 비슷한 동작이 발생하며, 타임 스탬프는 역시 한곗값으로 설정됩니다.

Added in version 3.6.

버전 3.6.2에서 변경: *filename* 매개 변수는 경로류 객체를 받아들입니다.

버전 3.8에서 변경: *strict_timestamps* 키워드 전용 매개 변수를 추가했습니다.

인스턴스에는 다음과 같은 메서드와 어트리뷰트가 있습니다:

`ZipInfo.`**`is_dir`**()

이 아카이브 멤버가 디렉터리이면 `True`를 반환합니다.

이것은 항목 이름을 사용합니다: 디렉터리는 항상 `/`로 끝나야 합니다.

Added in version 3.6.

`ZipInfo.`**`filename`**

아카이브에서의 파일의 이름.

ZipInfo.**date_time**

The time and date of the last modification to the archive member. This is a tuple of six values representing the "last [modified] file time" and "last [modified] file date" fields from the ZIP file' s central directory.

The tuple contains:

인덱스	값
0	연도 (>= 1980)
1	월 (1에서 시작)
2	월 중 일 (1에서 시작)
3	시간 (0에서 시작)
4	분 (0에서 시작)
5	초 (0에서 시작)

> **참고**
>
> The ZIP format supports multiple timestamp fields in different locations (central directory, extra fields for NTFS/UNIX systems, etc.). This attribute specifically returns the timestamp from the central directory. The central directory timestamp format in ZIP files does not support timestamps before 1980. While some extra field formats (such as UNIX timestamps) can represent earlier dates, this attribute only returns the central directory timestamp.
>
> The central directory timestamp is interpreted as representing local time, rather than UTC time, to match the behavior of other zip tools.

ZipInfo.**compress_type**

아카이브 멤버의 압축 유형.

ZipInfo.**comment**

bytes 객체로 제공되는 개별 아카이브 멤버에 대한 주석.

ZipInfo.**extra**

확장 필드 데이터. PKZIP Application Note는 이 *bytes* 객체에 포함된 데이터의 내부 구조에 대한 주석을 포함합니다.

ZipInfo.**create_system**

ZIP 아카이브를 만든 시스템.

ZipInfo.**create_version**

ZIP 아카이브를 만든 PKZIP 버전.

ZipInfo.**extract_version**

아카이브를 추출하기 위해 필요한 PKZIP 버전.

ZipInfo.**reserved**

반드시 0이어야 합니다.

ZipInfo.**flag_bits**

ZIP 플래그 비트.

ZipInfo.**volume**

파일 헤더의 볼륨 번호.

ZipInfo.**internal_attr**

내부 어트리뷰트.

ZipInfo.**external_attr**

외부 파일 어트리뷰트.

ZipInfo.**header_offset**
파일 헤더의 바이트 오프셋.

ZipInfo.**CRC**
압축되지 않은 파일의 CRC-32.

ZipInfo.**compress_size**
압축된 데이터의 크기.

ZipInfo.**file_size**
압축되지 않은 파일의 크기.

13.7.5 명령 줄 인터페이스

zipfile 모듈은 ZIP 아카이브와 상호 작용하기 위한 간단한 명령 줄 인터페이스를 제공합니다.

새 ZIP 아카이브를 만들려면 *-c* 옵션 뒤에 이름을 지정한 다음 포함해야 할 파일 이름을 나열하십시오:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

디렉터리 전달도 허용됩니다:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

ZIP 아카이브를 지정된 디렉터리로 추출하려면, *-e* 옵션을 사용하십시오:

```
$ python -m zipfile -e monty.zip target-dir/
```

ZIP 아카이브에 있는 파일 목록을 보려면, *-l* 옵션을 사용하십시오:

```
$ python -m zipfile -l monty.zip
```

명령 줄 옵션

-l <zipfile>
--list <zipfile>
zip 파일에 있는 파일을 나열합니다.

-c <zipfile> <source1> ... <sourceN>
--create <zipfile> <source1> ... <sourceN>
소스 파일로 zip 파일을 만듭니다.

-e <zipfile> <output_dir>
--extract <zipfile> <output_dir>
zip 파일을 대상 디렉터리로 추출합니다.

-t <zipfile>
--test <zipfile>
zip 파일이 유효한지 테스트합니다.

--metadata-encoding <encoding>
Specify encoding of member names for *-l*, *-e* and *-t*.

Added in version 3.11.

13.7.6 압축 해제 함정

아래 나열된 일부 함정으로 인해 zipfile 모듈에서의 추출이 실패할 수 있습니다.

파일 자체에서

잘못된 암호 / CRC 체크섬 / ZIP 형식 또는 지원되지 않는 압축 방법 / 암호 해독으로 인해 압축 해제에 실패할 수 있습니다.

파일 시스템 제한

다른 파일 시스템의 제한을 초과하면 압축 해제에 실패할 수 있습니다. 가령 디렉터리 항목에 허용되는 문자, 파일 이름 길이, 경로명 길이, 단일 파일 크기 및 파일 수 등.

자원 제한

메모리나 디스크 볼륨이 부족하면 압축 해제에 실패합니다. 예를 들어, 압축 해제 폭탄(일명 ZIP bomb)을 zipfile 라이브러리에 적용하면 디스크 볼륨이 소진될 수 있습니다.

중단

Ctrl-C 누르기나 압축 해제 프로세스를 죽이는 것과 같은 압축 해제 중 중단으로 인해 아카이브 압축 해제가 불완전할 수 있습니다.

추출의 기본 동작

기본 추출 동작을 모르면 예기치 않은 압축 해제 결과가 발생할 수 있습니다. 예를 들어, 같은 아카이브를 두 번 추출하면, 묻지 않고 파일을 덮어씁니다.

13.8 `tarfile` — tar 아카이브 파일 읽기와 쓰기

소스 코드: Lib/tarfile.py

`tarfile` 모듈을 사용하면 gzip, bz2 및 lzma 압축을 사용하는 것을 포함하여, tar 아카이브를 읽고 쓸 수 있습니다. `.zip` 파일을 읽거나 쓰려면 `zipfile` 모듈이나 *shutil*에 있는 고수준 함수를 사용하십시오.

몇 가지 세부 사항:

- reads and writes `gzip`, `bz2`, `compression.zstd`, and `lzma` compressed archives if the respective modules are available.
- POSIX.1-1988 (ustar) 형식에 대한 읽기/쓰기 지원.
- *longname*과 *longlink* 확장을 포함한 GNU tar 형식에 대한 읽기/쓰기 지원, 스파스(sparse) 파일 복원을 포함하여 모든 *sparse* 확장 변형에 대한 읽기 전용 지원.
- POSIX.1-2001 (pax) 형식에 대한 읽기/쓰기 지원.
- 디렉터리, 일반 파일, 하드 링크, 심볼릭 링크, fifo, 문자 장치 및 블록 장치를 처리하고 타임 스탬프, 액세스 권한 및 소유자와 같은 파일 정보를 획득하고 복원할 수 있습니다.

버전 3.3에서 변경: `lzma` 압축에 대한 지원이 추가되었습니다.

버전 3.12에서 변경: Archives are extracted using a *filter*, which makes it possible to either limit surprising/dangerous features, or to acknowledge that they are expected and the archive is fully trusted.

버전 3.14에서 변경: Set the default extraction filter to `data`, which disallows some dangerous features such as links to absolute paths or paths outside of the destination. Previously, the filter strategy was equivalent to `fully_trusted`.

버전 3.14에서 변경: Added support for Zstandard compression using `compression.zstd`.

`tarfile.`**`open`**(*name=None*, *mode='r'*, *fileobj=None*, *bufsize=10240*, ***kwargs*)

경로명 *name*에 대한 `TarFile` 객체를 반환합니다. `TarFile` 객체와 허용되는 키워드 인자에 대한 자세한 정보는 *TarFile 객체*를 참조하십시오.

*mode*는 `'filemode[:compression]'` 형식의 문자열이어야 하며, 기본값은 `'r'`입니다. 다음은 mode 조합의 전체 목록입니다:

모드	동작
'r' or 'r:*'	투명한 압축으로 읽기 위해 엽니다 (권장합니다).
'r:'	압축하지 않고 독점적으로 읽기 위해 엽니다.
'r:gz'	gzip 압축으로 읽기 위해 엽니다.
'r:bz2'	bzip2 압축으로 읽기 위해 엽니다.
'r:xz'	lzma 압축으로 읽기 위해 엽니다.
'r:zst'	Open for reading with Zstandard compression.
'x' 또 는 'x:'	압축하지 않고 독점적으로 tar 파일을 만듭니다. 이미 있으면 *FileExistsError* 예외를 발생시킵니다.
'x:gz'	gzip 압축으로 tar 파일을 만듭니다. 이미 있으면 *FileExistsError* 예외를 발생시킵니다.
'x:bz2'	bzip2 압축으로 tar 파일을 만듭니다. 이미 있으면 *FileExistsError* 예외를 발생시킵니다.
'x:xz'	lzma 압축으로 tar 파일을 만듭니다. 이미 있으면 *FileExistsError* 예외를 발생시킵니다.
'x:zst'	Create a tarfile with Zstandard compression. Raise a *FileExistsError* exception if it already exists.
'a' or 'a:'	압축하지 않고 추가하기 위해 엽니다. 파일이 없으면 만들어집니다.
'w' or 'w:'	압축되지 않은 쓰기를 위해 엽니다.
'w:gz'	gzip 압축 쓰기를 위해 엽니다.
'w:bz2'	bzip2 압축 쓰기를 위해 엽니다.
'w:xz'	lzma 압축 쓰기를 위해 엽니다.
'w:zst'	Open for Zstandard compressed writing.

'a:gz', 'a:bz2' 또는 'a:xz'는 불가능함에 유의하십시오. *mode*가 특정 (압축된) 파일을 읽기 위해 여는 데 적합하지 않으면 *ReadError*가 발생합니다. 이것을 피하려면 *mode* 'r'을 사용하십시오. 압축 방법이 지원되지 않으면, *CompressionError*가 발생합니다.

*fileobj*가 지정되면, *name*에 대해 바이너리 모드로 열린 파일 객체의 대안으로 사용됩니다. 위치 0에 있다고 가정합니다.

'w:gz', 'x:gz', 'w|gz', 'w:bz2', 'x:bz2', 'w|bz2' 모드의 경우, *tarfile.open()*은 파일의 압축 수준을 지정하는 키워드 인자 *compresslevel*(기본값 9)을 받아들입니다.

For modes 'w:xz', 'x:xz' and 'w|xz', *tarfile.open()* accepts the keyword argument *preset* to specify the compression level of the file.

For modes 'w:zst', 'x:zst' and 'w|zst', *tarfile.open()* accepts the keyword argument *level* to specify the compression level of the file. The keyword argument *options* may also be passed, providing advanced Zstandard compression parameters described by *CompressionParameter*. The keyword argument *zstd_dict* can be passed to provide a *ZstdDict*, a Zstandard dictionary used to improve compression of smaller amounts of data.

특별한 목적으로, *mode*에는 두 번째 형식이 있습니다: 'filemode|[compression]'. *tarfile.open()*은 데이터를 블록 스트림으로 처리하는 *TarFile* 객체를 반환합니다. 파일에서 무작위 탐색 (random seeking)을 수행하지 않습니다. 주어지면□ *fileobj*는 (*mode*에 따라) 바이트열과 호환되는 *read()*나 *write()* 메서드가 있는 임의의 객체일 수 있습니다. *bufsize*는 블록 크기를 지정하고 기본값은 20 * 512 바이트입니다. 이 변형을 예를 들어 sys.stdin.buffer, 소켓 파일 객체 또는 테이프 장치와 함께 사용하십시오. 그러나, 이러한 *TarFile* 객체는 무작위 액세스를 허용하지 않는다는 제한이 있습니다 (예를 참조하십시오). 현재 가능한 모드는 다음과 같습니다:

모드	동작
`'r|*'`	투명한 압축으로 읽기 위해 tar 블록의 스트림을 엽니다.
`'r|'`	읽기 위해 압축되지 않은 tar 블록의 스트림을 엽니다.
`'r|gz'`	읽기 위해 gzip 압축 스트림을 엽니다.
`'r|bz2'`	읽기 위해 bzip2 압축 스트림을 엽니다.
`'r|xz'`	읽기 위해 lzma 압축 스트림을 엽니다.
`'r|zst'`	Open a Zstandard compressed *stream* for reading.
`'w|'`	쓰기 위해 압축되지 않은 스트림을 엽니다.
`'w|gz'`	쓰기 위해 gzip 압축 스트림을 엽니다.
`'w|bz2'`	쓰기 위해 bzip2 압축 스트림을 엽니다.
`'w|xz'`	쓰기 위해 lzma 압축 스트림을 엽니다.
`'w|zst'`	Open a Zstandard compressed *stream* for writing.

버전 3.5에서 변경: `'x'` (독점 생성) 모드가 추가되었습니다.

버전 3.6에서 변경: *name* 매개 변수는 경로류 객체를 받아들입니다.

버전 3.12에서 변경: The *compresslevel* keyword argument also works for streams.

버전 3.14에서 변경: The *preset* keyword argument also works for streams.

class tarfile.**TarFile**

tar 아카이브를 읽고 쓰는 클래스. 이 클래스를 직접 사용하지 마십시오: 대신 `tarfile.open()`을 사용하십시오. *TarFile* 객체를 참조하십시오.

tarfile.**is_tarfile**(*name*)

*name*이 `tarfile` 모듈이 읽을 수 있는 tar 아카이브 파일이면 `True`를 반환합니다. *name*은 `str`, 파일 또는 파일류 객체일 수 있습니다.

버전 3.9에서 변경: 파일과 파일류 객체 지원.

`tarfile` 모듈은 다음 예외를 정의합니다:

exception tarfile.**TarError**

모든 `tarfile` 예외의 베이스 클래스.

exception tarfile.**ReadError**

`tarfile` 모듈에서 처리할 수 없거나 어떤 식으로든 유효하지 않은 tar 아카이브가 열릴 때 발생합니다.

exception tarfile.**CompressionError**

압축 방법이 지원되지 않거나 데이터를 올바르게 디코딩할 수 없을 때 발생합니다.

exception tarfile.**StreamError**

스트림류 `TarFile` 객체에 일반적인 제한 사항으로 인해 발생합니다.

exception tarfile.**ExtractError**

`TarFile.extract()`를 사용할 때 치명적이지 않은 에러에 대해 발생하지만, `TarFile.errorlevel== 2`일 때만 발생합니다.

exception tarfile.**HeaderError**

버퍼가 유효하지 않을 때 `TarInfo.frombuf()`에 의해 발생합니다.

exception tarfile.**FilterError**

Base class for members *refused* by filters.

tarinfo

Information about the member that the filter refused to extract, as *TarInfo*.

exception tarfile.**AbsolutePathError**

Raised to refuse extracting a member with an absolute path.

exception tarfile.**OutsideDestinationError**

Raised to refuse extracting a member outside the destination directory.

exception tarfile.**SpecialFileError**

Raised to refuse extracting a special file (e.g. a device or pipe).

exception tarfile.**AbsoluteLinkError**

Raised to refuse extracting a symbolic link with an absolute path.

exception tarfile.**LinkOutsideDestinationError**

Raised to refuse extracting a symbolic link pointing outside the destination directory.

exception tarfile.**LinkFallbackError**

Raised to refuse emulating a link (hard or symbolic) by extracting another archive member, when that member would be rejected by the filter location. The exception that was raised to reject the replacement member is available as `BaseException.__context__`.

Added in version 3.14.

모듈 수준에서 다음 상수를 사용할 수 있습니다:

tarfile.**ENCODING**

기본 문자 인코딩: 윈도우의 경우 `'utf-8'`, 그렇지 않으면 *sys.getfilesystemencoding()* 이 반환하는 값.

tarfile.**REGTYPE**

tarfile.**AREGTYPE**

A regular file *type*.

tarfile.**LNKTYPE**

A link (inside tarfile) *type*.

tarfile.**SYMTYPE**

A symbolic link *type*.

tarfile.**CHRTYPE**

A character special device *type*.

tarfile.**BLKTYPE**

A block special device *type*.

tarfile.**DIRTYPE**

A directory *type*.

tarfile.**FIFOTYPE**

A FIFO special device *type*.

tarfile.**CONTTYPE**

A contiguous file *type*.

tarfile.**GNUTYPE_LONGNAME**

A GNU tar longname *type*.

tarfile.**GNUTYPE_LONGLINK**

A GNU tar longlink *type*.

tarfile.**GNUTYPE_SPARSE**

A GNU tar sparse file *type*.

다음의 각 상수는 *tarfile* 모듈이 만들 수 있는 tar 아카이브 형식을 정의합니다. 자세한 내용은 *지원되는 tar 형식* 섹션을 참조하십시오.

tarfile.**USTAR_FORMAT**

POSIX.1-1988 (ustar) 형식.

tarfile.**GNU_FORMAT**

GNU tar 형식.

tarfile.**PAX_FORMAT**

POSIX.1-2001 (pax) 형식.

tarfile.**DEFAULT_FORMAT**

아카이브를 만들기 위한 기본 형식. 이것은 현재 *PAX_FORMAT*입니다.

버전 3.8에서 변경: 새 아카이브의 기본 형식이 *GNU_FORMAT*에서 *PAX_FORMAT*으로 변경되었습니다.

더 보기

모듈 *zipfile*
zipfile 표준 모듈의 설명서.

아카이브 연산
표준 *shutil* 모듈이 제공하는 고수준 아카이브 기능에 대한 설명서.

GNU tar manual, Basic Tar Format
GNU tar 확장을 포함한, tar 아카이브 파일에 대한 설명서.

13.8.1 TarFile 객체

TarFile 객체는 tar 아카이브에 대한 인터페이스를 제공합니다. tar 아카이브는 블록의 시퀀스입니다. 아카이브 멤버(저장된 파일)는 헤더 블록과 그 뒤에 오는 데이터 블록으로 구성됩니다. tar 아카이브에 파일을 여러 번 저장할 수 있습니다. 각 아카이브 멤버는 *TarInfo* 객체로 표현됩니다. 세부 사항은 *TarInfo* 객체를 참조하십시오.

TarFile 객체는 `with` 문에서 컨텍스트 관리자로 사용될 수 있습니다. 블록이 완료되면 자동으로 닫힙니다. 예외가 있는 경우, 쓰기 위해 열린 아카이브는 마무리되지 않음에 유의하십시오; 내부적으로 사용된 파일 객체만 닫힙니다. 사용 사례는 예 섹션을 참조하십시오.

Added in version 3.2: 컨텍스트 관리 프로토콜에 대한 지원이 추가되었습니다.

class tarfile.**TarFile**(*name=None, mode='r', fileobj=None, format=DEFAULT_FORMAT, tarinfo=TarInfo, dereference=False, ignore_zeros=False, encoding=ENCODING, errors='surrogateescape', pax_headers=None, debug=0, errorlevel=1, stream=False*)

다음의 모든 인자는 선택적이며 인스턴스 어트리뷰트로도 액세스 할 수 있습니다.

*name*은 아카이브의 경로명입니다. *name*은 경로류 객체일 수 있습니다. *fileobj*가 주어지면 생략할 수 있습니다. 이 경우, 파일 객체의 `name` 어트리뷰트가 있다면 사용됩니다.

*mode*는 기존 아카이브에서 읽는 `'r'`, 기존 파일에 데이터를 추가하는 `'a'`, 기존 파일을 덮어쓰는 새 파일을 만드는 `'w'` 또는 존재하지 않을 때만 새 파일을 만드는 `'x'`입니다.

*fileobj*가 주어지면, 데이터를 읽거나 쓰는 데 사용됩니다. 그것이 결정될 수 있다면, *mode*는 *fileobj*의 모드에 의해 재정의됩니다. *fileobj*는 위치 0에서 사용됩니다.

참고

*TarFile*이 닫힐 때, *fileobj*는 닫히지 않습니다.

*format*은 쓰기를 위한 아카이브 형식을 제어합니다. 모듈 수준에서 정의되는 상수 *USTAR_FORMAT*, *GNU_FORMAT* 또는 *PAX_FORMAT* 중 하나여야 합니다. 읽을 때, 형식은 자동 감지됩니다, 단일 아카이브에 다른 형식이 존재할 때조차 그렇습니다.

tarinfo 인자를 사용하여 기본 *TarInfo* 클래스를 다른 클래스로 바꿀 수 있습니다.

*dereference*가 `False`이면, 아카이브에 심볼릭과 하드 링크를 추가합니다. `True`이면, 대상 파일의 내용을 아카이브에 추가합니다. 이는 심볼릭 링크를 지원하지 않는 시스템에는 영향을 미치지 않습니다.

*ignore_zeros*가 `False`이면, 빈 블록을 아카이브의 끝으로 처리합니다. `True`이면, 비어있는 (그래서 잘못된) 블록을 건너뛰고 최대한 많은 멤버를 확보하려고 합니다. 이어붙였거나 손상된 아카이브를 읽을 때만 유용합니다.

*debug*는 `0`(디버그 메시지 없음)에서 `3`(모든 디버그 메시지)까지 설정할 수 있습니다. 메시지는 `sys.stderr`에 기록됩니다.

errorlevel controls how extraction errors are handled, see *the corresponding attribute*.

*encoding*과 *errors* 인자는 아카이브를 읽거나 쓰는 데 사용되는 문자 인코딩과 변환 에러 처리 방법을 정의합니다. 기본 설정은 대부분의 사용자에게 적용됩니다. 자세한 정보는 유니코드 문제 섹션을 참조하십시오.

pax_headers 인자는 선택적인 문자열의 딕셔너리로, *format*이 `PAX_FORMAT`이면 pax 전역 헤더로 추가됩니다.

If *stream* is set to `True` then while reading the archive info about files in the archive are not cached, saving memory.

버전 3.2에서 변경: *errors* 인자의 기본값으로 `'surrogateescape'`를 사용합니다.

버전 3.5에서 변경: `'x'` (독점 생성) 모드가 추가되었습니다.

버전 3.6에서 변경: *name* 매개 변수는 경로류 객체를 받아들입니다.

버전 3.13에서 변경: *stream* 매개 변수를 추가했습니다.

classmethod `TarFile.`**`open`**`(...)`

대체 생성자. `tarfile.open()` 함수는 실제로 이 클래스 메서드의 바로 가기입니다.

`TarFile.`**`getmember`**`(`*name*`)`

멤버 *name*에 대한 `TarInfo` 객체를 반환합니다. 아카이브에서 *name*을 찾을 수 없으면 `KeyError`가 발생합니다.

> **참고**
>
> 아카이브에서 멤버가 두 번 이상 등장하면, 마지막 등장을 최신 버전으로 간주합니다.

`TarFile.`**`getmembers`**`()`

아카이브 멤버를 `TarInfo` 객체의 리스트로 반환합니다. 리스트는 아카이브의 멤버와 순서가 같습니다.

`TarFile.`**`getnames`**`()`

멤버를 이름의 리스트로 반환합니다. `getmembers()`가 반환하는 리스트와 순서가 같습니다.

`TarFile.`**`list`**`(`*verbose=True, *, members=None*`)`

목차를 `sys.stdout`으로 인쇄합니다. *verbose*가 `False`이면, 멤버 이름만 인쇄됩니다. `True`이면, **`ls -l`**과 유사한 출력이 생성됩니다. 선택적 *members*가 제공되면, `getmembers()`가 반환한 리스트의 부분집합이어야 합니다.

버전 3.5에서 변경: *members* 매개 변수를 추가했습니다.

`TarFile.`**`next`**`()`

`TarFile`을 읽기 위해 열었을 때, 아카이브의 다음 멤버를 `TarInfo` 객체로 반환합니다. 더는 없으면, `None`을 반환합니다.

`TarFile.`**`extractall`**`(`*path='.', members=None, *, numeric_owner=False, filter=None*`)`

아카이브의 모든 멤버(members)를 현재 작업 디렉터리나 디렉터리 *path*로 추출합니다. 선택적 *members*가 제공되면, `getmembers()`가 반환하는 리스트의 부분집합이어야 합니다. 소유자, 수정 시간 및 권한과 같은 디렉터리 정보는 모든 멤버가 추출된 후에 설정됩니다. 이것은 두 가지 문제를 해결하기

위한 것입니다: 디렉터리의 수정 시간은 파일이 생성될 때마다 재설정됩니다. 또한 디렉터리의 권한이 쓰기를 허용하지 않으면, 파일 추출이 실패합니다.

*numeric_owner*가 `True`이면, tar 파일의 uid와 gid 번호는 추출된 파일의 소유자/그룹을 설정하는 데 사용됩니다. 그렇지 않으면, tar 파일의 이름값이 사용됩니다.

The *filter* argument specifies how `members` are modified or rejected before extraction. See *Extraction filters* for details. It is recommended to set this explicitly only if specific *tar* features are required, or as `filter='data'` to support Python versions with a less secure default (3.13 and lower).

> **경고**
>
> Never extract archives from untrusted sources without prior inspection.
>
> Since Python 3.14, the default (`data`) will prevent the most dangerous security issues. However, it will not prevent *all* unintended or insecure behavior. Read the *Extraction filters* section for details.

버전 3.5에서 변경: *numeric_owner* 매개 변수를 추가했습니다.

버전 3.6에서 변경: *path* 매개 변수는 경로류 객체를 받아들입니다.

버전 3.12에서 변경: *filter* 매개 변수를 추가했습니다.

버전 3.14에서 변경: The *filter* parameter now defaults to `'data'`.

`TarFile.`**`extract`**(*member*, *path=''*, *set_attrs=True*, **, numeric_owner=False*, *filter=None*)

전체 이름을 사용하여, 아카이브에서 현재 작업 디렉터리로 멤버(member)를 추출합니다. 파일 정보는 최대한 정확하게 추출됩니다. *member*는 파일명이나 `TarInfo` 객체일 수 있습니다. *path*를 사용하여 다른 디렉터리를 지정할 수 있습니다. *path*는 경로류 객체일 수 있습니다. *set_attrs*가 거짓이 아닌 한 파일 어트리뷰트(소유자, 수정 시간, 모드)가 설정됩니다.

The *numeric_owner* and *filter* arguments are the same as for `extractall()`.

> **참고**
>
> `extract()` 메서드는 여러 추출 문제를 처리하지 않습니다. 대부분의 경우 `extractall()` 메서드 사용을 고려해야 합니다.

> **경고**
>
> Never extract archives from untrusted sources without prior inspection. See the warning for `extractall()` for details.

버전 3.2에서 변경: *set_attrs* 매개 변수를 추가했습니다.

버전 3.5에서 변경: *numeric_owner* 매개 변수를 추가했습니다.

버전 3.6에서 변경: *path* 매개 변수는 경로류 객체를 받아들입니다.

버전 3.12에서 변경: *filter* 매개 변수를 추가했습니다.

`TarFile.`**`extractfile`**(*member*)

아카이브에서 멤버(member)를 파일 객체로 추출합니다. *member*는 파일명이나 `TarInfo` 객체일 수 있습니다. *member*가 일반 파일이나 링크이면, `io.BufferedReader` 객체가 반환됩니다. 다른 모든 기존 멤버에 대해서는, `None`이 반환됩니다. *member*가 아카이브에 없으면, `KeyError`가 발생합니다.

버전 3.3에서 변경: `io.BufferedReader` 객체를 반환합니다.

버전 3.13에서 변경: The returned `io.BufferedReader` object has the `mode` attribute which is always equal to `'rb'`.

TarFile.**errorlevel**: *int*

*errorlevel*이 0이면, `TarFile.extract()`와 `TarFile.extractall()`를 사용할 때 에러가 무시됩니다. 그런데도, *debug*이 0보다 크면, 디버그 출력에 에러 메시지로 나타납니다. 1이면 (기본값), 모든 치명적 에러가 `OSError`나 `FilterError` 예외로 발생합니다. 2이면, 모든 치명적이지 않은 에러도 `TarError` 예외로 발생합니다.

Some exceptions, e.g. ones caused by wrong argument types or data corruption, are always raised.

Custom *extraction filters* should raise `FilterError` for *fatal* errors and `ExtractError` for *non-fatal* ones.

Note that when an exception is raised, the archive may be partially extracted. It is the user' s responsibility to clean up.

TarFile.**extraction_filter**

Added in version 3.12.

The *extraction filter* used as a default for the *filter* argument of `extract()` and `extractall()`.

The attribute may be `None` or a callable. String names are not allowed for this attribute, unlike the *filter* argument to `extract()`.

If `extraction_filter` is `None` (the default), extraction methods will use the `data` filter by default.

The attribute may be set on instances or overridden in subclasses. It also is possible to set it on the `TarFile` class itself to set a global default, although, since it affects all uses of *tarfile*, it is best practice to only do so in top-level applications or `site configuration`. To set a global default this way, a filter function needs to be wrapped in `staticmethod()` to prevent injection of a `self` argument.

버전 3.14에서 변경: The default filter is set to `data`, which disallows some dangerous features such as links to absolute paths or paths outside of the destination. Previously, the default was equivalent to `fully_trusted`.

TarFile.**add**(*name, arcname=None, recursive=True, *, filter=None*)

name 파일을 아카이브에 추가합니다. *name*은 모든 유형의 파일(디렉터리, fifo, 심볼릭 링크 등)일 수 있습니다. 지정되면, *arcname*은 아카이브에 있는 파일의 대체 이름을 지정합니다. 디렉터리는 기본적으로 재귀적으로 추가됩니다. *recursive*를 `False`로 설정하면, 이를 피할 수 있습니다. 재귀는 항목을 정렬된 순서로 추가합니다. *filter*가 제공되면, `TarInfo` 객체 인자를 취하고 변경된 `TarInfo` 객체를 반환하는 함수여야 합니다. 이것이 대신 `None`을 반환하면, `TarInfo` 객체가 아카이브에서 제외됩니다. 예는 예를 참조하십시오.

버전 3.2에서 변경: *filter* 매개 변수를 추가했습니다.

버전 3.7에서 변경: 재귀는 항목을 정렬된 순서로 추가합니다.

TarFile.**addfile**(*tarinfo, fileobj=None*)

`TarInfo` 객체 *tarinfo*를 아카이브에 추가합니다. *tarinfo*가 0이 아닌 크기의 일반 파일이면, *fileobj* 인자는 바이너리 파일이어야 하고, 여기서 `tarinfo.size` 바이트를 읽어서 아카이브에 추가합니다. `TarInfo` 객체를 직접 혹은 `gettarinfo()`를 사용하여 만들 수 있습니다.

버전 3.13에서 변경: *fileobj* must be given for non-zero-sized regular files.

TarFile.**gettarinfo**(*name=None, arcname=None, fileobj=None*)

기존 파일에서 `os.stat()`이나 이와 동등한 것의 결과로부터 `TarInfo` 객체를 만듭니다. 파일은 *name*으로 이름을 지정하거나, 파일 기술자를 갖는 파일 객체 *fileobj*로 지정됩니다. *name*은 경로류 객체일 수 있습니다. 주어지면, *arcname*은 아카이브에 있는 파일의 대체 이름을 지정하고, 그렇지 않으면, 이름은 *fileobj*의 `name` 어트리뷰트나 *name* 인자에서 취합니다. 이름은 텍스트 문자열이어야 합니다.

`addfile()`을 사용하여 추가하기 전에 `TarInfo`의 일부 어트리뷰트를 수정할 수 있습니다. 파일 객체가 파일의 시작 부분에 위치한 일반 파일 객체가 아니면, `size`와 같은 어트리뷰트를 수정해야 할 수 있습니다. `GzipFile`과 같은 객체가 이 상황에 해당합니다. `name`도 수정될 수 있으며, 이 경우 *arcname*은 더미 문자열일 수 있습니다.

버전 3.6에서 변경: *name* 매개 변수는 경로류 객체를 받아들입니다.

TarFile.**close**()

`TarFile`을 닫습니다. 쓰기 모드에서는, 두 개의 마무리 0블록이 아카이브에 추가됩니다.

TarFile.**pax_headers**: *dict*

pax 전역 헤더의 키-값 쌍을 포함하는 딕셔너리.

13.8.2 TarInfo 객체

TarInfo 객체는 *TarFile*에 있는 하나의 멤버를 나타냅니다. 파일의 모든 필수 어트리뷰트(파일 유형, 크기, 시간, 권한, 소유자 등과 같은)를 저장하는 것 외에도, 파일 유형을 결정하는 유용한 메서드를 제공합니다. 파일의 데이터 자체를 포함하지 않습니다.

TarInfo 객체는 *TarFile*의 메서드 *getmember()*, *getmembers()* 및 *gettarinfo()*에 의해 반환됩니다.

Modifying the objects returned by *getmember()* or *getmembers()* will affect all subsequent operations on the archive. For cases where this is unwanted, you can use *copy.copy()* or call the *replace()* method to create a modified copy in one step.

Several attributes can be set to `None` to indicate that a piece of metadata is unused or unknown. Different *TarInfo* methods handle `None` differently:

- The *extract()* or *extractall()* methods will ignore the corresponding metadata, leaving it set to a default.
- *addfile()* will fail.
- *list()* will print a placeholder string.

class tarfile.**TarInfo**(*name=''*)

TarInfo 객체를 만듭니다.

classmethod TarInfo.**frombuf**(*buf*, *encoding*, *errors*)

문자열 버퍼 *buf*에서 *TarInfo* 객체를 만들고 반환합니다.

버퍼가 유효하지 않으면 *HeaderError*를 발생시킵니다.

classmethod TarInfo.**fromtarfile**(*tarfile*)

TarFile 객체 *tarfile*에서 다음 멤버를 읽고 *TarInfo* 객체로 반환합니다.

TarInfo.**tobuf**(*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='surrogateescape'*)

TarInfo 객체에서 문자열 버퍼를 만듭니다. 인자에 대한 정보는 *TarFile* 클래스의 생성자를 참조하십시오.

버전 3.2에서 변경: *errors* 인자의 기본값으로 `'surrogateescape'`를 사용합니다.

`TarInfo` 객체에는 다음과 같은 공개 데이터 어트리뷰트가 있습니다:

TarInfo.**name**: *str*

아카이브 멤버의 이름

TarInfo.**size**: *int*

바이트 단위의 크기.

TarInfo.**mtime**: *int* | *float*

Time of last modification in seconds since the *epoch*, as in *os.stat_result.st_mtime*.

버전 3.12에서 변경: Can be set to `None` for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**mode**: *int*

Permission bits, as for *os.chmod()*.

버전 3.12에서 변경: Can be set to `None` for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**type**

파일 유형. *type*은 일반적으로 다음 상수 중 하나입니다: *REGTYPE*, *AREGTYPE*, *LNKTYPE*, *SYMTYPE*, *DIRTYPE*, *FIFOTYPE*, *CONTTYPE*, *CHRTYPE*, *BLKTYPE*, *GNUTYPE_SPARSE*. *TarInfo* 객체의 유형을 더 편리하게 결정하려면, 아래 `is*()` 메서드를 사용하십시오.

TarInfo.**linkname**: *str*

대상 파일 이름의 이름, *LNKTYPE*과 *SYMTYPE* 유형의 *TarInfo* 객체에만 존재합니다.

For symbolic links (SYMTYPE), the *linkname* is relative to the directory that contains the link. For hard links (LNKTYPE), the *linkname* is relative to the root of the archive.

TarInfo.**uid**: *int*

이 멤버를 처음 저장한 사용자의 사용자 ID.

버전 3.12에서 변경: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**gid**: *int*

이 멤버를 처음 저장한 사용자의 그룹 ID.

버전 3.12에서 변경: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**uname**: *str*

사용자 이름.

버전 3.12에서 변경: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**gname**: *str*

그룹 이름.

버전 3.12에서 변경: Can be set to None for *extract()* and *extractall()*, causing extraction to skip applying this attribute.

TarInfo.**chksum**: *int*

Header checksum.

TarInfo.**devmajor**: *int*

Device major number.

TarInfo.**devminor**: *int*

Device minor number.

TarInfo.**offset**: *int*

The tar header starts here.

TarInfo.**offset_data**: *int*

The file' s data starts here.

TarInfo.**sparse**

Sparse member information.

TarInfo.**pax_headers**: *dict*

연관된 pax 확장 헤더의 키-값 쌍을 포함하는 딕셔너리.

TarInfo.**replace**(*name=..., mtime=..., mode=..., linkname=..., uid=..., gid=..., uname=..., gname=..., deep=True*)

Added in version 3.12.

Return a *new* copy of the TarInfo object with the given attributes changed. For example, to return a TarInfo with the group name set to 'staff', use:

```
new_tarinfo = old_tarinfo.replace(gname='staff')
```

By default, a deep copy is made. If *deep* is false, the copy is shallow, i.e. pax_headers and any custom attributes are shared with the original TarInfo object.

TarInfo 객체는 편리한 조회 메서드도 제공합니다:

`TarInfo.isfile()`

`TarInfo` 객체가 일반 파일이면 `True`를 반환합니다.

`TarInfo.isreg()`

`isfile()`과 같습니다.

`TarInfo.isdir()`

디렉터리이면 `True`를 반환합니다.

`TarInfo.issym()`

심볼릭 링크이면 `True`를 반환합니다.

`TarInfo.islnk()`

하드 링크이면 `True`를 반환합니다.

`TarInfo.ischr()`

문자 장치이면 `True`를 반환합니다.

`TarInfo.isblk()`

블록 장치이면 `True`를 반환합니다.

`TarInfo.isfifo()`

FIFO이면 `True`를 반환합니다.

`TarInfo.isdev()`

문자 장치, 블록 장치 또는 FIFO 중 하나이면 `True`를 반환합니다.

13.8.3 Extraction filters

Added in version 3.12.

The *tar* format is designed to capture all details of a UNIX-like filesystem, which makes it very powerful. Unfortunately, the features make it easy to create tar files that have unintended – and possibly malicious – effects when extracted. For example, extracting a tar file can overwrite arbitrary files in various ways (e.g. by using absolute paths, `..` path components, or symlinks that affect later members).

In most cases, the full functionality is not needed. Therefore, *tarfile* supports extraction filters: a mechanism to limit functionality, and thus mitigate some of the security issues.

> **경고**
>
> None of the available filters blocks *all* dangerous archive features. Never extract archives from untrusted sources without prior inspection. See also *Hints for further verification*.

> **더 보기**
>
> **PEP 706**
> Contains further motivation and rationale behind the design.

The *filter* argument to `TarFile.extract()` or `extractall()` can be:

- the string `'fully_trusted'`: Honor all metadata as specified in the archive. Should be used if the user trusts the archive completely, or implements their own complex verification.
- the string `'tar'`: Honor most *tar*-specific features (i.e. features of UNIX-like filesystems), but block features that are very likely to be surprising or malicious. See `tar_filter()` for details.
- the string `'data'`: Ignore or block most features specific to UNIX-like filesystems. Intended for extracting cross-platform data archives. See `data_filter()` for details.

- None (default): Use *TarFile.extraction_filter*.

  If that is also None (the default), the 'data' filter will be used.

  > 버전 3.14에서 변경: The default filter is set to *data*. Previously, the default was equivalent to *fully_trusted*.

- A callable which will be called for each extracted member with a *TarInfo* describing the member and the destination path to where the archive is extracted (i.e. the same path is used for all members):

```
filter(member: TarInfo, path: str, /) -> TarInfo | None
```

  The callable is called just before each member is extracted, so it can take the current state of the disk into account. It can:

  - return a *TarInfo* object which will be used instead of the metadata in the archive, or
  - return None, in which case the member will be skipped, or
  - raise an exception to abort the operation or skip the member, depending on *errorlevel*. Note that when extraction is aborted, *extractall()* may leave the archive partially extracted. It does not attempt to clean up.

Default named filters

The pre-defined, named filters are available as functions, so they can be reused in custom filters:

tarfile.**fully_trusted_filter**(*member*, *path*)

Return *member* unchanged.

This implements the 'fully_trusted' filter.

tarfile.**tar_filter**(*member*, *path*)

Implements the 'tar' filter.

- Strip leading slashes (/ and *os.sep*) from filenames.
- *Refuse* to extract files with absolute paths (in case the name is absolute even after stripping slashes, e.g. C:/foo on Windows). This raises *AbsolutePathError*.
- *Refuse* to extract files whose absolute path (after following symlinks) would end up outside the destination. This raises *OutsideDestinationError*.
- Clear high mode bits (setuid, setgid, sticky) and group/other write bits (*S_IWGRP* | *S_IWOTH*).

Return the modified TarInfo member.

tarfile.**data_filter**(*member*, *path*)

Implements the 'data' filter. In addition to what tar_filter does:

- Normalize link targets (*TarInfo.linkname*) using *os.path.normpath()*. Note that this removes internal .. components, which may change the meaning of the link if the path in TarInfo.linkname traverses symbolic links.
- *Refuse* to extract links (hard or soft) that link to absolute paths, or ones that link outside the destination.

  This raises *AbsoluteLinkError* or *LinkOutsideDestinationError*.

  Note that such files are refused even on platforms that do not support symbolic links.
- *Refuse* to extract device files (including pipes). This raises *SpecialFileError*.
- For regular files, including hard links:
  - Set the owner read and write permissions (*S_IRUSR* | *S_IWUSR*).
  - Remove the group & other executable permission (*S_IXGRP* | *S_IXOTH*) if the owner doesn' t have it (*S_IXUSR*).
- For other files (directories), set mode to None, so that extraction methods skip applying permission bits.

- Set user and group info (`uid`, `gid`, `uname`, `gname`) to `None`, so that extraction methods skip setting it.

Return the modified `TarInfo` member.

Note that this filter does not block *all* dangerous archive features. See *Hints for further verification* for details.

버전 3.14에서 변경: Link targets are now normalized.

Filter errors

When a filter refuses to extract a file, it will raise an appropriate exception, a subclass of `FilterError`. This will abort the extraction if `TarFile.errorlevel` is 1 or more. With `errorlevel=0` the error will be logged and the member will be skipped, but extraction will continue.

Hints for further verification

Even with `filter='data'`, *tarfile* is not suited for extracting untrusted files without prior inspection. Among other issues, the pre-defined filters do not prevent denial-of-service attacks. Users should do additional checks.

Here is an incomplete list of things to consider:

- Extract to a `new temporary directory` to prevent e.g. exploiting pre-existing links, and to make it easier to clean up after a failed extraction.
- Disallow symbolic links if you do not need the functionality.
- When working with untrusted data, use external (e.g. OS-level) limits on disk, memory and CPU usage.
- Check filenames against an allow-list of characters (to filter out control characters, confusables, foreign path separators, and so on).
- Check that filenames have expected extensions (discouraging files that execute when you "click on them", or extension-less files like Windows special device names).
- Limit the number of extracted files, total size of extracted data, filename length (including symlink length), and size of individual files.
- Check for files that would be shadowed on case-insensitive filesystems.

Also note that:

- Tar files may contain multiple versions of the same file. Later ones are expected to overwrite any earlier ones. This feature is crucial to allow updating tape archives, but can be abused maliciously.
- *tarfile* does not protect against issues with "live" data, e.g. an attacker tinkering with the destination (or source) directory while extraction (or archiving) is in progress.

Supporting older Python versions

Extraction filters were added to Python 3.12, but may be backported to older versions as security updates. To check whether the feature is available, use e.g. `hasattr(tarfile, 'data_filter')` rather than checking the Python version.

The following examples show how to support Python versions with and without the feature. Note that setting `extraction_filter` will affect any subsequent operations.

- Fully trusted archive:

```
my_tarfile.extraction_filter = (lambda member, path: member)
my_tarfile.extractall()
```

- Use the `'data'` filter if available, but revert to Python 3.11 behavior (`'fully_trusted'`) if this feature is not available:

```
my_tarfile.extraction_filter = getattr(tarfile, 'data_filter',
                                       (lambda member, path: member))
my_tarfile.extractall()
```

- Use the `'data'` filter; *fail* if it is not available:

```
my_tarfile.extractall(filter=tarfile.data_filter)
```

or:

```
my_tarfile.extraction_filter = tarfile.data_filter
my_tarfile.extractall()
```

- Use the `'data'` filter; *warn* if it is not available:

```
if hasattr(tarfile, 'data_filter'):
    my_tarfile.extractall(filter='data')
else:
    # 더는 필요하지 않을 때 제거하세요
    warn_the_user('Extracting may be unsafe; consider updating Python')
    my_tarfile.extractall()
```

Stateful extraction filter example

While *tarfile*' s extraction methods take a simple *filter* callable, custom filters may be more complex objects with an internal state. It may be useful to write these as context managers, to be used like this:

```
with StatefulFilter() as filter_func:
    tar.extractall(path, filter=filter_func)
```

Such a filter can be written as, for example:

```
class StatefulFilter:
    def __init__(self):
        self.file_count = 0

    def __enter__(self):
        return self

    def __call__(self, member, path):
        self.file_count += 1
        return member

    def __exit__(self, *exc_info):
        print(f'{self.file_count} files extracted')
```

13.8.4 명령 줄 인터페이스

Added in version 3.4.

`tarfile` 모듈은 tar 아카이브와 상호 작용하기 위한 간단한 명령 줄 인터페이스를 제공합니다.

새 tar 아카이브를 만들려면, `-c` 옵션 뒤에 이름을 지정한 다음 포함해야 하는 파일 이름을 나열하십시오:

```
$ python -m tarfile -c monty.tar  spam.txt eggs.txt
```

디렉터리 전달도 허용됩니다:

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

tar 아카이브를 현재 디렉터리로 추출하려면, `-e` 옵션을 사용하십시오:

```
$ python -m tarfile -e monty.tar
```

디렉터리 이름을 전달하여 tar 아카이브를 다른 디렉터리로 추출할 수도 있습니다:

```
$ python -m tarfile -e monty.tar  other-dir/
```

tar 아카이브에 있는 파일 목록을 보려면, *-l* 옵션을 사용하십시오:

```
$ python -m tarfile -l monty.tar
```

명령 줄 옵션

-l <tarfile>

--list <tarfile>

tarfile에 있는 파일을 나열합니다.

-c <tarfile> <source1> ... <sourceN>

--create <tarfile> <source1> ... <sourceN>

소스 파일에서 tarfile을 만듭니다.

-e <tarfile> [<output_dir>]

--extract <tarfile> [<output_dir>]

*output_dir*이 지정되지 않으면 tarfile을 현재 디렉터리로 추출합니다.

-t <tarfile>

--test <tarfile>

tarfile이 유효한지 테스트합니다.

-v, --verbose

상세한 출력.

--filter <filtername>

Specifies the *filter* for `--extract`. See *Extraction filters* for details. Only string names are accepted (that is, `fully_trusted`, `tar`, and `data`).

13.8.5 예

Reading examples

전체 tar 아카이브를 현재 작업 디렉터리로 추출하는 방법:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall(filter='data')
tar.close()
```

리스트 대신 제너레이터 함수를 사용하여 *TarFile.extractall()*로 tar 아카이브의 부분집합을 추출하는 방법:

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

gzip 압축 tar 아카이브를 읽고 일부 멤버 정보를 표시하는 방법:

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size and is ", end="")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

Writing examples

파일명 리스트로 압축되지 않은 tar 아카이브를 만드는 방법:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

`with` 문을 사용한 같은 예제:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

How to create and write an archive to stdout using `sys.stdout.buffer` in the *fileobj* parameter in `TarFile.add()`:

```
import sys
import tarfile
with tarfile.open("sample.tar.gz", "w|gz", fileobj=sys.stdout.buffer) as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

`TarFile.add()`의 *filter* 매개 변수를 사용하여 아카이브를 만들고 사용자 정보를 재설정하는 방법:

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

13.8.6 지원되는 tar 형식

`tarfile` 모듈로 만들 수 있는 tar 형식은 세 가지입니다:

- POSIX.1-1988 ustar 형식 (`USTAR_FORMAT`). 최대 256자 길이의 파일명과 최대 100자 링크 이름을 지원합니다. 최대 파일 크기는 8 GiB입니다. 이것은 오래되고 제한적이지만 널리 지원되는 형식입니다.

- GNU tar 형식 (`GNU_FORMAT`). 긴 파일명과 링크 이름, 8GiB보다 큰 파일과 스파스(sparse) 파일을 지원합니다. 이것은 GNU/Linux 시스템에서 사실상 표준입니다. `tarfile`은 긴 이름에 대한 GNU tar 확장을 완전히 지원하며 스파스(sparse) 파일 지원은 읽기 전용입니다.
- POSIX.1-2001 pax 형식 (`PAX_FORMAT`). 사실상 제한이 없는 가장 유연한 형식입니다. 긴 파일명과 링크 이름, 큰 파일을 지원하며 경로명을 이식성 있는 방식으로 저장합니다. GNU tar, bsdtar/libarchive 및 star를 포함한 최신 tar 구현은 확장된 *pax* 기능을 완벽하게 지원합니다; 일부 오래되었거나 유지 관리되지 않는 라이브러리는 그렇지 않을 수 있지만, *pax* 아카이브를 마치 범용적으로 지원되는 *ustar* 형식인 것처럼 취급해야 합니다. 새 아카이브의 현재 기본 형식입니다.

  다른 방법으로 저장할 수 없는 정보를 위해 추가 헤더를 사용하여 기존 *ustar* 형식을 확장합니다. pax 헤더에는 두 가지 종류가 있습니다: 확장(extended) 헤더는 후속 파일 헤더에만 영향을 미치며, 전역(global) 헤더는 전체 아카이브에 유효하며 뒤따르는 파일 모두에 영향을 미칩니다. pax 헤더의 모든 데이터는 이식성의 이유로 *UTF-8*로 인코딩됩니다.

읽을 수는 있지만 만들 수 없는 tar 형식의 변형이 더 있습니다:

- 고대 V7 형식. 이것은 일반 파일과 디렉터리 만 저장하는 유닉스 7판(Unix Seventh Edition)에서 온 첫 번째 tar 형식입니다. 이름은 100자 이하여야 합니다. 사용자/그룹 이름 정보는 없습니다. ASCII가 아닌 문자가 있는 필드의 경우 일부 아카이브에서 헤더 체크섬이 잘못 계산되었습니다.
- SunOS tar 확장 형식. 이 형식은 POSIX.1-2001 pax 형식의 변형이지만, 호환되지 않습니다.

13.8.7 유니코드 문제

tar 형식은 원래 파일 시스템 정보 보존에 중점을 두고 테이프 드라이브에서 백업하기 위해 고안되었습니다. 현재 tar 아카이브는 일반적으로 파일 배포와 네트워크를 통한 아카이브 교환에 사용됩니다. 원래 형식(다른 모든 형식의 기초)의 한 가지 문제점은 다른 문자 인코딩을 지원한다는 개념이 없다는 것입니다. 예를 들어, *UTF-8* 시스템에서 만들어진 일반 tar 아카이브는 *ASCII*가 아닌 문자가 포함되면 *Latin-1* 시스템에서 올바르게 읽을 수 없습니다. (파일명, 링크 이름, 사용자/그룹 이름과 같은) 텍스트 메타 데이터가 손상된 것으로 나타납니다. 불행히도, 아카이브의 인코딩을 자동 감지하는 방법은 없습니다. pax 형식은 이 문제를 해결하도록 설계되었습니다. 범용 문자 인코딩 *UTF-8*을 사용하여 ASCII가 아닌 메타 데이터를 저장합니다.

`tarfile`에서 문자 변환의 세부 사항은 `TarFile` 클래스의 *encoding*과 *errors* 키워드 인자에 의해 제어됩니다.

*encoding*은 아카이브의 메타 데이터에 사용할 문자 인코딩을 정의합니다. 기본값은 `sys.getfilesystemencoding()` 이고, 또는 폴 백으로 `'ascii'`입니다. 아카이브를 읽거나 쓰는지에 따라, 메타 데이터를 디코딩하거나 인코딩해야 합니다. *encoding*이 올바르게 설정되지 않으면, 이 변환이 실패할 수 있습니다.

errors 인자는 변환할 수 없는 문자를 처리하는 방법을 정의합니다. 가능한 값은 섹션 에러 처리기에 나열되어 있습니다. 기본 체계는 파이썬은 파일 시스템 호출에도 사용하는 `'surrogateescape'`입니다, 파일명, 명령 줄 인자 및 환경 변수를 참조하십시오.

`PAX_FORMAT` 아카이브(기본값)의 경우, 모든 메타 데이터가 *UTF-8*을 사용하여 저장기 때문에 *encoding*은 일반적으로 필요하지 않습니다. *encoding*은 바이너리 pax 헤더가 디코딩되거나 서로게이트 문자가 있는 문자열이 저장될 때와 같은 드문 경우에만 사용됩니다.

CHAPTER 14

파일 형식

이 장에서 설명하는 모듈들은 마크업 언어가 아니고 전자 메일과 무관한 다양한 파일 형식을 구문 분석합니다.

14.1 csv — CSV 파일 읽기와 쓰기

소스 코드: Lib/csv.py

소위 CSV (Comma Separated Values – 쉼표로 구분된 값) 형식은 스프레드시트와 데이터베이스에 대한 가장 일반적인 가져오기 및 내보내기 형식입니다. CSV 형식은 **RFC 4180**에서 표준화된 방식으로 형식을 기술하기 전에 여러 해 동안 사용되었습니다. 잘 정의된 표준이 없다는 것은 다른 애플리케이션에 의해 생성되고 소비되는 데이터에 미묘한 차이가 존재한다는 것을 의미합니다. 이러한 차이로 인해 여러 소스의 CSV 파일을 처리하는 것이 번거로울 수 있습니다. 그러나 분리 문자와 인용 문자가 다양하기는 해도, 전체 형식은 유사하여 프로그래머에게 데이터 읽기와 쓰기 세부 사항을 숨기면서도 이러한 데이터를 효율적으로 조작할 수 있는 단일 모듈을 작성하는 것이 가능합니다.

`csv` 모듈은 CSV 형식의 표 형식 데이터를 읽고 쓰는 클래스를 구현합니다. 이 모듈은 프로그래머가 Excel에서 사용하는 CSV 형식에 대한 자세한 내용을 알지 못해도, "Excel에서 선호하는 형식으로 이 데이터를 쓰세요" 나 "Excel에서 생성된 이 파일의 데이터를 읽으세요" 라고 말할 수 있도록 합니다. 프로그래머는 다른 응용 프로그램에서 이해할 수 있는 CSV 형식을 기술하거나 자신만의 특수 용도 CSV 형식을 정의할 수 있습니다.

`csv` 모듈의 `reader`와 `writer` 객체는 시퀀스를 읽고 씁니다. 프로그래머는 `DictReader`와 `DictWriter` 클래스를 사용하여 딕셔너리 형식으로 데이터를 읽고 쓸 수 있습니다.

더 보기

PEP 305 - CSV File API
파이썬에 이 모듈의 추가를 제안한 파이썬 개선 제안.

14.1.1 모듈 내용

`csv` 모듈은 다음 함수를 정의합니다:

csv.**reader**(*csvfile*, /, *dialect='excel'*, ***fmtparams*)

지정된 *csvfile*의 줄을 처리하는 판독기(*reader*) 객체를 반환합니다. csvfile은 문자열의 이터러블이어야 하며, 각 문자열은 판독기가 정의한 csv 형식이어야 합니다. csvfile은 보통 파일류 객체이거나 리스트입니다. *csvfile*가 파일 객체이면, `newline=''`로 열렸어야 합니다.[1] 특정 CSV 방언(dialect)에만 적용되는 파라미터 집합을 정의하는 데 사용되는 선택적 *dialect* 매개 변수를 지정할 수 있습니다. `Dialect` 클래스의 서브 클래스의 인스턴스이거나 `list_dialects()` 함수가 반환하는 문자열 중 하나일 수 있습니다. 다른 선택적 *fmtparams* 키워드 인자는 현재 방언의 개별 포매팅 파라미터를 대체 할 수 있습니다. 방언과 포매팅 파라미터에 대한 자세한 내용은 방언과 포매팅 파라미터 절을 참조하십시오.

Each row read from the csv file is returned as a list of strings. No automatic data type conversion is performed unless the `QUOTE_NONNUMERIC` format option is specified (in which case unquoted fields are transformed into floats).

간단한 사용 예:

```
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

csv.**writer**(*csvfile*, /, *dialect='excel'*, ***fmtparams*)

지정된 파일류 객체에 분리된 문자열로 사용자의 데이터를 변환하는 기록기(writer) 객체를 반환합니다. *csvfile*은 `write()` 메서드가 있는 모든 객체일 수 있습니다. *csvfile*이 파일 객체면, `newline=''`으로 열렸어야 합니다[1]. 특정 CSV 방언(dialect)에만 적용되는 파라미터 집합을 정의하는 데 사용되는 선택적 *dialect* 매개 변수를 지정할 수 있습니다. `Dialect` 클래스의 서브 클래스의 인스턴스이거나 `list_dialects()` 함수가 반환하는 문자열 중 하나일 수 있습니다. 다른 선택적 *fmtparams* 키워드 인자는 현재 방언의 개별 포매팅 파라미터를 대체 할 수 있습니다. 방언과 포매팅 파라미터에 대한 자세한 내용은 방언과 포매팅 파라미터 절을 참조하십시오. DB API를 구현하는 모듈과 가능한 한 쉽게 인터페이스 하기 위해, 값 `None`은 빈 문자열로 기록됩니다. 이것은 가역 변환이 아니지만, `cursor.fetch*` 호출에서 반환된 데이터를 전처리하지 않고도, SQL NULL 데이터값을 CSV 파일로 쉽게 덤프할 수 있습니다. 다른 모든 비 문자열 데이터는 기록 전에 `str()`로 문자열화 됩니다.

간단한 사용 예:

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

csv.**register_dialect**(*name*[, *dialect*[, ***fmtparams*]])

*dialect*를 *name*과 연관시킵니다. *name*은 문자열이어야 합니다. 방언은 `Dialect`의 서브 클래스 전달, *fmtparams* 키워드 인자 또는 둘 모두를 사용하여 지정할 수 있는데, 키워드 인자가 dialect의 매개 변수보다 우선합니다. 방언과 포매팅 파라미터에 대한 자세한 내용은 방언과 포매팅 파라미터 절을 참조하십시오.

csv.**unregister_dialect**(*name*)

방언(dialect) 등록소에서 *name*과 관련된 연관된 방언을 삭제합니다. *name*이 등록된 방언 이름이 아니면 `Error`가 발생합니다.

[1] If `newline=''` is not specified, newlines embedded inside quoted fields will not be interpreted correctly, and on platforms that use `\r\n` line endings on write an extra `\r` will be added. It should always be safe to specify `newline=''`, since the csv module does its own (*universal*) newline handling.

csv.**get_dialect**(*name*)

*name*과 연관된 방언을 반환합니다. *name*이 등록된 방언 이름이 아니면 `Error`가 발생합니다. 이 함수는 불변 `Dialect`를 반환합니다.

csv.**list_dialects**()

등록된 모든 방언의 이름을 반환합니다.

csv.**field_size_limit**([*new_limit*])

구문 분석기가 허락하는 현재의 최대 필드 크기를 반환합니다. *new_limit*가 주어지면, 이것이 새로운 한계가 됩니다.

`csv` 모듈은 다음 클래스를 정의합니다:

class csv.**DictReader**(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, *dialect='excel'*, **args*, ***kwds*)

일반 판독기처럼 작동하지만 각 행(row)의 정보를 키가 선택적 *fieldnames* 매개 변수로 지정된 `dict`로 매핑하는 객체를 만듭니다.

fieldnames 매개 변수는 시퀀스입니다. *fieldnames*를 생략하면, 파일 *f*의 첫 번째 행에 있는 값들을 fieldnames로 사용하며 결과에서 생략됩니다. *fieldnames*를 제공하면, 사용되고 첫 번째 행은 결과에 포함됩니다. 필드 이름이 어떻게 결정되는지와 관계없이, 딕셔너리는 원래 순서를 유지합니다.

행에 fieldnames보다 많은 필드가 있으면, 나머지 데이터가 리스트에 저장되고 *restkey*(기본값은 `None`)로 지정된 필드 이름으로 저장됩니다. 비어 있지 않은 행에 fieldnames보다 필드 수가 적다면, 빠진 값은 *restval*(기본값은 `None`)의 값으로 채워집니다.

다른 모든 선택적 또는 키워드 인자는 하부 `reader` 인스턴스에 전달됩니다.

If the argument passed to *fieldnames* is an iterator, it will be coerced to a `list`.

버전 3.6에서 변경: 반환된 행은 이제 `OrderedDict` 형입니다.

버전 3.8에서 변경: 반환된 행은 이제 `dict` 형입니다.

간단한 사용 예:

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
{'first_name': 'John', 'last_name': 'Cleese'}
```

class csv.**DictWriter**(*f*, *fieldnames*, *restval=''*, *extrasaction='raise'*, *dialect='excel'*, **args*, ***kwds*)

일반 기록기처럼 작동하지만 딕셔너리를 출력 행에 매핑하는 객체를 만듭니다. *fieldnames* 매개 변수는 키의 시퀀스인데, `writerow()` 메서드에 전달된 딕셔너리의 값이 *f* 파일에 기록되는 순서를 식별합니다. 선택적 *restval* 매개 변수는 딕셔너리에 *fieldnames*의 키가 빠졌을 때 기록될 값을 지정합니다. `writerow()` 메서드에 전달된 딕셔너리에 *fieldnames*에 없는 키가 포함되어 있으면, 선택적 *extrasaction* 매개 변수가 수행할 작업을 지시합니다. 기본값인 `'raise'`로 설정되면, `ValueError`가 발생합니다. `'ignore'`로 설정하면, 딕셔너리의 추가 값이 무시됩니다. 다른 선택적 또는 키워드 인자는 하부 `writer` 인스턴스에 전달됩니다.

`DictReader` 클래스와 달리 `DictWriter` 클래스의 *fieldnames* 매개 변수는 선택 사항이 아닙니다.

If the argument passed to *fieldnames* is an iterator, it will be coerced to a `list`.

간단한 사용 예:

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

class csv.**Dialect**

The *Dialect* class is a container class whose attributes contain information for how to handle doublequotes, whitespace, delimiters, etc. Due to the lack of a strict CSV specification, different applications produce subtly different CSV data. *Dialect* instances define how *reader* and *writer* instances behave.

All available *Dialect* names are returned by *list_dialects()*, and they can be registered with specific *reader* and *writer* classes through their initializer (`__init__`) functions like this:

```
import csv

with open('students.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, dialect='unix')
```

class csv.**excel**

excel 클래스는 Excel에서 생성한 CSV 파일의 일반적인 속성을 정의합니다. 방언 이름 `'excel'`로 등록됩니다.

class csv.**excel_tab**

excel_tab 클래스는 Excel에서 생성된 TAB 구분 파일의 일반적인 속성을 정의합니다. 방언 이름 `'excel-tab'`으로 등록됩니다.

class csv.**unix_dialect**

unix_dialect 클래스는 유닉스 시스템에서 생성된 CSV 파일의 일반적인 속성을 정의합니다. 즉, `'\n'`을 줄 종결자로 사용하고 모든 필드를 인용 처리합니다. 방언 이름 `'unix'`로 등록됩니다.

Added in version 3.2.

class csv.**Sniffer**

Sniffer 클래스는 CSV 파일의 형식을 추론하는 데 사용됩니다.

Sniffer 클래스는 두 가지 메서드를 제공합니다:

sniff(*sample*, *delimiters=None*)

지정된 *sample*을 분석하고 발견된 파라미터를 반영하는 *Dialect* 서브 클래스를 반환합니다. 선택적인 *delimiters* 매개 변수를 주면, 가능한 유효한 구분 문자를 포함하는 문자열로 해석됩니다.

has_header(*sample*)

sample 텍스트(CSV 형식으로 추정합니다)를 분석하고, 첫 번째 행이 일련의 열 머리글로 보이면 *True*를 반환합니다. 각 열을 검사하면서, 샘플에 헤더가 포함되어 있는지 추정하기 위해 두 가지 주요 기준 중 하나가 고려됩니다:

- the second through n-th rows contain numeric values
- the second through n-th rows contain strings where at least one value' s length differs from that of the putative header of that column.

Twenty rows after the first row are sampled; if more than half of columns + rows meet the criteria, *True* is returned.

> **참고**
>
> This method is a rough heuristic and may produce both false positives and negatives.

Sniffer 사용 예:

```
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... CSV 파일 내용을 여기에서 처리합니다 ...
```

csv 모듈은 다음 상수를 정의합니다:

csv.**QUOTE_ALL**

writer 객체에 모든 필드를 인용 처리하도록 지시합니다.

csv.**QUOTE_MINIMAL**

Instructs *writer* objects to only quote those fields which contain special characters such as *delimiter*, *quotechar*, `'\r'`, `'\n'` or any of the characters in *lineterminator*.

csv.**QUOTE_NONNUMERIC**

writer 객체에 모든 숫자가 아닌 필드를 인용 처리하도록 지시합니다.

Instructs *reader* objects to convert all non-quoted fields to type *float*.

> **참고**
>
> Some numeric types, such as *bool*, *Fraction*, or *IntEnum*, have a string representation that cannot be converted to *float*. They cannot be read in the *QUOTE_NONNUMERIC* and *QUOTE_STRINGS* modes.

csv.**QUOTE_NONE**

Instructs *writer* objects to never quote fields. When the current *delimiter*, *quotechar*, *escapechar*, `'\r'`, `'\n'` or any of the characters in *lineterminator* occurs in output data it is preceded by the current *escapechar* character. If *escapechar* is not set, the writer will raise *Error* if any characters that require escaping are encountered. Set *quotechar* to `None` to prevent its escaping.

reader 객체에게 인용 문자의 특별한 처리를 수행하지 않도록 지시합니다.

csv.**QUOTE_NOTNULL**

Instructs *writer* objects to quote all fields which are not `None`. This is similar to *QUOTE_ALL*, except that if a field value is `None` an empty (unquoted) string is written.

Instructs *reader* objects to interpret an empty (unquoted) field as `None` and to otherwise behave as *QUOTE_ALL*.

Added in version 3.12.

csv.**QUOTE_STRINGS**

Instructs *writer* objects to always place quotes around fields which are strings. This is similar to *QUOTE_NONNUMERIC*, except that if a field value is `None` an empty (unquoted) string is written.

Instructs *reader* objects to interpret an empty (unquoted) string as `None` and to otherwise behave as *QUOTE_NONNUMERIC*.

Added in version 3.12.

csv 모듈은 다음 예외를 정의합니다:

exception csv.**Error**

에러가 감지될 때 모든 함수가 발생시킵니다.

14.1.2 방언과 포매팅 파라미터

입력과 출력 레코드의 형식을 더 쉽게 지정할 수 있도록, 특정 포매팅 파라미터가 함께 방언으로 묶입니다. 방언(dialect)은 CSV 파일의 형식을 설명하는 다양한 어트리뷰트를 포함하는 `Dialect` 클래스의 서브클래스입니다. `reader`나 `writer` 객체를 만들 때, 프로그래머는 문자열이나 `Dialect` 클래스의 서브 클래스를 dialect 매개 변수로 지정할 수 있습니다. *dialect* 매개 변수에 추가하여, 또는 대신에, 프로그래머는 아래에서 `Dialect` 클래스에 대해 정의된 어트리뷰트와 같은 이름을 갖는 개별 포매팅 매개 변수를 지정할 수 있습니다.

방언은 다음 어트리뷰트를 지원합니다:

`Dialect.`**`delimiter`**

필드를 구분하는 데 사용되는 한 문자로 된 문자열. 기본값은 `','`입니다.

`Dialect.`**`doublequote`**

필드 안에 나타나는 *quotechar*의 인스턴스를 인용 처리하는 방법을 제어합니다. `True`일 때, 문자를 두 개로 늘립니다. `False`일 때, *escapechar*를 *quotechar*의 접두어로 사용합니다. 기본값은 `True`입니다.

출력 시, *doublequote*가 `False`이고 아무런 *escapechar*가 설정되지 않았으면, 필드에 *quotechar*가 있으면 `Error`가 발생합니다.

`Dialect.`**`escapechar`**

A one-character string used by the writer to escape characters that require escaping:

- the *delimiter*, the *quotechar*, `'\r'`, `'\n'` and any of the characters in *lineterminator* are escaped if *quoting* is set to `QUOTE_NONE`;
- the *quotechar* is escaped if *doublequote* is `False`;
- the *escapechar* itself.

On reading, the *escapechar* removes any special meaning from the following character. It defaults to `None`, which disables escaping.

버전 3.11에서 변경: An empty *escapechar* is not allowed.

`Dialect.`**`lineterminator`**

`writer`에 의해 생성된 행을 종료하는 데 사용되는 문자열. 기본값은 `'\r\n'`입니다.

> **참고**
>
> `reader`는 `'\r'`이나 `'\n'`을 줄 종료로 인식하도록 하드 코딩되어 있으며, *lineterminator*를 무시합니다. 이 동작은 앞으로 변경될 수 있습니다.

`Dialect.`**`quotechar`**

A one-character string used to quote fields containing special characters, such as the *delimiter* or the *quotechar*, or which contain new-line characters (`'\r'`, `'\n'` or any of the characters in *lineterminator*). It defaults to `'"'`. Can be set to `None` to prevent escaping `'"'` if *quoting* is set to `QUOTE_NONE`.

버전 3.11에서 변경: An empty *quotechar* is not allowed.

`Dialect.`**`quoting`**

Controls when quotes should be generated by the writer and recognised by the reader. It can take on any of the *QUOTE_* constants* and defaults to `QUOTE_MINIMAL` if *quotechar* is not `None`, and `QUOTE_NONE` otherwise.

`Dialect.`**`skipinitialspace`**

When `True`, spaces immediately following the *delimiter* are ignored. The default is `False`. When combining `delimiter=' '` with `skipinitialspace=True`, unquoted empty fields are not allowed.

`Dialect.`**`strict`**

`True`일 때, 잘못된 CSV 입력에서 예외 `Error`를 발생시킵니다. 기본값은 `False`입니다.

14.1.3 판독기 객체

판독기 객체(*DictReader* 인스턴스와 *reader()* 함수에서 반환한 객체)에는 다음과 같은 공용 메서드가 있습니다:

csvreader.**__next__**()

판독기의 이터러블 객체의 다음 행을 현재 *Dialect*에 따라 구문 분석하여 리스트(객체가 *reader()*에서 반환된 경우)나 딕셔너리(*DictReader* 인스턴스인 경우)로 반환합니다. 보통 이것을 next(reader) 처럼 호출합니다.

판독기 객체에는 다음과 같은 공용 어트리뷰트가 있습니다:

csvreader.**dialect**

구문 분석기가 사용 중인 방언의 읽기 전용 설명.

csvreader.**line_num**

소스 이터레이터에서 읽은 줄 수. 레코드가 여러 줄에 걸쳐 있을 수 있으므로, 이것은 반환된 레코드 수와 같지 않습니다.

DictReader 객체에는 다음과 같은 공용 어트리뷰트가 있습니다:

DictReader.**fieldnames**

객체를 만들 때 매개 변수로 전달되지 않았으면, 이 어트리뷰트는 첫 번째 액세스 시나 파일에서 첫 번째 레코드를 읽을 때 초기화됩니다.

14.1.4 기록기 객체

writer 객체(*DictWriter* 인스턴스와 *writer()* 함수에서 반환한 객체)에는 다음과 같은 공용 메서드가 있습니다. *row*는 *writer* 객체의 경우 문자열이나 숫자의 이터러블이어야 하며, *DictWriter* 객체의 경우 fieldnames를 (*str()*을 먼저 통과시킴으로써) 문자열이나 숫자로 매핑하는 딕셔너리이어야 합니다. 복소수는 괄호로 둘러싸여 기록됨에 유의하십시오. 이것은 CSV 파일을 읽는 다른 프로그램에서 문제를 일으킬 수 있습니다 (복소수를 지원한다고 가정할 때).

csvwriter.**writerow**(*row*)

row 매개 변수를 현재 *Dialect*에 따라 포매팅해서, 기록기의 파일 객체에 씁니다. 하부 파일 객체의 *write* 메서드 호출의 반환 값을 반환합니다.

버전 3.5에서 변경: 임의의 이터러블 지원 추가.

csvwriter.**writerows**(*rows*)

rows(위에서 설명한 *row* 객체의 이터러블)에 있는 모든 요소를 현재 방언에 따라 포매팅해서, 기록기의 파일 객체에 씁니다.

기록기 객체에는 다음과 같은 공용 어트리뷰트가 있습니다:

csvwriter.**dialect**

기록기가 사용 중인 방언의 읽기 전용 설명.

DictWriter 객체의 공용 메서드는 다음과 같습니다:

DictWriter.**writeheader**()

(생성자에 지정된 대로) 필드 이름을 담은 행을 현재 방언에 따라 포매팅해서, 기록기의 파일 객체에 씁니다. 내부적으로 사용되는 *csvwriter.writerow()* 호출의 반환 값을 반환합니다.

Added in version 3.2.

버전 3.8에서 변경: *writeheader()*는 이제 내부적으로 사용하는 *csvwriter.writerow()* 메서드에서 반환된 값도 반환합니다.

14.1.5 예제

CSV 파일을 읽는 가장 간단한 예:

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

다른 형식의 파일 읽기:

```
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print(row)
```

대응하는 가장 간단한 쓰기 예는 다음과 같습니다:

```
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

CSV 파일을 읽기로 여는 데 *open()*이 사용되므로, 파일은 기본적으로 시스템 기본 인코딩(*locale.getencoding()*를 참조하세요)을 사용하여 유니코드로 디코딩됩니다. 다른 인코딩을 사용하여 파일을 디코딩하려면 open의 `encoding` 인자를 사용하십시오:

```
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

시스템 기본 인코딩 이외의 다른 것으로 쓸 때도 마찬가지입니다: 출력 파일을 열 때 encoding 인자를 지정하십시오.

새로운 방언 등록하기:

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

판독기의 약간 더 고급 사용 — 에러 잡기와 보고:

```
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print(row)
    except csv.Error as e:
        sys.exit(f'file {filename}, line {reader.line_num}: {e}')
```

또한, 모듈이 문자열 구문 분석을 직접 지원하지는 않지만, 쉽게 수행할 수 있습니다:

```
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```

14.2 configparser — 구성 파일 구문 분석기

소스 코드: Lib/configparser.py

이 모듈은 마이크로소프트 윈도우 INI 파일과 유사한 구조를 제공하는 기본 구성 언어를 구현하는 *ConfigParser* 클래스를 제공합니다. 이를 사용하여 최종 사용자가 쉽게 사용자 정의 할 수 있는 파이썬 프로그램을 작성할 수 있습니다.

> **참고**
>
> 이 라이브러리는 윈도우 레지스트리 확장 버전의 INI 문법에 사용된 값-형 접두사를 해석하거나 기록하지 않습니다.

> **더 보기**
>
> 모듈 ***tomllib***
> TOML is a well-specified format for application configuration files. It is specifically designed to be an improved version of INI.
>
> 모듈 ***shlex***
> 응용 프로그램 구성 파일에도 사용할 수 있는 유닉스 셸과 유사한 미니 언어를 만드는 것에 관한 지원.
>
> 모듈 ***json***
> `json` 모듈은 자바스크립트 문법의 부분 집합을 구현하며, 때로는 구성에 사용되지만 주석을 지원하지 않습니다.

14.2.1 빠른 시작

다음과 같은 매우 기본적인 구성 파일을 봅시다:

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[forge.example]
User = hg

[topsecret.server.example]
Port = 50022
ForwardX11 = no
```

INI 파일의 구조는 다음 섹션에서 설명됩니다. 기본적으로, 파일은 섹션으로 구성되며, 각 섹션에는 값이 있는 키가 포함됩니다. *configparser* 클래스는 이러한 파일을 읽고 쓸 수 있습니다. 프로그래밍 방식으로 위의 구성 파일을 만드는 것으로 시작하겠습니다.

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                      'Compression': 'yes',
...                      'CompressionLevel': '9'}
>>> config['forge.example'] = {}
>>> config['forge.example']['User'] = 'hg'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> config['topsecret.server.example'] = {}
>>> topsecret = config['topsecret.server.example']
>>> topsecret['Port'] = '50022'     # 구문 분석기를 변경합니다
>>> topsecret['ForwardX11'] = 'no'  # 여기도 마찬가지입니다
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...   config.write(configfile)
...
```

보시다시피, 구성 구문 분석기는 딕셔너리처럼 취급할 수 있습니다. 나중에 설명되는 차이점이 있지만, 동작은 딕셔너리에서 기대하는 것과 매우 비슷합니다.

이제 구성 파일을 만들고 저장했으니, 파일을 다시 읽고 담긴 데이터를 탐색합시다.

```
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['forge.example', 'topsecret.server.example']
>>> 'forge.example' in config
True
>>> 'python.org' in config
False
>>> config['forge.example']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.example']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['forge.example']:
...     print(key)
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['forge.example']['ForwardX11']
'yes'
```

위에서 볼 수 있듯이, API는 매우 간단합니다. 유일한 마법은 `DEFAULT` 섹션인데, 다른 모든 섹션에 대한 기본값을 제공합니다[1]. 섹션의 키는 대소 문자를 구분하지 않으며 소문자로 저장됨에 유의하십시오[1].

It is possible to read several configurations into a single *ConfigParser*, where the most recently added configuration has the highest priority. Any conflicting keys are taken from the more recent configuration while the previously existing keys are retained. The example below reads in an `override.ini` file, which will override any conflicting keys from the `example.ini` file.

```
[DEFAULT]
ServerAliveInterval = -1
```

[1] 구성 구문 분석기는 심도 있는 사용자 정의를 허용합니다. 각주 참조로 요약된 동작을 변경하는 데 관심이 있으면 구문 분석기 동작 사용자 정의 섹션을 참조하십시오.

```
>>> config_override = configparser.ConfigParser()
>>> config_override['DEFAULT'] = {'ServerAliveInterval': '-1'}
>>> with open('override.ini', 'w') as configfile:
...     config_override.write(configfile)
...
>>> config_override = configparser.ConfigParser()
>>> config_override.read(['example.ini', 'override.ini'])
['example.ini', 'override.ini']
>>> print(config_override.get('DEFAULT', 'ServerAliveInterval'))
-1
```

This behaviour is equivalent to a `ConfigParser.read()` call with several files passed to the *filenames* parameter.

14.2.2 지원되는 데이터형

구성 구문 분석기는 구성 파일에 있는 값의 데이터형을 추측하지 않고, 항상 내부적으로 문자열로 저장합니다. 이것은 다른 데이터형이 필요하면, 직접 변환해야 함을 뜻합니다:

```
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

이 작업이 매우 흔하기 때문에, 구성 구문 분석기는 정수, 부동 소수점 및 불리언을 처리하는 편리한 게터(getter) 메서드를 제공합니다. 마지막 것이 가장 흥미로운데, `bool('False')` 가 여전히 `True` 이기 때문에 `bool()` 에 값을 전달하는 것만으로는 충분치 않기 때문입니다. 이것이 구성 구문 분석기가 `getboolean()`도 제공하는 이유입니다. 이 메서드는 대소 문자를 구분하지 않으며 `'yes'`/`'no'`, `'on'`/`'off'`, `'true'`/`'false'` 및 `'1'`/`'0'`에서 불리언 값을 인식합니다[Page 650, 1]. 예를 들면:

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['forge.example'].getboolean('ForwardX11')
True
>>> config.getboolean('forge.example', 'Compression')
True
```

`getboolean()` 외에도, 구성 구문 분석기는 동등한 `getint()`와 `getfloat()` 메서드를 제공합니다. 여러분 자신의 변환기를 등록하고 제공된 변환기를 사용자 정의 할 수 있습니다.[Page 650, 1]

14.2.3 대체 값

딕셔너리와 마찬가지로, 섹션의 `get()` 메서드를 사용하여 대체(fallback) 값을 제공할 수 있습니다:

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

기본값이 대체 값보다 우선함에 유의하십시오. 예를 들어, 이 예에서 `'CompressionLevel'` 키는 `'DEFAULT'` 섹션에서만 지정되었습니다. 이것을 `'topsecret.server.example'` 섹션에서 가져오려고 하면 대체 값을 지정하더라도 항상 기본값을 얻습니다:

```
>>> topsecret.get('CompressionLevel', '3')
'9'
```

알아 두어야 할 또 다른 사항은 구문 분석기 수준의 *get()* 메서드가 이전 버전과의 호환성을 위해 유지되는 더 복잡한 사용자 정의 인터페이스를 제공한다는 것입니다. 이 메서드를 사용할 때, `fallback` 키워드 전용 인자를 통해 대체 값을 제공할 수 있습니다:

```
>>> config.get('forge.example', 'monster',
...            fallback='No such things as monsters')
'No such things as monsters'
```

getint(), *getfloat()* 및 *getboolean()* 메서드에 같은 `fallback` 인자를 사용할 수 있습니다. 예를 들면:

```
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

14.2.4 지원되는 INI 파일 구조

구성 파일은 섹션으로 구성되며, 각 섹션은 `[section]` 헤더로 시작하고, 특정 문자열(기본적으로 `=`이나 `:`[Page 650, 1])로 구분된 키/값 항목들이 뒤따릅니다. 기본적으로, 섹션 이름은 대소 문자를 구분하지만 키는 구분하지 않습니다[Page 650, 1]. 선행과 후행 공백은 키와 값에서 제거됩니다. 구문 분석기가 이를 허용하도록 구성되어 있다면 값을 생략할 수 있으며[Page 650, 1], 이 경우 키/값 구분자도 생략될 수 있습니다. 값의 첫 줄보다 깊이 들여쓰기만 하면 값이 여러 줄에 걸쳐있을 수 있습니다. 구문 분석기의 모드에 따라, 빈 줄은 여러 줄 값의 일부로 취급되거나 무시될 수 있습니다.

By default, a valid section name can be any string that does not contain '\n'. To change this, see *ConfigParser.SECTCRE*.

The first section name may be omitted if the parser is configured to allow an unnamed top level section with `allow_unnamed_section=True`. In this case, the keys/values may be retrieved by *UNNAMED_SECTION* as in `config[UNNAMED_SECTION]`.

구성 파일에는 특정 문자(기본적으로 `#`과 `;`[Page 650, 1])를 접두사로 붙인 주석이 포함될 수 있습니다. 주석은 주석이 없다면 빈 줄일 곳에 있을 수 있으며, 들여쓰기 될 수 있습니다.[Page 650, 1]

예를 들면:

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I'm a lumberjack, and I'm okay
    I sleep all night and I work all day

[No Values]
key_without_value
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
empty string value here =

[You can use comments]
# 이것처럼
; 또는 이것처럼

# 기본적으로 빈 줄에만 허용됩니다.
# 인라인 주석은 사용자가 구분 문자를 값의 일부로
# 사용하지 못하게 하므로 해로울 수 있습니다.
# 그러해도, 이것을 사용자 정의할 수 있습니다.

    [Sections Can Be Indented]
        can_values_be_as_well = True
        does_that_mean_anything_special = False
        purpose = formatting for readability
        multiline_values = are
            handled just fine as
            long as they are indented
            deeper than the first line
            of a value
        # 주석도 들여 쓸 수 있다고 언급했나요?
```

14.2.5 Unnamed Sections

The name of the first section (or unique) may be omitted and values retrieved by the *UNNAMED_SECTION* attribute.

```
>>> config = """
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> unnamed = configparser.ConfigParser(allow_unnamed_section=True)
>>> unnamed.read_string(config)
>>> unnamed.get(configparser.UNNAMED_SECTION, 'option')
'value'
```

14.2.6 값의 보간

핵심 기능 위에, *ConfigParser*는 보간(interpolation)을 지원합니다. 이는 `get()` 호출에서 값을 반환하기 전에 값을 전처리할 수 있음을 의미합니다.

class configparser.**BasicInterpolation**

*ConfigParser*에서 사용되는 기본 구현. 같은 섹션의 다른 값이나 특수한 기본 섹션의 값을 참조하는 포맷 문자열을 포함하는 값을 사용할 수 있도록 합니다[Page 650, 1]. 초기화할 때 추가 기본값을 제공할 수 있습니다.

예를 들면:

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures

[Escape]
# %% 를 사용하여 % 기호를 이스케이프 합니다 (% 는 이스케이프 해야 하는 유일한 문자입
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
니다):
gain: 80%%
```

위의 예에서, *interpolation*이 `BasicInterpolation()` 으로 설정된 *ConfigParser*는 `%(home_dir)s`를 `home_dir`의 값(이 경우 `/Users`)으로 해석합니다. 결과적으로 `%(my_dir)s`는 `/Users/lumberjack`으로 해석됩니다. 모든 보간은 요청 시 수행되므로 참조 체인에 사용된 키를 구성 파일에서 특정 순서로 지정할 필요는 없습니다.

`interpolation`을 `None`으로 설정하면, 구문 분석기는 `my_pictures`의 값을 `%(my_dir)s/Pictures`로, `my_dir`의 값을 `%(home_dir)s/lumberjack`으로 반환합니다.

class configparser.**ExtendedInterpolation**

예를 들어 `zc.buildout`에서 사용되는 고급 문법을 구현하는 보간 대체 처리기. 확장 보간은 `${section:option}`을 사용하여 외부 섹션의 값을 나타냅니다. 보간은 여러 수준으로 확장될 수 있습니다. 편의상, `section:` 부분을 생략하면, 보간은 현재 섹션(그리고 가능하면 특수 섹션의 기본값)으로 기본 설정됩니다.

예를 들어, 기본 보간으로 위에서 지정한 구성은, 확장 보간으로는 다음과 같습니다:

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures

[Escape]
# $$ 를 사용하여 $ 기호를 이스케이프 합니다 ($ 는 이스케이프 해야 하는 유일한 문자입
니다):
cost: $$80
```

다른 섹션의 값도 가져올 수 있습니다:

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
nickname: Two Sheds
last_name: Jackson
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python/Versions/${Frameworks:Python}
```

14.2.7 매핑 프로토콜 액세스

Added in version 3.2.

매핑 프로토콜 액세스는 사용자 정의 객체를 딕셔너리처럼 사용하는 기능의 일반적인 이름입니다. *configparser*의 경우, 매핑 인터페이스 구현은 `parser['section']['option']` 표기법을 사용합니다.

`parser['section']`은 특히 구문 분석기의 섹션 데이터에 대한 프락시를 반환합니다. 이는 값은 복사되지 않지만 필요할 때 원래 구문 분석기에서 가져옴을 뜻합니다. 더욱 중요한 것은 섹션 프락시에서 값이 변경되면, 실제로 원래 구문 분석기에서 변경된다는 것입니다.

configparser 객체는 가능한 한 실제 딕셔너리에 가깝게 동작합니다. 매핑 인터페이스가 완전하며 *MutableMapping* ABC를 준수합니다. 그러나, 고려해야 하는 몇 가지 차이점이 있습니다:

- 기본적으로, 섹션의 모든 키는 대소 문자를 구분하지 않고 액세스 할 수 있습니다[Page 650, 1]. 예를 들어 `for option in parser["section"]`는 optionxform 변환된 옵션 키 이름만 산출합니다. 이것은 기본적으로 소문자 키를 의미합니다. 동시에, 키 `'a'`를 보유하는 섹션의 경우, 두 표현식 모두 `True`를 반환합니다:

```
"a" in parser["section"]
"A" in parser["section"]
```

- 모든 섹션에는 `DEFAULTSECT` 값도 포함되어 있는데, 섹션에 대한 `.clear()`가 섹션을 비어 보이게 만들 수 없다는 뜻입니다. 이것은 섹션에서 기본값을 삭제할 수 없기 때문입니다 (기술적으로는 기본값이 거기 없기 때문입니다). 섹션에서 재정의되었으면, 삭제하면 기본값이 다시 보입니다. 기본값을 삭제하려고 하면 *KeyError*가 발생합니다.
- `DEFAULTSECT`는 구문 분석기에서 제거할 수 없습니다:
  - 삭제하려고 하면 *ValueError*가 발생합니다,
  - `parser.clear()`는 이것을 그대로 남겨둡니다,
  - `parser.popitem()`은 이것을 절대 반환하지 않습니다.
- `parser.get(section, option, **kwargs)` - 두 번째 인자는 대체 값이 **아닙니다**. 그러나 섹션 수준 `get()` 메서드는 매핑 프로토콜과 클래식 configparser API와 모두 호환됨에 유의하십시오.
- `parser.items()`는 매핑 프로토콜과 호환됩니다 (DEFAULTSECT를 포함하여 *section_name*, *section_proxy* 쌍의 리스트를 반환합니다). 그러나, 이 메서드는 인자와 함께 호출 할 수도 있습니다: `parser.items(section, raw, vars)`. 후자의 호출은 지정된 `section`에 대한 *option*, *value* 쌍의 리스트를 반환하며, 모든 보간이 확장됩니다 (`raw=True`가 제공되지 않는 한).

매핑 프로토콜은 기존 레거시 API 위에 구현되므로 원래 인터페이스를 재정의하는 서브 클래스에서도 여전히 매핑이 작동해야 합니다.

14.2.8 구문 분석기 동작 사용자 정의

INI 형식을 사용하는 응용 프로그램만큼이나 많은 INI 형식 변형이 있습니다. *configparser*는 사용 가능한 가장 큰 INI 스타일 집합을 지원하기 위해 먼 길을 갔습니다. 기본 기능은 주로 역사적 배경에 의해 결정되며 일부 기능을 사용자 정의하고 싶을 가능성이 큽니다.

특정 구성 구문 분석기의 작동 방식을 변경하는 가장 흔한 방법은 `__init__()` 옵션을 사용하는 것입니다:

- *defaults*, 기본값: `None`

  이 옵션은 처음에 `DEFAULT` 섹션에 배치될 키-값 쌍의 딕셔너리를 받아들입니다. 이것은 지정하지 않으면 설명된 기본값과 같은 값이 되는 간결한 구성 파일을 지원하는 우아한 방법입니다.

  힌트: 특정 섹션에 대한 기본값을 지정하려면, 실제 파일을 읽기 전에 *read_dict()*를 사용하십시오.

- *dict_type*, 기본값: *dict*

  이 옵션은 매핑 프로토콜의 작동 방식과 기록된 구성 파일의 꼴에 큰 영향을 미칩니다. 표준 딕셔너리를 사용하면, 모든 섹션이 구문 분석기에 추가된 순서대로 저장됩니다. 섹션 내의 옵션도 마찬가지입니다.

  대체 딕셔너리 형을 사용하여 예를 들어 다시 쓸 때 섹션과 옵션을 정렬할 수 있습니다.

  참고: 단일 연산에서 키-값 쌍의 집합을 추가하는 방법이 있습니다. 이러한 연산에서 일반 딕셔너리를 사용하면 키 순서가 유지됩니다. 예를 들면:

```
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value1',
...                                'key2': 'value2',
...                                'key3': 'value3'},
...                   'section2': {'keyA': 'valueA',
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...                                 'keyB': 'valueB',
...                                 'keyC': 'valueC'},
...                   'section3': {'foo': 'x',
...                                'bar': 'y',
...                                'baz': 'z'}
... })
>>> parser.sections()
['section1', 'section2', 'section3']
>>> [option for option in parser['section3']]
['foo', 'bar', 'baz']
```

- *allow_no_value*, 기본값: `False`

  일부 구성 파일에는 값이 없는 설정이 포함되어 있지만, 그 외에는 `configparser` 에서 지원하는 구문을 준수하는 것으로 알려져 있습니다. 생성자에 대한 *allow_no_value* 매개 변수를 사용하여 이러한 값을 받아들여야 함을 표시할 수 있습니다:

```
>>> import configparser

>>> sample_config = """
... [mysqld]
...   user = mysql
...   pid-file = /var/run/mysqld/mysqld.pid
...   skip-external-locking
...   old_passwords = 1
...   skip-bdb
...   # we don't need ACID today
...   skip-innodb
... """
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)

>>> # 값이 있는 설정은 이전처럼 처리됩니다:
>>> config["mysqld"]["user"]
'mysql'

>>> # 값이 없는 설정은 None을 제공합니다:
>>> config["mysqld"]["skip-bdb"]

>>> # 지정되지 않은 설정은 여전히 에러를 발생시킵니다:
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
  ...
KeyError: 'does-not-exist'
```

- *delimiters*, 기본값: `('=', ':')`

  구분자(delimiters)는 섹션 내의 값에서 키를 구분하는 부분 문자열입니다. 줄에서 처음 나타나는 구분하는 부분 문자열을 구분자로 간주합니다. 이는 값에 (하지만 키는 아닙니다) 구분자가 포함될 수 있음을 의미합니다.

  `ConfigParser.write()` 에 대한 *space_around_delimiters* 인자도 참조하십시오.

- *comment_prefixes*, 기본값: `('#', ';')`

- *inline_comment_prefixes*, 기본값 : `None`

  주석 접두사는 구성 파일 내에서 유효한 주석의 시작을 나타내는 문자열입니다. *comment_prefixes* 는 주석이 없으면 빈 줄일 때만 (선택적으로 들여쓰기 됩니다) 사용되는 반면 *inline_comment_prefixes*

는 모든 유효한 값 (예를 들어 섹션 이름, 옵션 및 빈 줄 역시) 뒤에 사용될 수 있습니다. 기본적으로 인라인 주석은 비활성화되어 있으며 `'#'`과 `';'`이 전체 줄 주석의 접두사로 사용됩니다.

버전 3.2에서 변경: *configparser*의 이전 버전에서는 동작이 `comment_prefixes=('#',';')`와 `inline_comment_prefixes=(';',)`와 일치했습니다.

구성 구문 분석기는 주석 접두사 이스케이프를 지원하지 않아서 *inline_comment_prefixes*를 사용하면 사용자가 주석 접두사로 사용되는 문자로 옵션값을 지정하지 못할 수 있습니다. 확실하지 않으면, *inline_comment_prefixes*를 설정하지 마십시오. 어떤 상황에서든, 여러 줄 값에서 줄의 시작 부분에 주석 접두사 문자를 저장하는 유일한 방법은 접두사를 보간하는 것입니다, 예를 들어:

```
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
... shebang =
...   ${hash}!/usr/bin/env python
...   ${hash} -*- coding: utf-8 -*-
...
... extensions =
...   enabled_extension
...   another_extension
...   #disabled_by_comment
...   yet_another_extension
...
... interpolation not necessary = if # is not at line start
... even in multiline values = line #1
...   line #2
...   line #3
... """)
>>> print(parser['hashes']['shebang'])

#!/usr/bin/env python
# -*- coding: utf-8 -*-
>>> print(parser['hashes']['extensions'])

enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3
```

- *strict*, 기본값: `True`

  `True`로 설정하면, 구문 분석기는 단일 소스에서 읽는 (*read_file()*, *read_string()* 또는 *read_dict()*를 사용해서) 동안 섹션이나 옵션 중복을 허용하지 않습니다. 새로운 응용 프로그램에서는 엄격한(strict) 구문 분석기를 사용하는 것이 좋습니다.

  버전 3.2에서 변경: *configparser*의 이전 버전에서는 동작이 `strict=False`와 일치했습니다.

- *empty_lines_in_values*, 기본값: `True`

  구성 구문 분석기에서는, 값을 담는 키보다 많이 들여쓰기만 하면 값이 여러 줄에 걸쳐있을 수 있

습니다. 기본적으로 구문 분석기는 빈 줄도 값의 일부가 되도록 합니다. 동시에, 가독성을 높이기 위해 키를 임의로 들여 쓸 수 있습니다. 결과적으로, 구성 파일이 커지고 복잡해지면, 사용자가 파일 구조를 쉽게 놓칠 수 있습니다. 예를 들어 봅시다:

```
[Section]
key = multiline
  value with a gotcha

 this = is still a part of the multiline value of 'key'
```

이것은 사용자가 가변 폭 글꼴을 사용하여 파일을 편집하고 있다면, 보는 데 특히 문제가 될 수 있습니다. 따라서 응용 프로그램에 빈 줄이 있는 값이 필요하지 않으면, 허용하지 않는 것이 좋습니다. 이렇게 하면 빈 줄이 매번 키를 분리합니다. 위의 예에서는, `key`와 `this`의 두 키를 생성합니다.

- *default_section*, 기본값: `configparser.DEFAULTSECT` (즉: `"DEFAULT"`)

  다른 섹션이나 보간 목적으로 기본값의 특수한 섹션을 허용하는 규칙은 이 라이브러리의 강력한 개념으로, 사용자가 복잡한 선언적 구성을 만들 수 있도록 합니다. 이 섹션은 일반적으로 `"DEFAULT"` 라고 하지만 다른 유효한 섹션 이름을 가리키도록 사용자 정의할 수 있습니다. 몇 가지 흔한 값은 이렇습니다: `"general"` 이나 `"common"`. 제공된 이름은 모든 소스에서 읽을 때 기본값 섹션을 인식하는 데 사용되며 구성을 파일에 다시 쓸 때 사용됩니다. 현재 값은 `parser_instance.default_section` 어트리뷰트를 사용하여 꺼낼 수 있으며 실행 시간에 수정될 수 있습니다 (즉 파일을 한 형식에서 다른 형식으로 변환하기 위해).

- *interpolation*, 기본값: `configparser.BasicInterpolation`

  interpolation 인자를 통해 사용자 정의 처리기를 제공하여 보간 동작을 사용자 정의할 수 있습니다. `None`은 보간을 완전히 끄는 데 사용할 수 있으며, `ExtendedInterpolation()` 은 `zc.buildout`에서 영감을 얻은 고급 변형을 제공합니다. 이 주제에 관한 자세한 내용은 전용 설명서 섹션에 있습니다. *`RawConfigParser`* 의 기본값은 `None` 입니다.

- *converters*, 기본값: 설정되지 않음

  구성 구문 분석기는 형 변환을 수행하는 옵션값 게터를 제공합니다. 기본적으로 *`getint()`*, *`getfloat()`* 및 *`getboolean()`*가 구현됩니다. 다른 게터가 바람직하다면, 사용자는 그것들을 서브 클래스에 정의하거나 각 키가 변환기의 이름이고 각 값이 이 변환을 구현하는 콜러블인 딕셔너리를 전달할 수 있습니다. 예를 들어, `{'decimal': decimal.Decimal}`을 전달하면 구문 분석기 객체와 모든 섹션 프락시 모두에 `getdecimal()` 이 추가됩니다. 즉, `parser_instance.getdecimal('section', 'key', fallback=0)` 과 `parser_instance['section'].getdecimal('key', 0)` 을 모두 쓸 수 있습니다.

  변환기가 구문 분석기의 상태에 액세스해야 하면, 구성 구문 분석기 서브 클래스에서 메서드로 구현될 수 있습니다. 이 메서드의 이름이 `get`으로 시작하면, 모든 섹션 프락시에서 dict 호환 형식으로 사용할 수 있습니다 (위의 `getdecimal()` 예를 참조하십시오).

이러한 구문 분석기 어트리뷰트의 기본값을 재정의하여 더 고급 사용자 정의를 수행할 수 있습니다. 기본값은 클래스에서 정의되므로, 서브 클래스나 어트리뷰트 대입으로 재정의할 수 있습니다.

`ConfigParser.`**`BOOLEAN_STATES`**

*`getboolean()`*을 사용할 때 기본적으로, 구성 구문 분석기는 다음 값들을 `True`로 간주하고: `'1'`, `'yes'`, `'true'`, `'on'` 다음 값들을 `False`로 간주합니다: `'0'`, `'no'`, `'false'`, `'off'`. 문자열과 불리언 결과를 사용자 정의 딕셔너리로 지정하여 이를 재정의할 수 있습니다. 예를 들면:

```
>>> custom = configparser.ConfigParser()
>>> custom['section1'] = {'funky': 'nope'}
>>> custom['section1'].getboolean('funky')
Traceback (most recent call last):
...
ValueError: Not a boolean: nope
>>> custom.BOOLEAN_STATES = {'sure': True, 'nope': False}
>>> custom['section1'].getboolean('funky')
False
```

다른 일반적인 불리언 쌍에는 accept/reject 나 enabled/disabled 가 포함됩니다.

ConfigParser.**optionxform**(*option*)

이 메서드는 모든 읽기, get 또는 set 연산에서 옵션 이름을 변환합니다. 기본값은 이름을 소문자로 변환합니다. 이는 또한 구성 파일을 기록할 때 모든 키가 소문자가 됨을 의미합니다. 이것이 부적절하다면 이 메서드를 재정의하십시오. 예를 들면:

```
>>> config = """
... [Section1]
... Key = Value
...
... [Section2]
... AnotherKey = Value
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']
```

참고

optionxform 함수는 옵션 이름을 규범적 형식으로 변환합니다. 이 함수는 멱등적(idempotent) 함수여야 합니다: 이름이 이미 규범적 형식이면, 변경되지 않은 상태로 반환해야 합니다.

ConfigParser.**SECTCRE**

섹션 헤더를 구문 분석하는 데 사용되는 컴파일된 정규식. 기본값은 [section] 을 이름 "section" 과 일치시킵니다. 공백은 섹션 이름의 일부로 간주해서, [larch] 는 이름이 " larch " 인 섹션으로 읽힙니다. 이것이 부적절하다면 이 어트리뷰트를 재정의하십시오. 예를 들면:

```
>>> import re
>>> config = """
... [Section 1]
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', '  Section 2  ']
>>> custom = configparser.ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^]]+?) *\]")
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']
```

> **참고**
>
> ConfigParser 객체는 옵션 줄을 인식하기 위해 `OPTCRE` 어트리뷰트도 사용하지만, 생성자 옵션 *allow_no_value*와 *delimiters*를 방해하기 때문에 재정의하지 않는 것이 좋습니다.

14.2.9 레거시 API 예제

주로 이전 버전과의 호환성 문제로 인해, *configparser*는 명시적인 `get`/`set` 메서드로 레거시 API도 제공합니다. 아래에 설명된 메서드에 대한 유효한 사용 사례가 있지만, 새 프로젝트에는 매핑 프로토콜 액세스가 선호됩니다. 레거시 API는 때때로 더 고급이고, 저수준(low-level)이며 완전히 반 직관적입니다.

구성 파일에 쓰는 예:

```
import configparser

config = configparser.RawConfigParser()

# RawConfigParser 의 set 함수를 사용하면, 문자열이 아닌 값을
# 키에 내부적으로 대입할 수 있지만, 파일에 쓰려고 할 때나
# 비 원시 모드로 가져올 때 에러가 발생함에 유의하십시오.
# 매핑 프로토콜이나 ConfigParser의 set()을 사용하여
# 값을 설정하면 이러한 대입이 허용되지 않습니다.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# 구성 파일을 'example.cfg'에 기록하기
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

구성 파일을 다시 읽는 예:

```
import configparser

config = configparser.RawConfigParser()
config.read('example.cfg')

# 값이 float가 아니면 getfloat()는 예외를 발생시킵니다
# getint()와 getboolean() 또한 해당 유형에 대해 이렇게 합니다
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# 다음 출력은 '%(bar)s' 나 '%(baz)s' 를 보간하지 않음에 유의하십시오.
# RawConfigParser() 를 사용하고 있기 때문입니다.
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))
```

보간을 얻으려면, *ConfigParser*를 사용하십시오:

```
import configparser

cfg = configparser.ConfigParser()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
cfg.read('example.cfg')

# 단일 get 연산에서 보간을 사용하지 않으려면 get()의 선택적 *raw* 인자를
# True로 설정하십시오.
print(cfg.get('Section1', 'foo', raw=False))  # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))   # -> "%(bar)s is %(baz)s!"

# 선택적 *vars* 인자는 보간에서 우선하는 멤버가
# 포함된 dict입니다.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# 선택적 *fallback* 인자를 사용하여 대체 값을 제공할 수 있습니다
print(cfg.get('Section1', 'foo'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'foo', fallback='Monty is not.'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'monster', fallback='No such things as monsters.'))
      # -> "No such things as monsters."

# print(cfg.get('Section1', 'monster')) 만은 NoOptionError 를 발생시킵니다
# 하지만 이렇게 사용할 수도 있습니다:

print(cfg.get('Section1', 'monster', fallback=None))
      # -> None
```

기본값은 두 가지 유형의 ConfigParser 모두에서 사용할 수 있습니다. 사용된 옵션이 다른 곳에 정의되어 있지 않으면 보간에 사용됩니다.

```
import configparser

# 'bar'와 'baz'의 기본값이 각각 'Life'와 'hard'인 새 인스턴스
config = configparser.ConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print(config.get('Section1', 'foo'))     # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))     # -> "Life is hard!"
```

14.2.10 ConfigParser 객체

class configparser.**ConfigParser**(*defaults=None, dict_type=dict, allow_no_value=False, *, delimiters=('=', ':'), comment_prefixes=('#', ';'), inline_comment_prefixes=None, strict=True, empty_lines_in_values=True, default_section=configparser.DEFAULTSECT, interpolation=BasicInterpolation(), converters={}, allow_unnamed_section=False*)

메인 구성 구문 분석기. *defaults*가 주어지면, 내장 기본값의 딕셔너리로 초기화됩니다. *dict_type*이 제공되면, 섹션 리스트, 섹션 내의 옵션 및 기본값에 대한 딕셔너리 객체를 만드는 데 사용됩니다.

*delimiters*가 주어지면, 키를 값과 나누는 부분 문자열 집합으로 사용됩니다. *comment_prefixes*가 주어지면, 주석이 없다면 빈 줄일 곳에서 주석을 시작하는 접두사의 부분 문자열 집합으로 사용됩니다.

주석은 들여 쓸 수 있습니다. *inline_comment_prefixes*가 주어지면, 비어 있지 않은 줄에서 주석을 시작하는 접두사의 부분 문자열 집합으로 사용됩니다.

*strict*가 `True`(기본값)이면, 구문 분석기는 단일 소스(파일, 문자열 또는 딕셔너리)에서 읽는 동안 섹션이나 옵션의 중복을 허용하지 않고, `DuplicateSectionError` 나 `DuplicateOptionError`를 발생시킵니다. *empty_lines_in_values*가 `False`이면 (기본값: `True`), 각 빈 줄은 옵션의 끝을 나타냅니다. 그렇지 않으면, 여러 줄 옵션의 내부 빈 줄이 값의 일부로 유지됩니다. *allow_no_value*가 `True`이면 (기본값: `False`), 값이 없는 옵션이 허용됩니다; 이들에 대해 저장되는 값은 `None`이며 후행 구분자 없이 직렬화됩니다.

*default_section*이 주어지면, 다른 섹션과 보간 목적의 기본값을 담는 특수한 섹션의 이름을 지정합니다 (보통 `"DEFAULT"`라는 이름). 이 값은 `default_section` 인스턴스 어트리뷰트를 사용하여 실행 시간에 꺼내고 변경할 수 있습니다. 이는 이미 구문 분석된 구성 파일을 재평가하지 않지만, 구문 분석된 설정을 새 구성 파일에 기록할 때 사용됩니다.

interpolation 인자를 통해 사용자 정의 처리기를 제공하여 보간 동작을 사용자 정의할 수 있습니다. `None`은 보간을 완전히 끄는 데 사용할 수 있으며, `ExtendedInterpolation()` 은 `zc.buildout`에서 영감을 얻은 고급 변형을 제공합니다. 이 주제에 관한 자세한 내용은 전용 설명서 섹션에 있습니다.

보간에 사용된 모든 옵션 이름은 다른 옵션 이름 참조와 마찬가지로 `optionxform()` 메서드를 통해 전달됩니다. 예를 들어, (옵션 이름을 소문자로 변환하는) `optionxform()`의 기본 구현을 사용하면, 값 `foo %(bar)s`와 `foo %(BAR)s`가 동등합니다.

*converters*가 주어지면, 각 키는 형 변환기의 이름을 나타내고 각 값은 문자열에서 원하는 데이터형으로의 변환을 구현하는 콜러블인 딕셔너리이어야 합니다. 모든 변환기는 구문 분석기 객체와 섹션 프락시에서 자신만의 해당 `get*()` 메서드를 갖습니다.

When *allow_unnamed_section* is `True` (default: `False`), the first section name can be omitted. See the "*Unnamed Sections*" *section*.

It is possible to read several configurations into a single `ConfigParser`, where the most recently added configuration has the highest priority. Any conflicting keys are taken from the more recent configuration while the previously existing keys are retained. The example below reads in an `override.ini` file, which will override any conflicting keys from the `example.ini` file.

```
[DEFAULT]
ServerAliveInterval = -1
```

```
>>> config_override = configparser.ConfigParser()
>>> config_override['DEFAULT'] = {'ServerAliveInterval': '-1'}
>>> with open('override.ini', 'w') as configfile:
...     config_override.write(configfile)
...
>>> config_override = configparser.ConfigParser()
>>> config_override.read(['example.ini', 'override.ini'])
['example.ini', 'override.ini']
>>> print(config_override.get('DEFAULT', 'ServerAliveInterval'))
-1
```

버전 3.1에서 변경: 기본 *dict_type*은 `collections.OrderedDict`입니다.

버전 3.2에서 변경: *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* 및 *interpolation*이 추가되었습니다.

버전 3.5에서 변경: *converters* 인자가 추가되었습니다.

버전 3.7에서 변경: *defaults* 인자는 `read_dict()`로 읽혀, 구문 분석기 전체에 일관된 동작을 제공합니다: 문자열이 아닌 키와 값은 묵시적으로 문자열로 변환됩니다.

버전 3.8에서 변경: 이제 삽입 순서를 유지하므로, 기본 *dict_type*은 `dict`입니다.

버전 3.13에서 변경: Raise a `MultilineContinuationError` when *allow_no_value* is `True`, and a key without a value is continued with an indented line.

버전 3.13에서 변경: *allow_unnamed_section* 인자가 추가되었습니다.

defaults()

인스턴스 전체 기본값을 포함하는 딕셔너리를 반환합니다.

sections()

사용 가능한 섹션 리스트를 반환합니다; 기본값 섹션은 리스트에 포함되지 않습니다.

add_section(*section*)

*section*이라는 섹션을 인스턴스에 추가합니다. 주어진 이름의 섹션이 이미 존재하면, `DuplicateSectionError` 가 발생합니다. 기본값 섹션 이름이 전달되면, `ValueError`가 발생합니다. 섹션의 이름은 문자열이어야 합니다; 그렇지 않으면, `TypeError`가 발생합니다.

버전 3.2에서 변경: 문자열이 아닌 섹션 이름은 `TypeError`를 발생시킵니다.

has_section(*section*)

이름 지정된 *section*이 구성에 있는지를 나타냅니다. 기본값 섹션은 인정되지 않습니다.

options(*section*)

지정된 *section*에서 사용 가능한 옵션 리스트를 반환합니다.

has_option(*section*, *option*)

주어진 *section*이 존재하고, 주어진 *option*을 포함하면, `True`를 반환합니다; 그렇지 않으면 `False`를 반환합니다. 지정된 *section*이 `None`이거나 빈 문자열이면, DEFAULT로 가정합니다.

read(*filenames*, *encoding=None*)

파일명들의 이터러블을 읽고 구문 분석하여, 성공적으로 구문 분석된 파일명의 리스트를 반환합니다.

*filenames*가 문자열, `bytes` 객체 또는 경로류 객체이면 단일 파일명으로 처리됩니다. *filenames*에서 이름 지정된 파일을 열 수 없으면, 해당 파일은 무시됩니다. 이는 잠재적인 구성 파일 위치(예를 들어, 현재 디렉터리, 사용자의 홈 디렉터리 및 일부 시스템 전체 디렉터리)의 이터러블을 지정할 수 있도록 설계되었으며, 이터러블에 있는 존재하는 모든 구성 파일을 읽습니다.

제공된 이름의 파일이 아무것도 없으면, `ConfigParser` 인스턴스는 빈 데이터 집합을 포함합니다. 파일에서 초깃값을 로드해야 하는 응용 프로그램은 선택적 파일에 대해 `read()`를 호출하기 전에 `read_file()`을 사용하여 필수 파일을 로드해야 합니다:

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

버전 3.2에서 변경: *encoding* 매개 변수를 추가했습니다. 이전에는, `open()`의 기본 인코딩을 사용하여 모든 파일을 읽었습니다.

버전 3.6.1에서 변경: *filenames* 매개 변수는 경로류 객체를 받아들입니다.

버전 3.7에서 변경: *filenames* 매개 변수는 `bytes` 객체를 받아들입니다.

read_file(*f*, *source=None*)

*f*에서 구성 데이터를 읽고 구문 분석합니다. *f*는 유니코드 문자열을 산출하는 이터러블 이어야 합니다 (예를 들어 텍스트 모드로 열린 파일).

선택적 인자 *source*는 읽을 파일의 이름을 지정합니다. 지정하지 않고 *f*에 `name` 어트리뷰트가 있으면, 이것이 *source*로 사용됩니다; 기본값은 `'<???>'`입니다.

Added in version 3.2: `readfp()`를 대체합니다.

read_string(*string*, *source='<string>'*)

문자열에서 구성 데이터를 구문 분석합니다.

선택적 인자 *source*는 전달된 string의 문맥 특정 이름을 지정합니다. 지정하지 않으면, `'<string>'`이 사용됩니다. 일반적으로 파일 시스템 경로나 URL이어야 합니다.

Added in version 3.2.

read_dict(*dictionary*, *source='<dict>'*)

딕셔너리와 같은 `items()` 메서드를 제공하는 임의의 객체에서 구성을 로드합니다. 키는 섹션 이름이며, 값은 섹션에 있어야 하는 키와 값이 들어 있는 딕셔너리입니다. 사용된 딕셔너리 형이 순서를 유지하면, 섹션과 해당 키가 순서대로 추가됩니다. 값은 자동으로 문자열로 변환됩니다.

선택적 인자 *source*는 전달된 dictionary의 문맥 특정 이름을 지정합니다. 지정하지 않으면, `<dict>`가 사용됩니다.

이 메서드를 사용하면 구문 분석 기간에 상태를 복사할 수 있습니다.

Added in version 3.2.

get(*section*, *option*, **, *raw=False*, *vars=None*[, *fallback*])

명명된 *section*에서 *option* 값을 가져옵니다. *vars*가 제공되면, 딕셔너리이어야 합니다. *option*은 *vars* (제공되면), *section* 및 *DEFAULTSECT* 에서 순서대로 조회됩니다. 키를 찾을 수 없고 *fallback*이 제공되면, 대체 값으로 사용됩니다. `None`은 *fallback* 값으로 제공될 수 있습니다.

raw 인자가 참이 아닌 한, 모든 `'%'` 보간이 반환 값에서 확장됩니다. 보간 키의 값은 옵션과 같은 방식으로 조회됩니다.

버전 3.2에서 변경: 인자 *raw*, *vars* 및 *fallback*은 사용자가 세 번째 인자를 *fallback* 폴 백으로 사용하지 못하도록 하기 위해 키워드 전용입니다 (특히 매핑 프로토콜을 사용할 때).

getint(*section*, *option*, **, *raw=False*, *vars=None*[, *fallback*])

지정된 *section*의 *option*을 정수로 강제 변환하는 편의 메서드. *raw*, *vars* 및 *fallback*에 대한 설명은 `get()`을 참조하십시오.

getfloat(*section*, *option*, **, *raw=False*, *vars=None*[, *fallback*])

지정된 *section*의 *option*을 부동 소수점 수로 강제 변환하는 편의 메서드. *raw*, *vars* 및 *fallback*에 대한 설명은 `get()`을 참조하십시오.

getboolean(*section*, *option*, **, *raw=False*, *vars=None*[, *fallback*])

지정된 *section*의 *option*을 불리언 값으로 강제 변환하는 편의 메서드. 옵션에 허용되는 값은 이 메서드가 `True`를 반환하게 하는 `'1'`, `'yes'`, `'true'` 및 `'on'`과 `False`를 반환하게 하는 `'0'`, `'no'`, `'false'` 및 `'off'`입니다. 이 문자열 값은 대소 문자를 구분하지 않고 확인됩니다. 다른 모든 값은 `ValueError`를 발생시킵니다. *raw*, *vars* 및 *fallback*에 대한 설명은 `get()`을 참조하십시오.

items(*raw=False*, *vars=None*)

items(*section*, *raw=False*, *vars=None*)

*section*이 제공되지 않으면, DEFAULTSECT를 포함하여, *section_name*, *section_proxy* 쌍의 리스트를 반환합니다.

그렇지 않으면, 주어진 *section*의 옵션에 대해 *name*, *value* 쌍의 리스트를 반환합니다. 선택적 인자는 `get()` 메서드에서와 같은 의미입니다.

버전 3.8에서 변경: *vars*에 있는 항목은 더는 결과에 나타나지 않습니다. 이전 동작은 실제 구문 분석기 옵션과 보간을 위해 제공된 변수를 혼합했습니다.

set(*section*, *option*, *value*)

주어진 섹션이 존재하면, 주어진 옵션을 지정된 값으로 설정합니다; 그렇지 않으면 `NoSectionError`를 발생시킵니다. *option*과 *value*는 문자열이어야 합니다; 그렇지 않으면, `TypeError`가 발생합니다.

write(*fileobject*, *space_around_delimiters=True*)

텍스트 모드로 열렸어야 하는 (문자열을 받아들이는), 지정된 파일 객체에 구성의 표현을 기록합니다. 이 표현은 향후 `read()` 호출로 구문 분석할 수 있습니다. *space_around_delimiters*가 참이면, 키와 값 사이의 구분자는 공백으로 둘러싸입니다.

버전 3.14에서 변경: Raises InvalidWriteError if this would write a representation which cannot be accurately parsed by a future `read()` call from this parser.

> **참고**
>
> Comments in the original configuration file are not preserved when writing the configuration back. What is considered a comment, depends on the given values for *comment_prefix* and *inline_comment_prefix*.

remove_option(*section*, *option*)

지정된 *section*에서 지정된 *option*을 제거합니다. 섹션이 존재하지 않으면, `NoSectionError`를 발생시킵니다. 제거되는 옵션이 존재했으면 `True`를 반환합니다; 그렇지 않으면 `False`를 반환합니다.

remove_section(*section*)

지정된 *section*을 구성에서 제거합니다. 실제로 섹션이 존재하면, `True`를 반환합니다. 그렇지 않으면 `False`를 반환합니다.

optionxform(*option*)

입력 파일에서 발견되거나 클라이언트 코드에서 전달된 옵션 이름 *option*을 내부 구조에서 사용되어야 하는 형식으로 변환합니다. 기본 구현은 *option*의 소문자 버전을 반환합니다; 이 동작에 영향을 주기 위해 서브 클래스가 이것을 재정의하거나 클라이언트 코드가 인스턴스에 이 이름의 어트리뷰트를 설정할 수 있습니다.

이 메서드를 사용하기 위해 구문 분석기를 서브 클래싱할 필요는 없으며, 문자열 인자를 취해서 문자열을 반환하는 함수로 인스턴스에 설정할 수도 있습니다. 예를 들어 `str`로 설정하면 옵션 이름이 대소 문자를 구분하게 됩니다:

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

구성 파일을 읽을 때, `optionxform()`이 호출되기 전에 옵션 이름 주위의 공백이 제거됨에 유의하십시오.

configparser.**UNNAMED_SECTION**

A special object representing a section name used to reference the unnamed section (see *Unnamed Sections*).

configparser.**MAX_INTERPOLATION_DEPTH**

raw 매개 변수가 거짓일 때 `get()`의 재귀 보간의 최대 깊이. 기본 *interpolation*을 사용할 때만 의미 있습니다.

14.2.11 RawConfigParser 객체

class configparser.**RawConfigParser**(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*, *interpolation=BasicInterpolation()*, *converters={}*, *allow_unnamed_section=False*)

`ConfigParser`의 레거시 변형. 기본적으로 보간이 비활성화되어 있으며 레거시 `defaults=` 키워드 인자 처리뿐만 아니라 안전하지 않은 `add_section`과 `set` 메서드를 통해 문자열이 아닌 섹션 이름, 옵션 이름 및 값을 허용합니다.

버전 3.2에서 변경: *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* 및 *interpolation*이 추가되었습니다.

버전 3.5에서 변경: *converters* 인자가 추가되었습니다.

버전 3.8에서 변경: 이제 삽입 순서를 유지하므로, 기본 *dict_type*은 `dict`입니다.

버전 3.13에서 변경: *allow_unnamed_section* 인자가 추가되었습니다.

> **참고**
>
> 내부에 저장되는 값의 형을 검사하는 `ConfigParser`를 대신 사용하는 것을 고려하십시오. 보간을 원하지 않으면, `ConfigParser(interpolation=None)` 을 사용할 수 있습니다.

add_section(*section*)

Add a section named *section* or `UNNAMED_SECTION` to the instance.

If the given section already exists, `DuplicateSectionError` is raised. If the *default section* name is passed, `ValueError` is raised. If `UNNAMED_SECTION` is passed and support is disabled, `UnnamedSectionDisabledError` is raised.

*section*의 형을 검사하지 않아서 사용자가 문자열이 아닌 섹션을 만들 수 있도록 합니다. 이 동작은 지원되지 않으며 내부 에러를 일으킬 수 있습니다.

버전 3.14에서 변경: Added support for `UNNAMED_SECTION`.

set(*section*, *option*, *value*)

주어진 섹션이 존재하면, 주어진 옵션을 지정된 값으로 설정합니다; 그렇지 않으면 `NoSectionError`를 발생시킵니다. 문자열이 아닌 값을 내부에 저장하도록 `RawConfigParser`(또는 *raw* 매개 변수가 참으로 설정된 `ConfigParser`)를 사용할 수 있지만, 전체 기능(보간과 파일로의 출력을 포함하는)은 문자열 값으로만 수행할 수 있습니다.

이 메서드를 사용하면 문자열이 아닌 값을 키에 내부적으로 대입할 수 있습니다. 이 동작은 지원되지 않으며 파일에 쓰려고 하거나 비 원시 모드에서 가져오려고 할 때 에러가 발생합니다. 이러한 대입을 허용하지 않는 **매핑 프로토콜 API를 사용하십시오**.

14.2.12 예외

exception configparser.**Error**

다른 모든 `configparser` 예외의 베이스 클래스.

exception configparser.**NoSectionError**

지정된 섹션을 찾지 못할 때 발생하는 예외.

exception configparser.**DuplicateSectionError**

이미 존재하는 이름으로 `add_section()`을 호출하거나 엄격한(strict) 구문 분석기에서 단일 입력 파일, 문자열 또는 딕셔너리에서 섹션이 두 번 이상 발견될 때 발생하는 예외.

버전 3.2에서 변경: 선택적 *source*와 *lineno* 어트리뷰트 및 `__init__()` 에 대한 인자를 추가했습니다.

exception configparser.**DuplicateOptionError**

단일 파일, 문자열 또는 딕셔너리에서 읽는 동안 단일 옵션이 두 번 나타날 때 엄격한(strict) 구문 분석기가 발생시키는 예외. 이는 맞춤법과 대소 문자 구분과 관련된 에러를 잡습니다, 예를 들어 딕셔너리에는 대소 문자를 구분하지 않는 같은 구성 키를 나타내는 두 개의 키가 있을 수 있습니다.

exception configparser.**NoOptionError**

지정된 섹션에서 지정된 옵션을 찾지 못할 때 발생하는 예외.

exception configparser.**InterpolationError**

문자열 보간 수행 시 문제가 발생할 때 발생하는 예외의 베이스 클래스.

exception configparser.**InterpolationDepthError**

이터레이션 횟수가 `MAX_INTERPOLATION_DEPTH`를 초과하여 문자열 보간을 완료할 수 없을 때 발생하는 예외. `InterpolationError`의 서브 클래스.

exception configparser.**InterpolationMissingOptionError**

값에서 참조된 옵션이 존재하지 않을 때 발생하는 예외. `InterpolationError`의 서브 클래스.

exception configparser.**InterpolationSyntaxError**

치환이 이루어질 소스 텍스트가 요구되는 문법을 준수하지 않을 때 발생하는 예외. *InterpolationError*의 서브 클래스.

exception configparser.**MissingSectionHeaderError**

섹션 헤더가 없는 파일을 구문 분석하려고 할 때 발생하는 예외.

exception configparser.**ParsingError**

파일 구문 분석 중에 에러가 발생할 때 발생하는 예외.

버전 3.12에서 변경: 일관성을 위해 `filename` 어트리뷰트와 `__init__()` 생성자 인자가 제거되었습니다. 3.2부터 `source`라는 이름으로 사용할 수 있었습니다.

exception configparser.**MultilineContinuationError**

Exception raised when a key without a corresponding value is continued with an indented line.

Added in version 3.13.

exception configparser.**UnnamedSectionDisabledError**

Exception raised when attempting to use the *UNNAMED_SECTION* without enabling it.

Added in version 3.14.

exception configparser.**InvalidWriteError**

Exception raised when an attempted *ConfigParser.write()* would not be parsed accurately with a future *ConfigParser.read()* call.

Ex: Writing a key beginning with the *ConfigParser.SECTCRE* pattern would parse as a section header when read. Attempting to write this will raise this exception.

Added in version 3.14.

14.3 tomllib — Parse TOML files

Added in version 3.11.

Source code: Lib/tomllib

This module provides an interface for parsing TOML 1.0.0 (Tom' s Obvious Minimal Language, https://toml.io). This module does not support writing TOML.

> **더 보기**
>
> The Tomli-W package is a TOML writer that can be used in conjunction with this module, providing a write API familiar to users of the standard library *marshal* and *pickle* modules.

> **더 보기**
>
> The TOML Kit package is a style-preserving TOML library with both read and write capability. It is a recommended replacement for this module for editing already existing TOML files.

This module defines the following functions:

tomllib.**load**(*fp*, /, *, *parse_float=float*)

Read a TOML file. The first argument should be a readable and binary file object. Return a *dict*. Convert TOML types to Python using this *conversion table*.

parse_float will be called with the string of every TOML float to be decoded. By default, this is equivalent to `float(num_str)`. This can be used to use another datatype or parser for TOML floats (e.g. `decimal.Decimal`). The callable must not return a `dict` or a `list`, else a `ValueError` is raised.

A `TOMLDecodeError` will be raised on an invalid TOML document.

`tomllib.loads`(*s*, /, *, *parse_float=float*)

Load TOML from a `str` object. Return a `dict`. Convert TOML types to Python using this *conversion table*. The *parse_float* argument has the same meaning as in `load()`.

A `TOMLDecodeError` will be raised on an invalid TOML document.

The following exceptions are available:

exception `tomllib.TOMLDecodeError`(*msg*, *doc*, *pos*)

Subclass of `ValueError` with the following additional attributes:

msg

The unformatted error message.

doc

The TOML document being parsed.

pos

The index of *doc* where parsing failed.

lineno

The line corresponding to *pos*.

colno

The column corresponding to *pos*.

버전 3.14에서 변경: Added the *msg*, *doc* and *pos* parameters. Added the `msg`, `doc`, `pos`, `lineno` and `colno` attributes.

버전 3.14부터 폐지됨: Passing free-form positional arguments is deprecated.

14.3.1 Examples

Parsing a TOML file:

```
import tomllib

with open("pyproject.toml", "rb") as f:
    data = tomllib.load(f)
```

Parsing a TOML string:

```
import tomllib

toml_str = """
python-version = "3.11.0"
python-implementation = "CPython"
"""

data = tomllib.loads(toml_str)
```

14.3.2 Conversion Table

TOML	Python
TOML document	dict
string	str
integer	int
float	float (configurable with *parse_float*)
boolean	bool
offset date-time	datetime.datetime (`tzinfo` attribute set to an instance of `datetime.timezone`)
local date-time	datetime.datetime (`tzinfo` attribute set to `None`)
local date	datetime.date
local time	datetime.time
array	list
table	dict
inline table	dict
array of tables	list of dicts

14.4 `netrc` — netrc 파일 처리

소스 코드: Lib/netrc.py

---

netrc 클래스는 유닉스 **ftp** 프로그램과 다른 FTP 클라이언트가 사용하는 netrc 파일 형식을 구문 분석하고 캡슐화합니다.

class netrc.**netrc**([*file*])

A *netrc* instance or subclass instance encapsulates data from a netrc file. The initialization argument, if present, specifies the file to parse. If no argument is given, the file `.netrc` in the user' s home directory – as determined by *os.path.expanduser()* – will be read. Otherwise, a *FileNotFoundError* exception will be raised. Parse errors will raise *NetrcParseError* with diagnostic information including the file name, line number, and terminating token.

If no argument is specified on a POSIX system, the presence of passwords in the `.netrc` file will raise a *NetrcParseError* if the file ownership or permissions are insecure (owned by a user other than the user running the process, or accessible for read or write by any other user). This implements security behavior equivalent to that of ftp and other programs that use `.netrc`. Such security checks are not available on platforms that do not support *os.getuid()*.

버전 3.4에서 변경: POSIX 권한 검사를 추가했습니다.

버전 3.7에서 변경: *file*이 인자로 전달되지 않으면 *os.path.expanduser()*가 `.netrc` 파일의 위치를 찾는 데 사용됩니다.

버전 3.10에서 변경: *netrc* try UTF-8 encoding before using locale specific encoding. The entry in the netrc file no longer needs to contain all tokens. The missing tokens' value default to an empty string. All the tokens and their values now can contain arbitrary characters, like whitespace and non-ASCII characters. If the login name is anonymous, it won' t trigger the security check.

exception netrc.**NetrcParseError**

소스 텍스트에 문법적인 에러가 있을 때 *netrc* 클래스에서 발생하는 예외. 이 예외 인스턴스는 세 가지 흥미로운 어트리뷰트를 제공합니다.

msg

에러의 텍스트 설명.

filename

소스 파일의 이름.

lineno

에러가 발견된 줄 번호.

14.4.1 netrc 객체

netrc 인스턴스에는 다음과 같은 메서드가 있습니다:

netrc.**authenticators**(*host*)

*host*에 대한 인증 자의 3-tuple (login, account, password)를 반환합니다. netrc 파일에 주어진 호스트에 대한 항목이 없으면 'default' 항목과 연관된 튜플을 반환합니다. 일치하는 호스트도 기본 항목도 사용할 수 없으면 None을 반환합니다.

netrc.**__repr__**()

클래스 데이터를 netrc 파일의 형식의 문자열로 덤프합니다. (이것은 주석을 버리고 엔트리를 재정렬할 수 있습니다.)

*netrc*의 인스턴스에는 공개 인스턴스 변수가 있습니다:

netrc.**hosts**

호스트 이름을 (login, account, password) 튜플에 매핑하는 딕셔너리. 'default' 항목이 있으면 그 이름의 의사 호스트로 표시됩니다.

netrc.**macros**

매크로 이름을 문자열 리스트에 매핑하는 딕셔너리.

14.5 plistlib — 애플 .plist 파일 생성과 구문 분석

소스 코드: Lib/plistlib.py

이 모듈은 애플, 주로 macOS와 iOS에서 사용되는 "프로퍼티 리스트(property list)" 파일을 읽고 쓰는 인터페이스를 제공합니다. 이 모듈은 바이너리와 XML plist 파일을 모두 지원합니다.

프로퍼티 리스트 (.plist) 파일 형식은 딕셔너리, 리스트, 숫자 및 문자열과 같은 기본 객체 형을 지원하는 간단한 직렬화입니다. 일반적으로 최상위 객체는 딕셔너리입니다.

plist 파일을 쓰고 구문 분석하려면, *dump()*와 *load()* 함수를 사용하십시오.

plist 데이터를 바이트열이나 문자열 객체로 작업하려면, *dumps()*와 *loads()*를 사용하십시오.

값은 문자열, 정수, 부동 소수점, 논릿값, 튜플, 리스트, 딕셔너리 (단, 문자열 키만 가능), *bytes*, *bytearray* 또는 *datetime.datetime* 객체일 수 있습니다.

버전 3.4에서 변경: 새 API, 이전 API는 폐지되었습니다. 바이너리 형식 plist에 대한 지원이 추가되었습니다.

버전 3.8에서 변경: NSKeyedArchiver 와 NSKeyedUnarchiver 에서 사용되듯이 바이너리 plist에서 *UID* 토큰을 읽고 쓰는 것에 대한 지원이 추가되었습니다.

버전 3.9에서 변경: 낡은 API가 제거되었습니다.

더 보기

PList 매뉴얼 페이지
애플의 파일 형식 설명서.

이 모듈은 다음 함수를 정의합니다:

plistlib.**load**(*fp*, *, *fmt=None*, *dict_type=dict*, *aware_datetime=False*)

plist 파일을 읽습니다. *fp*는 읽을 수 있는 바이너리 파일 객체여야 합니다. 해독된 루트 객체를 반환합니다 (일반적으로 딕셔너리입니다).

*fmt*는 파일의 형식이며 다음 값이 유효합니다:

- *None*: 파일 형식을 자동 감지

- *FMT_XML*: XML 파일 형식
- *FMT_BINARY*: 바이너리 plist 형식

*dict_type*은 plist 파일에서 읽은 딕셔너리에 사용되는 형입니다.

When *aware_datetime* is true, fields with type `datetime.datetime` will be created as *aware object*, with `tzinfo` as *datetime.UTC*.

FMT_XML 형식의 XML 데이터는 *xml.parsers.expat* 의 Expat 구문 분석기로 구문 분석됩니다 – 잘못된 형식의 XML로 인한 예외에 대해서는 해당 설명서를 참조하십시오. 알 수 없는 엘리먼트는 plist 구문분석기에서 단순히 무시됩니다.

구문 분석기는 파일을 구문 분석할 수 없을 때 *InvalidFileException*를 발생시킵니다.

Added in version 3.4.

버전 3.13에서 변경: 키워드 전용 매개 변수 *aware_datetime*이 추가되었습니다.

`plistlib.`**`loads`**(*data, *, fmt=None, dict_type=dict, aware_datetime=False*)

바이트열이나 문자열 객체에서 plist를 로드합니다. 키워드 인자에 대한 설명은 *load()*를 참조하십시오.

Added in version 3.4.

버전 3.13에서 변경: *data* can be a string when *fmt* equals *FMT_XML*.

`plistlib.`**`dump`**(*value, fp, *, fmt=FMT_XML, sort_keys=True, skipkeys=False, aware_datetime=False*)

plist 파일에 *value*를 씁니다. *fp*는 쓰기 가능한 바이너리 파일 객체여야 합니다.

fmt 인자는 plist 파일의 형식을 지정하며 다음 값 중 하나일 수 있습니다:

- *FMT_XML*: XML 형식의 plist 파일
- *FMT_BINARY*: 바이너리 형식의 plist 파일

*sort_keys*가 참(기본값)이면 딕셔너리의 키가 정렬된 순서로 plist에 기록되고, 그렇지 않으면 딕셔너리의 이터레이션 순서로 기록됩니다.

*skipkeys*가 거짓(기본값)일 때, 딕셔너리의 키가 문자열이 아니면 함수는 *TypeError*를 발생시킵니다. 그렇지 않으면 해당 키를 건너뜁니다.

When *aware_datetime* is true and any field with type `datetime.datetime` is set as an *aware object*, it will convert to UTC timezone before writing it.

객체가 지원되지 않는 형이거나 지원되지 않는 형의 객체를 포함하는 컨테이너면 *TypeError*가 발생합니다.

(바이너리) plist 파일에서 표현할 수 없는 정숫값은 *OverflowError*를 발생시킵니다.

Added in version 3.4.

버전 3.13에서 변경: 키워드 전용 매개 변수 *aware_datetime*이 추가되었습니다.

`plistlib.`**`dumps`**(*value, *, fmt=FMT_XML, sort_keys=True, skipkeys=False, aware_datetime=False*)

plist 형식의 바이트열 객체로 *value*를 반환합니다. 이 함수의 키워드 인자에 대한 설명은 *dump()* 설명서를 참조하십시오.

Added in version 3.4.

다음 클래스를 사용할 수 있습니다:

class `plistlib.`**`UID`**(*data*)

*int*를 감쌉니다. 이것은 UID를 포함하는 NSKeyedArchiver 인코딩된 데이터를 읽거나 쓸 때 사용됩니다 (PList 매뉴얼을 참조하십시오).

`data`

UID의 int 값. 반드시 범위 `0 <= data < 2**64`에 있어야 합니다.

Added in version 3.8.

다음 상수를 사용할 수 있습니다:

plistlib.**FMT_XML**

plist 파일의 XML 형식.

Added in version 3.4.

plistlib.**FMT_BINARY**

plist 파일의 바이너리 형식

Added in version 3.4.

이 모듈은 다음 예외를 정의합니다:

exception plistlib.**InvalidFileException**

파일을 구문 분석할 수 없을 때 발생합니다.

Added in version 3.4.

14.5.1 예제

plist 만들기:

```
import datetime
import plistlib

pl = dict(
    aString = "Doodah",
    aList = ["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict = dict(
        anotherString = "<hello & hi there!>",
        aThirdString = "M\xe4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = b"<binary gunk>",
    someMoreData = b"<lots of binary gunk>" * 10,
    aDate = datetime.datetime.now()
)
print(plistlib.dumps(pl).decode())
```

plist 구문 분석하기:

```
import plistlib

plist = b"""<plist version="1.0">
<dict>
    <key>foo</key>
    <string>bar</string>
</dict>
</plist>"""
pl = plistlib.loads(plist)
print(pl["foo"])
```

CHAPTER 15

암호화 서비스

이 장에서 설명하는 모듈은 다양한 암호화 알고리즘을 구현합니다. 가용성은 설치에 달려있습니다. 다음은 개요입니다:

15.1 hashlib — 보안 해시와 메시지 요약

소스 코드: Lib/hashlib.py

This module implements a common interface to many different hash algorithms. Included are the FIPS secure hash algorithms SHA224, SHA256, SHA384, SHA512, (defined in the FIPS 180-4 standard), the SHA-3 series (defined in the FIPS 202 standard) as well as the legacy algorithms SHA1 (formerly part of FIPS) and the MD5 algorithm (defined in internet **RFC 1321**).

> **참고**
>
> adler32나 crc32 해시 함수를 원한다면, `zlib` 모듈에 있습니다.

15.1.1 해시 알고리즘

해시(*hash*)의 유형마다 이름이 지정된 생성자 메서드가 있습니다. 모두 같은 간단한 인터페이스를 갖는 해시 객체를 반환합니다. 예를 들어: SHA-256 해시 객체를 만들려면 `sha256()`을 사용하십시오. 이제 `update` 메서드를 사용하여 이 객체에 바이트열류 객체(보통 `bytes`)를 공급할 수 있습니다. 언제든지 `digest()`나 `hexdigest()` 메서드를 사용하여 지금까지 공급된 데이터의 연결에 대한 요약(*digest*)을 요청할 수 있습니다.

To allow multithreading, the Python *GIL* is released while computing a hash supplied more than 2047 bytes of data at once in its constructor or `.update` method.

이 모듈에 항상 존재하는 해시 알고리즘의 생성자는 `sha1()`, `sha224()`, `sha256()`, `sha384()`, `sha512()`, `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()`, `blake2b()` 및 `blake2s()`입니다. `md5()`는 일반적으로 사용할 수 있지만, 드문 "FIPS 호환" 파이썬 빌드를 사용하는 경우에는 빠지거나 차단될 수 있습니다. 이는 `algorithms_guaranteed`에 해당합니다.

Additional algorithms may also be available if your Python distribution' s `hashlib` was linked against a build of OpenSSL that provides others. Others *are not guaranteed available* on all installations and will only be accessible by name via `new()`. See `algorithms_available`.

> **경고**
>
> (MD5와 SHA1을 포함하는) 일부 알고리즘은 해시 충돌 약점(hash collision weaknesses)이 알려져 있습니다. Attacks on cryptographic hash algorithms와 이 문서의 끝에 있는 더 보기 섹션을 참조하십시오.

Added in version 3.6: SHA3(Keccak)과 SHAKE 생성자 `sha3_224()`, `sha3_256()`, `sha3_384()`, `sha3_512()`, `shake_128()`, `shake_256()`이 추가되었습니다. `blake2b()`와 `blake2s()`가 추가되었습니다. 버전 3.9에서 변경: 모든 hashlib 생성자는 기본값이 `True`인 키워드 전용 인자 *usedforsecurity*를 취합니다. 값이 거짓이면 제한된 환경에서 안전하지 않고 차단된 해싱 알고리즘 사용을 허락합니다. `False`는 해싱 알고리즘이 보안 문맥에서 사용되지 않음을 나타냅니다, 예를 들어 암호화가 아닌 단방향 압축 함수로.

버전 3.9에서 변경: Hashlib는 이제 OpenSSL 에서 SHA3와 SHAKE를 제공하는 경우 이를 사용합니다.

버전 3.12에서 변경: For any of the MD5, SHA1, SHA2, or SHA3 algorithms that the linked OpenSSL does not provide we fall back to a verified implementation from the HACL* project.

15.1.2 사용법

바이트 문자열 `b"Nobody inspects the spammish repetition"`의 요약을 얻으려면 다음을 수행하십시오:

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}Ae\x15\x93\xc5\xfe\\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:\xa6\xaf\x0c\
↪x95\x0fK\x94\x06'
>>> m.hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

더 압축하면:

```
>>> hashlib.sha256(b"Nobody inspects the spammish repetition").hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

15.1.3 생성자

`hashlib.`**`new`**(*name*, [*data*,]*, *usedforsecurity=True*)

원하는 알고리즘의 문자열 *name*을 첫 번째 매개 변수로 취하는 일반 생성자입니다. 또한 위에 나열된 해시뿐만 아니라 OpenSSL 라이브러리가 제공할 수 있는 다른 알고리즘에 대한 액세스를 허용하기 위해 존재합니다.

알고리즘 이름으로 `new()` 사용하기:

```
>>> h = hashlib.new('sha256')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

`hashlib.`**`md5`**([*data*,]*, *usedforsecurity=True*)

`hashlib.`**`sha1`**([*data*,]*, *usedforsecurity=True*)

`hashlib.`**`sha224`**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha256**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha384**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha512**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_224**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_256**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_384**([*data*,]*, *usedforsecurity=True*)

hashlib.**sha3_512**([*data*,]*, *usedforsecurity=True*)

이름 붙은 생성자는 알고리즘 이름을 `new()`로 전달하는 것보다 빠릅니다.

15.1.4 어트리뷰트

hashlib는 다음과 같은 상수 모듈 어트리뷰트를 제공합니다:

hashlib.**algorithms_guaranteed**

모든 플랫폼에서 이 모듈이 지원하도록 보장된 해시 알고리즘의 이름을 포함하는 집합. 'md5' 는 일부 업스트림 공급자가 이를 제외하는 이상한 "FIPS 호환" 파이썬 빌드를 제공하지만, 이 목록에 있음에 유의하십시오.

Added in version 3.2.

hashlib.**algorithms_available**

실행 중인 파이썬 인터프리터에서 사용 가능한 해시 알고리즘의 이름이 포함된 집합. 이 이름들은 `new()`에 전달될 때 인식됩니다. `algorithms_guaranteed`는 항상 부분 집합입니다. 이 집합에서 같은 알고리즘이 다른 이름으로 여러 번 나타날 수 있습니다 (OpenSSL 덕분입니다).

Added in version 3.2.

15.1.5 해시 객체

다음 값은 생성자가 반환한 해시 객체의 상수 어트리뷰트로 제공됩니다:

hash.**digest_size**

결과 해시의 바이트 단위의 크기.

hash.**block_size**

해시 알고리즘의 바이트 단위의 내부 블록 크기.

해시 객체에는 다음과 같은 어트리뷰트가 있습니다:

hash.**name**

이 해시의 규범적 이름, 항상 소문자이며 항상 이 유형의 다른 해시를 만들기 위한 `new()`에 대한 매개변수로 적합합니다.

버전 3.4에서 변경: name 어트리뷰트는 처음부터 CPython에 존재했지만, 파이썬 3.4 이전에는 공식적으로 지정되지 않아서, 일부 플랫폼에는 존재하지 않을 수 있습니다.

해시 객체에는 다음과 같은 메서드가 있습니다:

hash.**update**(*data*)

바이트열류 객체로 해시 객체를 갱신합니다. 반복되는 호출은 모든 인자를 이어붙인 단일 호출과 동등합니다: `m.update(a); m.update(b)`는 `m.update(a+b)`와 동등합니다.

hash.**digest**()

지금까지 `update()` 메서드에 전달된 데이터의 요약을 반환합니다. 이것은 `digest_size` 크기의 바이트열 객체이며 0에서 255까지의 전체 범위에 있는 바이트를 포함할 수 있습니다.

hash.**hexdigest**()

digest() 와 유사하지만, 요약은 16진수 숫자만 포함하는 두 배 길이의 문자열 객체로 반환됩니다. 전자 메일이나 기타 바이너리가 아닌 환경에서 값을 안전하게 교환하는 데 사용할 수 있습니다.

hash.**copy**()

해시 객체의 사본("복제본")을 반환합니다. 이것은 공통된 초기 부분 문자열을 공유하는 데이터의 요약을 효율적으로 계산하는 데 사용될 수 있습니다.

15.1.6 SHAKE 가변 길이 요약

hashlib.**shake_128**([*data*,]*, *usedforsecurity=True*)

hashlib.**shake_256**([*data*,]*, *usedforsecurity=True*)

shake_128() 과 shake_256() 알고리즘은 length_in_bits//2 (최대 128이나 256) 비트의 보안성으로 가변 길이 요약을 제공합니다. 따라서 digest 메서드에는 길이(length)가 필요합니다. 최대 길이는 SHAKE 알고리즘에 의해 제한되지 않습니다.

shake.**digest**(*length*)

지금까지 update() 메서드에 전달된 데이터의 요약을 반환합니다. 이것은 *length* 크기의 바이트열 객체이며 0에서 255까지의 전체 범위에 있는 바이트를 포함할 수 있습니다.

shake.**hexdigest**(*length*)

digest() 와 유사하지만, 요약은 16진수 숫자만 포함하는 두 배 길이의 문자열 객체로 반환됩니다. 전자 메일이나 기타 바이너리가 아닌 환경에서 값을 교환하는 데 사용할 수 있습니다.

사용 예:

```
>>> h = hashlib.shake_256(b'Nobody inspects the spammish repetition')
>>> h.hexdigest(20)
'44709d6fcb83d92a76dcb0b668c98e1b1d3dafe7'
```

15.1.7 파일 해싱

The hashlib module provides a helper function for efficient hashing of a file or file-like object.

hashlib.**file_digest**(*fileobj*, *digest*, /)

Return a digest object that has been updated with contents of file object.

fileobj must be a file-like object opened for reading in binary mode. It accepts file objects from builtin open(), BytesIO instances, SocketIO objects from socket.socket.makefile(), and similar. *fileobj* must be opened in blocking mode, otherwise a BlockingIOError may be raised.

The function may bypass Python' s I/O and use the file descriptor from fileno() directly. *fileobj* must be assumed to be in an unknown state after this function returns or raises. It is up to the caller to close *fileobj*.

digest must either be a hash algorithm name as a *str*, a hash constructor, or a callable that returns a hash object.

예제:

```
>>> import io, hashlib, hmac
>>> with open("library/hashlib.rst", "rb") as f:
...     digest = hashlib.file_digest(f, "sha256")
...
>>> digest.hexdigest()
'...'
```

```
>>> buf = io.BytesIO(b"somedata")
>>> mac1 = hmac.HMAC(b"key", digestmod=hashlib.sha512)
>>> digest = hashlib.file_digest(buf, lambda: mac1)
```

```
>>> digest is mac1
True
>>> mac2 = hmac.HMAC(b"key", b"somedata", digestmod=hashlib.sha512)
>>> mac1.digest() == mac2.digest()
True
```

Added in version 3.11.

버전 3.14에서 변경: Now raises a `BlockingIOError` if the file is opened in blocking mode. Previously, spurious null bytes were added to the digest.

15.1.8 키 파생

키 파생(key derivation)과 키 확장(key stretching) 알고리즘은 안전한 암호 해싱을 위해 설계되었습니다. `sha1(password)` 와 같은 순진한 알고리즘은 무차별 대입 공격에 내성이 없습니다. 올바른 암호 해싱 함수는 조정할 수 있고, 느리고, 솔트(salt)를 포함해야 합니다.

`hashlib.`**`pbkdf2_hmac`**(*hash_name*, *password*, *salt*, *iterations*, *dklen=None*)

이 함수는 PKCS#5 암호 기반 키 파생 함수 2를 제공합니다. 의사 난수 함수로 HMAC을 사용합니다.

문자열 *hash_name*은 원하는 HMAC을 위한 해시 요약 알고리즘의 이름입니다, 예를 들어 'sha1' 이나 'sha256'. *password*와 *salt*는 바이트 버퍼로 해석됩니다. 응용 프로그램과 라이브러리는 *password*를 적당한 길이(예를 들어 1024)로 제한해야 합니다. *salt*는 적절한 소스(예를 들어 `os.urandom()`)로부터 온 약 16이나 그 이상의 바이트여야 합니다.

The number of *iterations* should be chosen based on the hash algorithm and computing power. As of 2022, hundreds of thousands of iterations of SHA-256 are suggested. For rationale as to why and how to choose what is best for your application, read *Appendix A.2.2* of NIST-SP-800-132. The answers on the stackexchange pbkdf2 iterations question explain in detail.

*dklen*은 파생 키의 바이트 수로 표현한 길이입니다. *dklen*이 `None`이면 해시 알고리즘 *hash_name*의 요약 크기가 사용됩니다, 예를 들어 SHA-512의 경우 64.

```
>>> from hashlib import pbkdf2_hmac
>>> our_app_iters = 500_000  # Application specific, read above.
>>> dk = pbkdf2_hmac('sha256', b'password', b'bad salt' * 2, our_app_iters)
>>> dk.hex()
'15530bba69924174860db778f2c6f8104d3aaf9d26241840c8c4a641c8d000a9'
```

Function only available when Python is compiled with OpenSSL.

Added in version 3.4.

버전 3.12에서 변경: Function now only available when Python is built with OpenSSL. The slow pure Python implementation has been removed.

`hashlib.`**`scrypt`**(*password*, **, salt*, *n*, *r*, *p*, *maxmem=0*, *dklen=64*)

이 함수는 **RFC 7914**에 정의된 대로 scrypt 암호 기반 키 파생 함수를 제공합니다.

*password*와 *salt*는 바이트열류 객체여야 합니다. 응용 프로그램과 라이브러리는 *password*를 적당한 길이(예를 들어 1024)로 제한해야 합니다. *salt*는 적절한 소스(예를 들어 `os.urandom()`)로부터 온 약 16이나 그 이상의 바이트여야 합니다.

*n*은 CPU/ 메모리 비용 계수, *r*은 블록 크기, *p*는 병렬화 계수이고 *maxmem*은 메모리를 제한합니다 (OpenSSL 1.1.0의 기본값은 32 MiB 입니다). *dklen*은 파생 키의 바이트 수로 표현한 길이입니다.

Added in version 3.6.

15.1.9 BLAKE2

BLAKE2는 **RFC 7693**에 정의된 암호화 해시 함수로, 두 가지 방식으로 제공됩니다:

- **BLAKE2b**, 64비트 플랫폼에 최적화되어 있으며 1에서 64바이트 사이의 모든 크기의 요약을 생성합니다,
- **BLAKE2s**, 8비트에서 32비트 플랫폼에 최적화되어 있으며 1에서 32바이트 사이의 모든 크기의 요약을 생성합니다.

BLAKE2는 **키 모드(keyed mode)** (HMAC의 더 빠르고 간단한 대체), **솔트 해싱(salted hashing)**, **개인화(personalization)** 및 **트리 해싱(tree hashing)**을 지원합니다.

이 모듈의 해시 객체는 표준 라이브러리의 `hashlib` 객체의 API를 따릅니다.

해시 객체 만들기

생성자 함수를 호출하여 새 해시 객체를 만듭니다:

`hashlib.`**`blake2b`**(*data=b'', *, digest_size=64, key=b'', salt=b'', person=b'', fanout=1, depth=1, leaf_size=0, node_offset=0, node_depth=0, inner_size=0, last_node=False, usedforsecurity=True*)

`hashlib.`**`blake2s`**(*data=b'', *, digest_size=32, key=b'', salt=b'', person=b'', fanout=1, depth=1, leaf_size=0, node_offset=0, node_depth=0, inner_size=0, last_node=False, usedforsecurity=True*)

이 함수는 BLAKE2b나 BLAKE2s를 계산하기위한 해당 해시 객체를 반환합니다. 선택적으로 다음과 같은 일반 매개 변수를 취합니다:

- *data*: 해시 할 초기 데이터 청크, 바이트열류 객체여야 합니다. 위치 인자로만 전달될 수 있습니다.
- *digest_size*: 바이트 단위의 출력 요약 크기.
- *key*: 키 해싱을 위한 키 (BLAKE2b의 경우 최대 64바이트, BLAKE2s의 경우 최대 32바이트).
- *salt*: 무작위 해싱을 위한 솔트 (BLAKE2b의 경우 최대 16바이트, BLAKE2s의 경우 최대 8바이트).
- *person*: 개인화 문자열 (BLAKE2b의 경우 최대 16바이트, BLAKE2s의 경우 최대 8바이트).

다음 표는 일반 매개 변수의 제한(바이트 단위)을 보여줍니다:

해시	digest_size	len(key)	len(salt)	len(person)
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

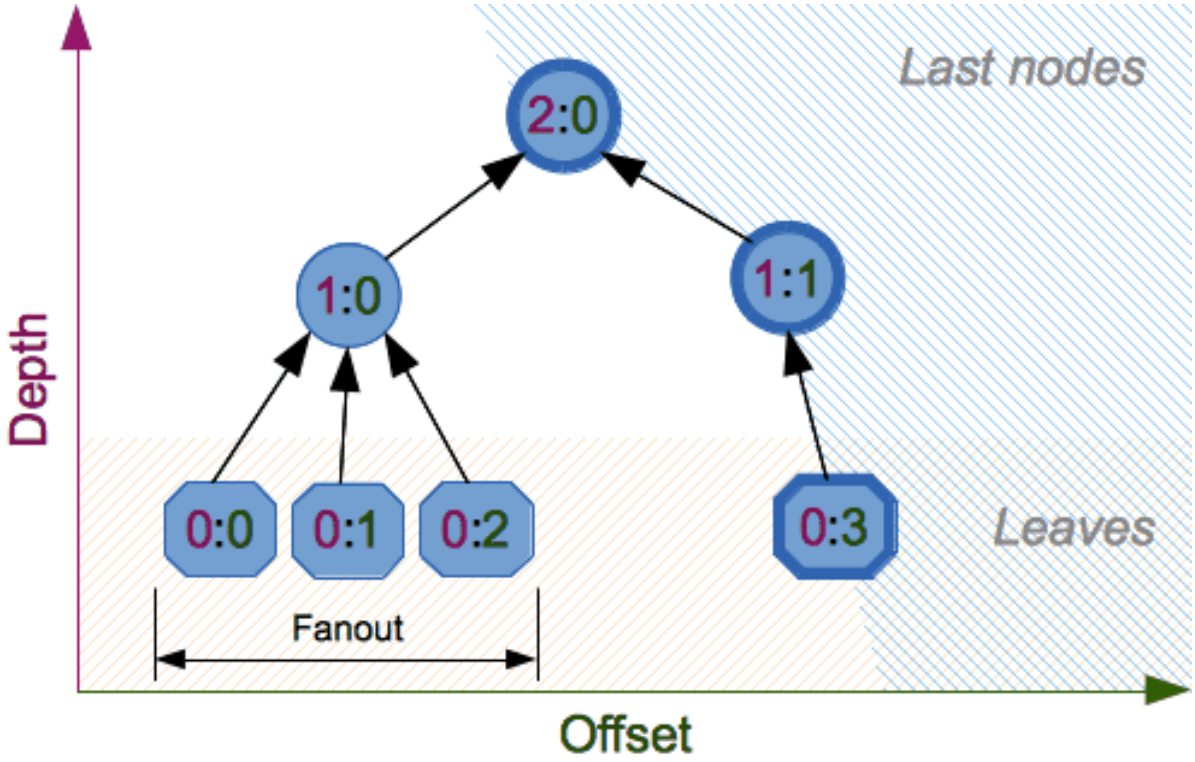
> **참고**
>
> BLAKE2 명세는 솔트와 개인화 매개 변수에 대해 상수 길이를 정의하지만, 편의상, 이 구현에서는 지정된 길이까지 모든 크기의 바이트 문자열을 받아들입니다. 매개 변수의 길이가 지정된 길이보다 작으면, 0으로 채워지므로, 예를 들어 `b'salt'`와 `b'salt\x00'`은 같은 값입니다. (*key*의 경우에는 해당하지 않습니다.)

이 크기는 아래 설명된 모듈 상수로 제공됩니다.

생성자 함수는 다음 트리 해싱 매개 변수도 받아들입니다:

- *fanout*: 팬아웃 (0에서 255, 무제한이면 0, 순차적 모드(sequential mode)이면 1).
- *depth*: 트리의 최대 깊이 (1에서 255, 무제한이면 255, 순차적 모드이면 1).
- *leaf_size*: 리프의 최대 바이트 길이 (0에서 `2**32-1`, 무제한이거나 순차적 모드이면 0).
- *node_offset*: 노드 오프셋 (BLAKE2b의 경우 0에서 `2**64-1`, BLAKE2s의 경우 0에서 `2**48-1`, 첫 번째, 가장 왼쪽, 리프 또는 순차적 모드이면 0).
- *node_depth*: 노드 깊이 (0에서 255, 리프나 순차적 모드이면 0).

- *inner_size*: 내부 요약 크기 (BLAKE2b의 경우 0에서 64, BLAKE2s의 경우 0에서 32, 순차적 모드이면 0).
- *last_node*: 처리된 노드가 마지막 노드인지를 나타내는 불리언 (순차적 모드이면 `False`).



트리 해싱에 대한 포괄적인 리뷰는 BLAKE2 명세의 섹션 2.10을 참조하십시오.

상수

`blake2b.`**`SALT_SIZE`**

`blake2s.`**`SALT_SIZE`**

솔트 길이 (생성자가 허용하는 최대 길이).

`blake2b.`**`PERSON_SIZE`**

`blake2s.`**`PERSON_SIZE`**

개인화 문자열 길이 (생성자가 허용하는 최대 길이).

`blake2b.`**`MAX_KEY_SIZE`**

`blake2s.`**`MAX_KEY_SIZE`**

최대 키 크기.

`blake2b.`**`MAX_DIGEST_SIZE`**

`blake2s.`**`MAX_DIGEST_SIZE`**

해시 함수가 출력할 수 있는 최대 요약 크기.

예

간단한 해싱

어떤 데이터의 해시를 계산하려면, 먼저 적절한 생성자 함수(*blake2b()*나 *blake2s()*)를 호출하여 해시 객체를 생성한 다음, 객체에서 *update()*를 호출하여 데이터로 갱신하고, 마지막으로 *digest()*(또는 16 진수 인코딩 문자열의 경우 *hexdigest()*)를 호출하여 객체에서 요약을 가져와야 합니다.

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb4
↪'
```

줄여서, 첫 번째 데이터 청크를 위치 인자로 생성자에 전달하여 직접 갱신할 수 있습니다:

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb4
↪'
```

해시를 반복적으로 갱신하는 데 필요한 만큼 *hash.update()*를 호출할 수 있습니다:

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...     h.update(item)
...
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb4
↪'
```

다른 요약 크기 사용하기

BLAKE2는 BLAKE2b의 경우 최대 64바이트, BLAKE2s의 경우 최대 32바이트까지 요약 크기를 구성할 수 있습니다. 예를 들어, 출력 크기를 변경하지 않고 SHA-1을 BLAKE2b로 바꾸려면, BLAKE2b에 20바이트 요약을 생성하도록 지시할 수 있습니다:

```
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
>>> h.update(b'Replacing SHA1 with the more secure function')
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```

요약 크기가 다른 해시 객체의 출력은 완전히 다릅니다 (짧은 해시는 긴 해시의 접두사가 아닙니다); BLAKE2b와 BLAKE2s는 출력 길이가 같더라도 다른 출력을 생성합니다:

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

키 해싱

키 해싱(keyed hashing)은 해시 기반 메시지 인증 코드(HMAC - Hash-based message authentication code)을 더 빠르고 간단하게 대체하는 인증에 사용할 수 있습니다. BLAKE2는 BLAKE에서 상속된 비구별성(indifferentiability) 속성 덕분에 접두사-MAC 모드에서 안전하게 사용할 수 있습니다.

이 예는 키 `b'pseudorandom key'`로 메시지 `b'message data'`에 대한 (16진 인코딩된) 128비트 인증 코드를 얻는 방법을 보여줍니다:

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

실용적인 예로, 웹 응용 프로그램은 사용자에게 전송된 쿠키에 대칭적으로 서명한 후 나중에 변조되지 않았는지 확인할 수 있습니다:

```
>>> from hashlib import blake2b
>>> from hmac import compare_digest
>>>
>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16
>>>
>>> def sign(cookie):
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
...     h.update(cookie)
...     return h.hexdigest().encode('utf-8')
>>>
>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)
>>>
>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("{0},{1}".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

네이티브 키 해싱 모드가 있더라도, 물론 BLAKE2를 *hmac* 모듈을 사용하여 HMAC 구성에 사용할 수 있습니다:

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c8142'
```

무작위 해싱

salt 매개 변수를 설정하면 해시 함수에 무작위화를 도입할 수 있습니다. 무작위 해싱은 디지털 서명에 사용된 해시 함수에 대한 충돌 공격(collision attacks)을 방지하는 데 유용합니다.

> 무작위 해싱은 한 당사자(메시지 준비자)가 두 번째 당사자(메시지 서명자)가 서명할 메시지의 전부나 일부를 생성하는 상황을 위해 설계되었습니다. 메시지 준비자가 암호화 해시 함수 충돌(즉, 같은 해시값을 생성하는 두 메시지)을 찾을 수 있으면, 같은 해시값과 디지털 서명을 생성하는 의미 있는 메시지 버전을 준비할 수 있지만, 결과는 다릅니다 (예를 들어, 계정으로 $10 대신에 $1,000,000을 이체하는 행위). 암호화 해시 함수는 주요 목표로 충돌 내성을 갖도록 설계되었지만, 현재 암호화 해시 함수 공격에 대한 집중으로 인해 주어진 암호화 해시 함수가 예상보다 적은 충돌 내성을 제공할 수 있습니다. 무작위 해싱은 준비자가 디지털 서명 생성 프로세스 동안 궁극적으로 같은 해시값을 산출하는 두 개 이상의 메시지를 생성할 가능성을 줄여서, 서명자에게 추가적인 보호를 제공합니다 – 설사 해시 함수의 충돌을 찾는 것이 실용적이더라도. 그러나, 무작위 해싱을 사용하면 메시지의 모든 부분을 서명자가 준비할 때 디지털 서명이 제공하는 보안의 양을 줄일 수 있습니다.
>
> (NIST SP-800-106 "Randomized Hashing for Digital Signatures")

BLAKE2에서 솔트는 각 압축 함수에 대한 입력이 아니라 초기화 중에 해시 함수에 대한 일회성 입력으로 처리됩니다.

경고

BLAKE2나 SHA-256과 같은 기타 범용 암호화 해시 함수를 사용하는 솔트 해싱(*salted hashing*)(또는 그냥 해싱)은 암호(password) 해싱에 적합하지 않습니다. 자세한 정보는 BLAKE2 FAQ를 참조하십시오.

```
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
>>> h1 = blake2b(salt=salt1)
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

개인화

때로는 해시 함수가 다른 목적으로 같은 입력에 대해 다른 요약을 생성하도록 강제하는 것이 유용합니다. Skein 해시 함수의 저자를 인용합니다:

> 모든 응용 프로그램 설계자는 이렇게 하는 것을 진지하게 고려하도록 권장합니다; 우리는 유사하거나 관련된 데이터에 대해 두 해시 계산이 수행되었기 때문에, 프로토콜의 한 부분에서 계산된 해시가 완전히 다른 부분에서 사용될 수 있는 프로토콜을 많이 보았으며, 공격자는 응용 프로그램이 해시 입력을 갖게 만들도록 강제할 수 있습니다. 프로토콜에 사용된 각 해시 함수를 개인화하면 이러한 유형의 공격이 중단됩니다.

(The Skein Hash Function Family, p. 21)

BLAKE2는 바이트열을 *person* 인자에 전달하여 개인화할 수 있습니다:

```
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9aea52264a80b75005e65619778de59f383a3'
```

키 모드와 함께 개인화를 사용하여, 한 키에서 다른 키들을 파생시킬 수도 있습니다.

```
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key = b64decode(b'Rm5EPJai72qcK3RGBpW3vPNfZy5OZothY+kHY6h21KM=')
>>> enc_key = blake2s(key=orig_key, person=b'kEncrypt').digest()
>>> mac_key = blake2s(key=orig_key, person=b'kMAC').digest()
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuhB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWPlYk1e/nWfu0WSEb0KRcjhDeP/o=
```

트리 모드

다음은 두 개의 리프 노드를 갖는 최소 트리를 해싱하는 예입니다:

```
10
 /  \
00  01
```

이 예는 64바이트 내부 요약을 사용하고, 32바이트 최종 요약을 반환합니다:

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
>>> buf = bytearray(6000)
>>>
>>> # 왼쪽 리프
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=0, last_node=False)
>>> # 오른쪽 리프
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=1, node_depth=0, last_node=True)
>>> # 루트 노드
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...                 node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'
```

크레딧

BLAKE2는 *Jean-Philippe Aumasson*, *Luca Henzen*, *Willi Meier* 및 *Raphael C.-W. Phan*이 만든 SHA-3 파이널리스트 BLAKE를 기반으로 *Jean-Philippe Aumasson*, *Samuel Neves*, *Zooko Wilcox-O' Hearn* 및 *Christian Winnerlein*이 설계했습니다.

*Daniel J. Bernstein*이 설계한 ChaCha 암호(cipher)의 핵심 알고리즘을 사용합니다.

표준 라이브러리 구현은 pyblake2 모듈에 기반합니다. 이것은 *Samuel Neves*가 작성한 C 구현을 기반으로 *Dmitry Chestnykh*가 작성했습니다. 이 설명서는 pyblake2에서 복사했으며 *Dmitry Chestnykh*가 작성했습니다.

C 코드는 *Christian Heimes*가 파이썬 용으로 부분적으로 재작성했습니다.

다음 공개 도메인 기부는 C 해시 함수 구현, 확장 코드 및 이 설명서 모두에 적용됩니다:

> 법률에 따라 가능한 범위 내에서, 저자(들)는 이 소프트웨어에 대한 모든 저작권과 관련되고 둘러싼 권리를 전 세계 공개 도메인에 기부했습니다. 이 소프트웨어는 보증 없이 배포됩니다.
>
> 이 소프트웨어와 함께 CC0 Public Domain Dedication의 사본을 받았어야 합니다. 그렇지 않으면, https://creativecommons.org/publicdomain/zero/1.0/ 을 참조하십시오.

다음과 같은 사람들은 Creative Commons Public Domain Dedication 1.0 Universal에 따라 개발을 돕거나 프로젝트와 공개 도메인에 변경에 기여했습니다:

- *Alexandr Sokolovskiy*

더 보기

모듈 `hmac`
해시를 사용하여 메시지 인증 코드를 생성하는 모듈.

모듈 `base64`
바이너리가 아닌 환경을 위해 바이너리 해시를 인코딩하는 다른 방법.

https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf
Secure Hash Algorithms에 관한 FIPS 180-4 발행물.

https://csrc.nist.gov/pubs/fips/202/final
SHA-3 표준에 관한 FIPS 202 발행물.

https://www.blake2.net/
공식 BLAKE2 웹 사이트.

https://en.wikipedia.org/wiki/Cryptographic_hash_function
어떤 알고리즘에 알려진 문제가 있고 그것이 사용에 어떤 의미가 있는지에 대한 정보가 포함된 위키피디아 기사.

https://www.ietf.org/rfc/rfc8018.txt
PKCS #5: Password-Based Cryptography Specification Version 2.1

https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf
NIST Recommendation for Password-Based Key Derivation.

15.2 hmac — 메시지 인증을 위한 키 해싱

소스 코드: Lib/hmac.py

This module implements the HMAC algorithm as described by **RFC 2104**. The interface allows to use any hash function with a *fixed* digest size. In particular, extendable output functions such as SHAKE-128 or SHAKE-256 cannot be used with HMAC.

hmac.**new**(*key*, *msg=None*, *digestmod*)

새로운 hmac 객체를 반환합니다. *key*는 비밀 키를 제공하는 바이트열이나 바이트 배열(bytearray) 객체입니다. *msg*가 있으면, `update(msg)` 메서드 호출이 수행됩니다. *digestmod*는 다이제스트 이름, 다이제스트 생성자 또는 HMAC 객체가 사용할 모듈입니다. `hashlib.new()`에 적합한 모든 이름일 수 있습니다. 인자의 위치에도 불구하고, 필수입니다.

버전 3.4에서 변경: 매개 변수 *key*는 바이트열 또는 바이트 배열 객체일 수 있습니다. 매개 변수 *msg*는 `hashlib`가 지원하는 모든 형이 될 수 있습니다. 매개 변수 *digestmod*는 해시 알고리즘의 이름이 될 수 있습니다.

버전 3.8에서 변경: *digestmod* 인자는 이제 필수입니다. 초기 *msg*가 없을 때 어색함을 피하려면 키워드 인자로 전달하십시오.

hmac.**digest**(*key*, *msg*, *digest*)

주어진 비밀 *key*와 *digest*로 *msg*의 다이제스트를 반환합니다. 이 함수는 `HMAC(key, msg, digest).digest()`와 동등하지만, 최적화된 C 나 인라인 구현을 사용해서, 메모리에 맞는 메시지에는 더 빠릅니다. 매개 변수 *key*, *msg* 및 *digest*는 `new()`에서와 같은 뜻입니다.

CPython 구현 세부 사항, 최적화된 C 구현은 *digest*가 문자열이고 OpenSSL에서 지원하는 다이제스트 알고리즘의 이름일 때만 사용됩니다.

Added in version 3.7.

class hmac.**HMAC**

HMAC 객체에는 다음과 같은 메서드가 있습니다:

HMAC.**update**(*msg*)

hmac 객체를 *msg*로 갱신합니다. 반복되는 호출은 모든 인자를 이어붙인 단일 호출과 동등합니다: `m.update(a); m.update(b)`는 `m.update(a + b)`와 동등합니다.

버전 3.4에서 변경: 매개 변수 *msg*는 `hashlib`가 지원하는 모든 형이 될 수 있습니다.

HMAC.**digest**()

지금까지 `update()` 메서드로 전달된 바이트들의 다이제스트를 반환합니다. 이 바이트열 객체는 생성자에게 주어진 다이제스트의 *digest_size*와 길이가 같습니다. NUL 바이트를 포함하여 비 ASCII 바이트를 포함할 수 있습니다.

> **경고**
>
> 검증 루틴에서 `digest()`의 출력을 외부에서 제공되는 다이제스트와 비교할 때, `==` 연산자 대신 `compare_digest()` 함수를 사용하여 타이밍 공격의 취약점을 줄이는 것이 좋습니다.

HMAC.**hexdigest**()

다이제스트가 16진수만 포함하는 길이가 두 배인 문자열로 반환된다는 점을 제외하고는 `digest()`와 같습니다. 이것은 전자 메일이나 기타 비 바이너리 환경에서 값을 안전하게 교환하는 데 사용될 수 있습니다.

> **경고**
>
> 검증 루틴에서 `hexdigest()`의 출력을 외부에서 제공되는 다이제스트와 비교할 때, `==` 연산자 대신 `compare_digest()` 함수를 사용하여 타이밍 공격의 취약점을 줄이는 것이 좋습니다.

`HMAC.copy()`

hmac 객체의 복사본("클론")을 반환합니다. 이것은 공통 초기 부분 문자열을 공유하는 문자열들의 다이제스트를 효율적으로 계산하는 데 사용할 수 있습니다.

hmac 객체에는 다음과 같은 어트리뷰트가 있습니다:

`HMAC.digest_size`

결과 HMAC 다이제스트의 크기(바이트).

`HMAC.block_size`

해시 알고리즘의 내부 블록 크기(바이트).

Added in version 3.4.

`HMAC.name`

이 HMAC의 규범적 이름, 항상 소문자, 예를 들어 `hmac-md5`.

Added in version 3.4.

버전 3.10에서 변경: 설명되지 않은 어트리뷰트 `HMAC.digest_cons`, `HMAC.inner` 및 `HMAC.outer`를 제거했습니다.

이 모듈은 또한 다음 도우미 함수를 제공합니다:

`hmac.compare_digest(a, b)`

`a == b`를 반환합니다. 이 함수는 내용 기반의 단락(short circuiting) 동작을 피함으로써 타이밍 분석을 방지하도록 설계된 접근법을 사용해서 암호화에 적합하게 만듭니다. *a*와 *b*는 모두 같은 형이어야 합니다: *str* (ASCII만, 예를 들어 *HMAC.hexdigest()*에 의해 반환된 것과 같은 것) 이나 바이트열류 객체.

> **참고**
>
> *a*와 *b*의 길이가 다르거나 에러가 발생하면, 타이밍 공격이 이론적으로는 *a* 와 *b*의 형과 길이에 관한 정보를 드러낼 수 있습니다 - 하지만 그 값은 아닙니다.

Added in version 3.3.

버전 3.10에서 변경: 이 함수는 사용할 수 있으면 내부적으로 OpenSSL의 `CRYPTO_memcmp()`를 사용합니다.

> **더 보기**
>
> **모듈 *hashlib***
> 안전한 해시 함수를 제공하는 파이썬 모듈.

15.3 secrets — 비밀 관리를 위한 안전한 난수 생성

Added in version 3.6.

소스 코드: Lib/secrets.py

---

secrets 모듈은 암호, 계정 인증, 보안 토큰 및 관련 비밀과 같은 데이터를 관리하는 데 적합한 암호학적으로 강력한 난수를 생성하는 데 사용됩니다.

특히, *secrets*는 보안이나 암호화가 아닌 모델링과 시뮬레이션용으로 설계된 *random* 모듈의 기본 의사 난수 생성기보다 먼저 사용해야 합니다.

더 보기

PEP 506

15.3.1 난수

secrets 모듈은 운영 체제가 제공하는 가장 안전한 무작위 소스에 대한 액세스를 제공합니다.

class secrets.**SystemRandom**

운영 체제에서 제공하는 최고 품질의 소스를 사용하여 난수를 생성하는 클래스. 자세한 내용은 *random.SystemRandom*를 참조하십시오.

secrets.**choice**(*seq*)

비어있지 않은 시퀀스로부터 무작위로 선택된 요소를 돌려줍니다.

secrets.**randbelow**(*exclusive_upper_bound*)

범위 [0, *exclusive_upper_bound*)에서 무작위 int를 돌려줍니다.

secrets.**randbits**(*k*)

k 무작위 비트를 가지는 음이 아닌 int를 돌려줍니다.

15.3.2 토큰 생성

secrets 모듈은 암호 재설정, 추측하기 어려운 URL 등과 같은 응용에 적합한 보안 토큰을 생성하는 함수를 제공합니다.

secrets.**token_bytes**([*nbytes=None*])

nbytes 바이트를 포함하는 임의의 바이트열을 반환합니다. *nbytes*가 `None`이거나 제공되지 않으면, 적절한 기본값이 사용됩니다.

```
>>> token_bytes(16)
b'\xebr\x17D*t\xae\xd4\xe3S\xb6\xe2\xebP1\x8b'
```

secrets.**token_hex**([*nbytes=None*])

무작위 16진수 텍스트 문자열을 돌려줍니다. 이 문자열에는 *nbytes* 무작위 바이트가 있으며, 각 바이트는 두 자리 16진수로 변환됩니다. *nbytes*가 `None`이거나 제공되지 않으면, 적절한 기본값이 사용됩니다.

```
>>> token_hex(16)
'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

secrets.**token_urlsafe**([*nbytes=None*])

*nbytes*의 무작위 바이트를 포함한, URL 안전한 무작위 텍스트 문자열을 돌려줍니다. 텍스트는 Base64로 인코딩되어 있으므로, 평균적으로 각 바이트는 약 1.3 문자가 됩니다. *nbytes*가 `None`이거나 제공되지 않으면, 적절한 기본값이 사용됩니다.

```
>>> token_urlsafe(16)
'Drmhze6EPcv0fN_81Bj-nA'
```

토큰은 몇 바이트를 사용해야 합니까?

무차별 공격으로부터 안전하려면, 토큰에 충분한 무작위성이 있어야 합니다. 불행하게도, 컴퓨터가 더 강력해지고 더 짧은 기간에 더 많은 추측을 할 수 있게 됨에 따라, 충분하다고 여겨지는 것은 필연적으로 증가합니다. 2015년 현재, *secrets* 모듈로 예상되는 일반적인 사용 사례에는 32바이트(256비트)의 무작위성으로 충분하다고 여겨집니다.

자신의 토큰 길이를 관리하려는 사용자는, 여러 `token_*` 함수에 *int* 인자를 제공하여 토큰에 사용되는 무작위성의 양을 명시적으로 지정할 수 있습니다. 이 인자는 사용할 무작위성의 바이트 수로 사용됩니다.

그렇지 않으면, 인자가 제공되지 않거나 인자가 `None` 이면, `token_*` 함수는 적절한 기본값을 대신 사용합니다.

> **참고**
>
> 이 기본값은 유지 보수 배포를 포함하여 언제든지 변경될 수 있습니다.

15.3.3 기타 함수

`secrets.compare_digest`(*a*, *b*)

문자열 혹은 바이트열류 객체 *a*와 *b*가 같으면 `True`를, 그렇지 않으면 `False`를 반환하는데, "고정 시간 비교(constant-time compare)"를 사용하여 타이밍 공격의 위험을 줄입니다. 자세한 내용은 *hmac.compare_digest()*를 참조하십시오.

15.3.4 조리법과 모범 사례

이 절에서는 *secrets*를 사용하여 기본 보안 수준을 관리하는 조리법과 모범 사례를 보여줍니다.

8문자 영숫자 암호 생성합니다:

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
password = ''.join(secrets.choice(alphabet) for i in range(8))
```

> **참고**
>
> 응용 프로그램은 평문인지 암호문인지 관계없이, 암호를 복원 가능한 형식으로 저장해서는 안 됩니다. 이들은 암호학적으로 강력한 단방향(비가역적) 해시 함수를 사용하여 솔트되고 해시 되어야 합니다.

적어도 하나의 소문자, 적어도 하나의 대문자 및 적어도 3개의 숫자가 있는 10자의 영숫자 암호를 생성합니다:

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(secrets.choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

XKCD-스타일 암호문을 생성합니다:

```
import secrets
# 표준 리눅스 시스템에서는, 편리한 딕셔너리 파일을 사용하십시오.
# 다른 플랫폼에서는, 자신의 단어 목록을 제공해야 할 수도 있습니다.
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(secrets.choice(words) for i in range(4))
```

암호 복구 응용에 적합한 보안 토큰을 포함하는 추측하기 어려운 임시 URL을 생성합니다:

```
import secrets
url = 'https://example.com/reset=' + secrets.token_urlsafe()
```

CHAPTER 16

일반 운영 체제 서비스

이 장에서 설명하는 모듈은 파일 및 시계와 같은 (거의) 모든 운영 체제에서 사용할 수 있는 운영 체제 기능에 대한 인터페이스를 제공합니다. 인터페이스는 일반적으로 유닉스 또는 C 인터페이스를 모델로 하지만, 대부분의 다른 시스템에서도 사용할 수 있습니다. 다음은 개요입니다:

16.1 os — Miscellaneous operating system interfaces

소스 코드: Lib/os.py

이 모듈은 운영 체제 종속 기능을 사용하는 이식성 있는 방법을 제공합니다. 파일을 읽거나 쓰고 싶으면 `open()` 을 보세요, 경로를 조작하려면 `os.path` 모듈을 보시고, 명령 줄에서 주어진 모든 파일의 모든 줄을 읽으려면 `fileinput` 모듈을 보십시오. 임시 파일과 디렉터리를 만들려면 `tempfile` 모듈을 보시고, 고수준의 파일과 디렉터리 처리는 `shutil` 모듈을 보십시오.

이러한 기능의 가용성에 대한 참고 사항:

- 내장된 모든 운영 체제 종속적인 파이썬 모듈의 설계는, 같은 기능을 사용할 수 있는 한, 같은 인터페이스를 사용합니다; 예를 들어, 함수 `os.stat(path)` 는 *path* 에 대한 stat 정보를 같은 (POSIX 인터페이스에서 기원한) 형식으로 반환합니다.
- 특정 운영 체제에 고유한 확장도 `os` 모듈을 통해서 사용할 수 있지만, 이러한 기능을 사용하는 것은 물론 이식성에 대한 위협입니다.
- 경로 또는 파일명을 받아들이는 모든 함수는 바이트열과 문자열 객체를 모두 허용하며, 경로나 파일명이 반환되면 같은 형의 객체를 반환합니다.
- On VxWorks, os.popen, os.fork, os.execv and os.spawn*p* are not supported.
- On WebAssembly platforms, Android and iOS, large parts of the `os` module are not available or behave differently. APIs related to processes (e.g. `fork()`, `execve()`) and resources (e.g. `nice()`) are not available. Others like `getuid()` and `getpid()` are emulated or stubs. WebAssembly platforms also lack support for signals (e.g. `kill()`, `wait()`).

> **참고**
>
> 이 모듈의 모든 함수는, 올바르지 않거나 액세스할 수 없는 파일명과 경로일 때, 또는 올바른 형의 인자이지만, 운영 체제에서 허용하지 않으면 `OSError`(또는 이것의 서브 클래스)를 발생시킵니다.

exception os.**error**

내장 *OSError* 예외의 별칭.

os.**name**

임포트 된 운영 체제 종속 모듈의 이름. 다음과 같은 이름이 현재 등록되어있습니다: 'posix', 'nt', 'java'.

> **더 보기**
>
> *sys.platform* has a finer granularity. *os.uname()* gives system-dependent version information.
>
> *platform* 모듈은 시스템의 아이덴티티에 대한 자세한 검사를 제공합니다.

16.1.1 파일명, 명령 줄 인자 및 환경 변수

In Python, file names, command line arguments, and environment variables are represented using the string type. On some systems, decoding these strings to and from bytes is necessary before passing them to the operating system. Python uses the *filesystem encoding and error handler* to perform this conversion (see *sys.getfilesystemencoding()*).

The *filesystem encoding and error handler* are configured at Python startup by the PyConfig_Read() function: see filesystem_encoding and filesystem_errors members of PyConfig.

버전 3.1에서 변경: On some systems, conversion using the file system encoding may fail. In this case, Python uses the *surrogateescape encoding error handler*, which means that undecodable bytes are replaced by a Unicode character U+DC*xx* on decoding, and these are again translated to the original byte on encoding.

The *file system encoding* must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise *UnicodeError*.

See also the *locale encoding*.

16.1.2 Python UTF-8 Mode

Added in version 3.7: See **PEP 540** for more details.

The Python UTF-8 Mode ignores the *locale encoding* and forces the usage of the UTF-8 encoding:

- Use UTF-8 as the *filesystem encoding*.
- *sys.getfilesystemencoding()* returns 'utf-8'.
- *locale.getpreferredencoding()* returns 'utf-8' (the *do_setlocale* argument has no effect).
- *sys.stdin*, *sys.stdout*, and *sys.stderr* all use UTF-8 as their text encoding, with the surrogateescape *error handler* being enabled for *sys.stdin* and *sys.stdout* (*sys.stderr* continues to use backslashreplace as it does in the default locale-aware mode)
- On Unix, *os.device_encoding()* returns 'utf-8' rather than the device encoding.

Note that the standard stream settings in UTF-8 mode can be overridden by PYTHONIOENCODING (just as they can be in the default locale-aware mode).

As a consequence of the changes in those lower level APIs, other higher level APIs also exhibit different default behaviours:

- Command line arguments, environment variables and filenames are decoded to text using the UTF-8 encoding.
- *os.fsdecode()* and *os.fsencode()* use the UTF-8 encoding.
- *open()*, *io.open()*, and *codecs.open()* use the UTF-8 encoding by default. However, they still use the strict error handler by default so that attempting to open a binary file in text mode is likely to raise an exception rather than producing nonsense data.

The *Python UTF-8 Mode* is enabled if the LC_CTYPE locale is `C` or `POSIX` at Python startup (see the `PyConfig_Read()` function).

It can be enabled or disabled using the `-X utf8` command line option and the `PYTHONUTF8` environment variable.

If the `PYTHONUTF8` environment variable is not set at all, then the interpreter defaults to using the current locale settings, *unless* the current locale is identified as a legacy ASCII-based locale (as described for `PYTHONCOERCECLOCALE`), and locale coercion is either disabled or fails. In such legacy locales, the interpreter will default to enabling UTF-8 mode unless explicitly instructed not to do so.

The Python UTF-8 Mode can only be enabled at the Python startup. Its value can be read from `sys.flags.utf8_mode`.

See also the UTF-8 mode on Windows and the *filesystem encoding and error handler*.

더 보기

PEP 686
Python 3.15 will make *Python UTF-8 Mode* default.

16.1.3 프로세스 매개 변수

이 함수들과 데이터 항목은 현재 프로세스와 사용자에 관한 정보와 관련 연산을 제공합니다.

`os.ctermid()`

프로세스의 제어 터미널에 해당하는 파일명을 반환합니다.

가용성: Unix, not WASI.

`os.environ`

A *mapping* object where keys and values are strings that represent the process environment. For example, `environ['HOME']` is the pathname of your home directory (on some platforms), and is equivalent to `getenv("HOME")` in C.

This mapping is captured the first time the `os` module is imported, typically during Python startup as part of processing `site.py`. Changes to the environment made after this time are not reflected in `os.environ`, except for changes made by modifying `os.environ` directly.

이 매핑은 환경을 조회하는 것뿐 아니라 환경을 수정하는 데도 사용될 수 있습니다. 매핑이 수정될 때 `putenv()`가 자동으로 호출됩니다.

유닉스에서, 키와 값은 `sys.getfilesystemencoding()` 과 `'surrogateescape'` 에러 처리기를 사용합니다. 다른 인코딩을 사용하려면 `environb`를 사용하십시오.

On Windows, the keys are converted to uppercase. This also applies when getting, setting, or deleting an item. For example, `environ['monty'] = 'python'` maps the key `'MONTY'` to the value `'python'`.

참고

Calling `putenv()` directly does not change `os.environ`, so it' s better to modify `os.environ`.

참고

On some platforms, including FreeBSD and macOS, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv()`.

You can delete items in this mapping to unset environment variables. `unsetenv()` will be called automatically when an item is deleted from `os.environ`, and when one of the `pop()` or `clear()` methods is called.

> **더 보기**
>
> The `os.reload_environ()` function.

버전 3.9에서 변경: **PEP 584**의 병합(|)과 업데이트(|=) 연산자를 지원하도록 갱신되었습니다.

os.**environb**

Bytes version of `environ`: a *mapping* object where both keys and values are `bytes` objects representing the process environment. `environ` and `environb` are synchronized (modifying `environb` updates `environ`, and vice versa).

`environb` is only available if `supports_bytes_environ` is `True`.

Added in version 3.2.

버전 3.9에서 변경: **PEP 584**의 병합(|)과 업데이트(|=) 연산자를 지원하도록 갱신되었습니다.

os.**reload_environ**()

The `os.environ` and `os.environb` mappings are a cache of environment variables at the time that Python started. As such, changes to the current process environment are not reflected if made outside Python, or by `os.putenv()` or `os.unsetenv()`. Use `os.reload_environ()` to update `os.environ` and `os.environb` with any such changes to the current process environment.

> **경고**
>
> This function is not thread-safe. Calling it while the environment is being modified in an other thread is an undefined behavior. Reading from `os.environ` or `os.environb`, or calling `os.getenv()` while reloading, may return an empty result.

Added in version 3.14.

os.**chdir**(*path*)

os.**fchdir**(*fd*)

os.**getcwd**()

이 함수는 파일과 디렉터리에 설명되어 있습니다.

os.**fsencode**(*filename*)

Encode *path-like filename* to the *filesystem encoding and error handler*; return `bytes` unchanged.

`fsdecode()`는 역 함수입니다.

Added in version 3.2.

버전 3.6에서 변경: `os.PathLike` 인터페이스를 구현하는 객체를 받아들이도록 지원이 추가되었습니다.

os.**fsdecode**(*filename*)

Decode the *path-like filename* from the *filesystem encoding and error handler*; return `str` unchanged.

`fsencode()`는 역 함수입니다.

Added in version 3.2.

버전 3.6에서 변경: `os.PathLike` 인터페이스를 구현하는 객체를 받아들이도록 지원이 추가되었습니다.

os.**fspath**(*path*)

경로의 파일 시스템 표현을 돌려줍니다.

`str`이나 `bytes`가 전달되면, 변경되지 않은 상태로 반환됩니다. 그렇지 않으면 `__fspath__()`가 호출되고, 해당 값이 `str`이나 `bytes` 객체인 한 그 값이 반환됩니다. 다른 모든 경우에는 `TypeError`가 발생합니다.

Added in version 3.6.

class os.**PathLike**

파일 시스템 경로를 나타내는 객체(예를 들어 *pathlib.PurePath*)의 추상 베이스 클래스입니다.

Added in version 3.6.

abstractmethod **__fspath__**()

객체의 파일 시스템 경로 표현을 돌려줍니다.

이 메서드는 *str*이나 *bytes* 객체만 반환해야 하며, *str*을 선호합니다.

os.**getenv**(*key*, *default=None*)

Return the value of the environment variable *key* as a string if it exists, or *default* if it doesn' t. *key* is a string. Note that since *getenv()* uses *os.environ*, the mapping of *getenv()* is similarly also captured on import, and the function may not reflect future environment changes.

유닉스에서, 키와 값은 *sys.getfilesystemencoding()* 과 'surrogateescape' 에러 처리기로 디코딩됩니다. 다른 인코딩을 사용하려면 *os.getenvb()*를 사용하십시오.

가용성: Unix, Windows.

os.**getenvb**(*key*, *default=None*)

Return the value of the environment variable *key* as bytes if it exists, or *default* if it doesn' t. *key* must be bytes. Note that since *getenvb()* uses *os.environb*, the mapping of *getenvb()* is similarly also captured on import, and the function may not reflect future environment changes.

getenvb() is only available if *supports_bytes_environ* is True.

가용성: Unix.

Added in version 3.2.

os.**get_exec_path**(*env=None*)

셸과 비슷하게, 프로세스를 시작할 때 지정된 이름의 실행 파일을 검색할 디렉터리 리스트를 반환합니다. (지정된다면) *env* 는 PATH를 조회할 환경 변수 딕셔너리 여야 합니다. 기본적으로, *env* 가 None 이면, *environ*이 사용됩니다.

Added in version 3.2.

os.**getegid**()

현재 프로세스의 유효(effective) 그룹 ID를 반환합니다. 이것은 현재 프로세스에서 실행 중인 파일의 "set id" 비트에 해당합니다.

가용성: Unix, not WASI.

os.**geteuid**()

현재 프로세스의 유효(effective) 사용자 ID를 반환합니다.

가용성: Unix, not WASI.

os.**getgid**()

현재 프로세스의 실제(real) 그룹 ID를 반환합니다.

가용성: Unix.

The function is a stub on WASI, see 웹어셈블리 플랫폼 for more information.

os.**getgrouplist**(*user*, *group*, /)

Return list of group ids that *user* belongs to. If *group* is not in the list, it is included; typically, *group* is specified as the group ID field from the password record for *user*, because that group ID will otherwise be potentially omitted.

가용성: Unix, not WASI.

Added in version 3.3.

os.**getgroups**()

현재 프로세스와 관련된 보충(supplemental) 그룹 ID 목록을 반환합니다.

가용성: Unix, not WASI.

> **참고**
>
> On macOS, *getgroups()* behavior differs somewhat from other Unix platforms. If the Python interpreter was built with a deployment target of `10.5` or earlier, *getgroups()* returns the list of effective group ids associated with the current user process; this list is limited to a system-defined number of entries, typically 16, and may be modified by calls to *setgroups()* if suitably privileged. If built with a deployment target greater than `10.5`, *getgroups()* returns the current group access list for the user associated with the effective user id of the process; the group access list may change over the lifetime of the process, it is not affected by calls to *setgroups()*, and its length is not limited to 16. The deployment target value, `MACOSX_DEPLOYMENT_TARGET`, can be obtained with *sysconfig.get_config_var()*.

os.**getlogin**()

프로세스의 제어 터미널에 로그인한 사용자의 이름을 반환합니다. 대부분 목적에서, *getpass.getuser()*를 사용하는 것이 더 유용한데, 이 함수는 환경 변수 `LOGNAME` 이나 `USERNAME`을 검사하여 사용자가 누구인지 알아내고, 현재 실제 사용자 ID의 로그인 이름을 얻기 위해 `pwd.getpwuid(os.getuid())[0]`로 폴백 하기 때문입니다.

가용성: Unix, Windows, not WASI.

os.**getpgid**(*pid*)

프로세스 ID *pid* 를 갖는 프로세스의 프로세스 그룹 ID를 반환합니다. *pid* 가 0이면, 현재 프로세스의 프로세스 그룹 id가 반환됩니다.

가용성: Unix, not WASI.

os.**getpgrp**()

현재 프로세스 그룹의 ID를 반환합니다.

가용성: Unix, not WASI.

os.**getpid**()

현재의 프로세스 ID를 반환합니다.

The function is a stub on WASI, see 웹어셈블리 플랫폼 for more information.

os.**getppid**()

부모의 프로세스 ID를 반환합니다. 부모 프로세스가 종료했으면, 유닉스에서 반환된 id는 init 프로세스 (1) 중 하나이며, 윈도우에서는 여전히 같은 id인데, 다른 프로세스에서 이미 재사용했을 수 있습니다.

가용성: Unix, Windows, not WASI.

버전 3.2에서 변경: 윈도우에 대한 지원이 추가되었습니다.

os.**getpriority**(*which*, *who*)

프로그램 스케줄 우선순위를 얻습니다. *which* 값은 *PRIO_PROCESS*, *PRIO_PGRP* 또는 *PRIO_USER* 중 하나이고, *who*는 *which* 에 상대적으로 해석됩니다 (*PRIO_PROCESS* 면 프로세스 식별자, *PRIO_PGRP* 면 프로세스 그룹 식별자, *PRIO_USER* 면 사용자 ID). 0 값의 *who*는 (각각) 호출하는 프로세스, 호출하는 프로세스의 프로세스 그룹, 호출하는 프로세스의 실제 사용자 ID를 나타냅니다.

가용성: Unix, not WASI.

Added in version 3.3.

os.**PRIO_PROCESS**

os.**PRIO_PGRP**

os.**PRIO_USER**

getpriority() 와 *setpriority()* 함수의 매개 변숫값

가용성: Unix, not WASI.

Added in version 3.3.

os.**PRIO_DARWIN_THREAD**

os.**PRIO_DARWIN_PROCESS**

os.**PRIO_DARWIN_BG**

os.**PRIO_DARWIN_NONUI**

getpriority() 와 *setpriority()* 함수의 매개 변숫값

가용성: macOS

Added in version 3.12.

os.**getresuid**()

현재 프로세스의 실제(real), 유효(effective) 및 저장된(saved) 사용자 ID를 나타내는 튜플 (ruid, euid, suid)를 반환합니다.

가용성: Unix, not WASI.

Added in version 3.2.

os.**getresgid**()

현재 프로세스의 실제(real), 유효(effective) 및 저장된(saved) 그룹 ID를 나타내는 튜플 (rgid, egid, sgid)를 반환합니다.

가용성: Unix, not WASI.

Added in version 3.2.

os.**getuid**()

현재 프로세스의 실제(real) 사용자 ID를 반환합니다.

가용성: Unix.

The function is a stub on WASI, see 웹어셈블리 플랫폼 for more information.

os.**initgroups**(*username, gid, /*)

지정된 사용자 이름이 구성원인 모든 그룹과 지정된 그룹 ID로 구성된 그룹 액세스 목록을 초기화하기 위해 시스템 initgroups()를 호출합니다.

가용성: Unix, not WASI, not Android.

Added in version 3.2.

os.**putenv**(*key, value, /*)

key 라는 환경 변수를 문자열 *value* 로 설정합니다. 이러한 환경의 변화는 *os.system()*, *popen()* 또는 *fork()* 및 *execv()*로 시작된 자식 프로세스에 영향을 줍니다.

Assignments to items in *os.environ* are automatically translated into corresponding calls to *putenv()*; however, calls to *putenv()* don' t update *os.environ*, so it is actually preferable to assign to items of *os.environ*. This also applies to *getenv()* and *getenvb()*, which respectively use *os.environ* and *os.environb* in their implementations.

See also the *os.reload_environ()* function.

> **참고**
>
> On some platforms, including FreeBSD and macOS, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv()`.

key, value를 인자로 감사 이벤트(*auditing event*) os.putenv를 발생시킵니다.

버전 3.9에서 변경: 이 함수는 이제 항상 사용할 수 있습니다.

os.**setegid**(*egid, /*)

현재 프로세스의 유효 그룹 ID를 설정합니다.

가용성: Unix, not WASI, not Android.

os.**seteuid**(*euid, /*)

현재 프로세스의 유효 사용자 ID를 설정합니다.

가용성: Unix, not WASI, not Android.

os.**setgid**(*gid, /*)

현재 프로세스의 그룹 ID를 설정합니다.

가용성: Unix, not WASI, not Android.

os.**setgroups**(*groups, /*)

현재 프로세스와 연관된 보충(supplemental) 그룹 ID의 목록을 *groups*로 설정합니다. *groups* 는 시퀀스여야 하며, 각 요소는 그룹을 식별하는 정수여야 합니다. 이 연산은 대개 슈퍼 유저만 사용할 수 있습니다.

가용성: Unix, not WASI.

> **참고**
>
> On macOS, the length of *groups* may not exceed the system-defined maximum number of effective group ids, typically 16. See the documentation for `getgroups()` for cases where it may not return the same group list set by calling setgroups().

os.**setns**(*fd, nstype=0*)

Reassociate the current thread with a Linux namespace. See the `setns(2)` and `namespaces(7)` man pages for more details.

If *fd* refers to a `/proc/pid/ns/` link, `setns()` reassociates the calling thread with the namespace associated with that link, and *nstype* may be set to one of the *CLONE_NEW* constants* to impose constraints on the operation (`0` means no constraints).

Since Linux 5.8, *fd* may refer to a PID file descriptor obtained from `pidfd_open()`. In this case, `setns()` reassociates the calling thread into one or more of the same namespaces as the thread referred to by *fd*. This is subject to any constraints imposed by *nstype*, which is a bit mask combining one or more of the *CLONE_NEW* constants*, e.g. `setns(fd, os.CLONE_NEWUTS | os.CLONE_NEWPID)`. The caller' s memberships in unspecified namespaces are left unchanged.

fd can be any object with a `fileno()` method, or a raw file descriptor.

This example reassociates the thread with the `init` process' s network namespace:

```
fd = os.open("/proc/1/ns/net", os.O_RDONLY)
os.setns(fd, os.CLONE_NEWNET)
os.close(fd)
```

가용성: Linux >= 3.0 with glibc >= 2.14.

Added in version 3.12.

> **더 보기**
>
> The `unshare()` function.

os.**setpgrp**()

Call the system call `setpgrp()` or `setpgrp(0, 0)` depending on which version is implemented (if any). See the Unix manual for the semantics.

가용성: Unix, not WASI.

os.**setpgid**(*pid, pgrp, /*)

Call the system call `setpgid()` to set the process group id of the process with id *pid* to the process group with id *pgrp*. See the Unix manual for the semantics.

가용성: Unix, not WASI.

os.**setpriority**(*which, who, priority*)

프로그램 스케줄 우선순위를 설정합니다. *which* 값은 `PRIO_PROCESS`, `PRIO_PGRP` 또는 `PRIO_USER` 중 하나이고, *who*는 *which* 에 상대적으로 해석됩니다 (`PRIO_PROCESS` 면 프로세스 식별자, `PRIO_PGRP` 면 프로세스 그룹 식별자, `PRIO_USER` 면 사용자 ID). 0 값의 *who*는 (각각) 호출하는 프로세스, 호출하는 프로세스의 프로세스 그룹, 호출하는 프로세스의 실제 사용자 ID를 나타냅니다. *priority* 는 -20 에서 19 사이의 값입니다. 기본 우선순위는 0입니다; 우선순위가 낮으면 더 유리하게 스케줄 됩니다.

가용성: Unix, not WASI.

Added in version 3.3.

os.**setregid**(*rgid, egid, /*)

현재 프로세스의 실제(real) 및 유효한(effective) 그룹 ID를 설정합니다.

가용성: Unix, not WASI, not Android.

os.**setresgid**(*rgid, egid, sgid, /*)

현재 프로세스의 실제(real), 유효(effective) 및 저장된(saved) 그룹 ID를 설정합니다.

가용성: Unix, not WASI, not Android.

Added in version 3.2.

os.**setresuid**(*ruid, euid, suid, /*)

현재 프로세스의 실제(real), 유효(effective) 및 저장된(saved) 사용자 ID를 설정합니다.

가용성: Unix, not WASI, not Android.

Added in version 3.2.

os.**setreuid**(*ruid, euid, /*)

현재 프로세스의 실제(real) 및 유효(effective) 사용자 ID를 설정합니다.

가용성: Unix, not WASI, not Android.

os.**getsid**(*pid, /*)

Call the system call `getsid()`. See the Unix manual for the semantics.

가용성: Unix, not WASI.

os.**setsid**()

Call the system call `setsid()`. See the Unix manual for the semantics.

가용성: Unix, not WASI.

os.**setuid**(*uid, /*)

현재 프로세스의 사용자 ID를 설정합니다.

가용성: Unix, not WASI, not Android.

os.**strerror**(*code, /*)

Return the error message corresponding to the error code in *code*. On platforms where `strerror()` returns `NULL` when given an unknown error number, `ValueError` is raised.

os.**supports_bytes_environ**

환경의 원시 OS 형이 바이트열이면 True (예를 들어, 윈도우에서는 False).

Added in version 3.2.

os.**umask**(*mask*, /)

현재 숫자 umask를 설정하고 이전 umask를 반환합니다.

The function is a stub on WASI, see 웹어셈블리 플랫폼 for more information.

os.**uname**()

현재 운영 체제를 식별하는 정보를 반환합니다. 반환 값은 5가지 어트리뷰트를 가진 객체입니다:

- sysname - 운영 체제 이름
- nodename - 네트워크상의 기계 이름 (구현이 정의)
- release - 운영 체제 릴리스
- version - 운영 체제 버전
- machine - 하드웨어 식별자

하위 호환성을 위해, 이 객체는 이터러블이기도 해서, sysname, nodename, release, version 및 machine이 이 순서로 포함된 5-튜플처럼 작동합니다.

일부 시스템에서는 nodename을 8자나 선행 구성 요소로 자릅니다; 호스트 이름을 얻는 더 좋은 방법은 *socket.gethostname()* 또는 더 나아가 socket.gethostbyaddr(socket.gethostname()) 입니다.

On macOS, iOS and Android, this returns the *kernel* name and version (i.e., 'Darwin' on macOS and iOS; 'Linux' on Android). *platform.uname()* can be used to get the user-facing operating system name and version on iOS and Android.

가용성: Unix.

버전 3.3에서 변경: 반환형이 튜플에서 이름이 지정된 어트리뷰트를 가진 튜플류 객체로 변경되었습니다.

os.**unsetenv**(*key*, /)

key 라는 이름의 환경 변수를 삭제합니다. 이러한 환경 변화는 *os.system()*, *popen()* 또는 *fork()* 및 *execv()*로 시작된 자식 프로세스에 영향을 줍니다.

Deletion of items in *os.environ* is automatically translated into a corresponding call to *unsetenv()*; however, calls to *unsetenv()* don' t update *os.environ*, so it is actually preferable to delete items of *os.environ*.

See also the *os.reload_environ()* function.

key를 인자로 감사 이벤트(*auditing event*) os.unsetenv를 발생시킵니다.

버전 3.9에서 변경: 이 함수는 이제 항상 사용할 수 있고 윈도우에서도 사용할 수 있습니다.

os.**unshare**(*flags*)

Disassociate parts of the process execution context, and move them into a newly created namespace. See the *unshare(2)* man page for more details. The *flags* argument is a bit mask, combining zero or more of the *CLONE_* constants*, that specifies which parts of the execution context should be unshared from their existing associations and moved to a new namespace. If the *flags* argument is 0, no changes are made to the calling process' s execution context.

가용성: Linux >= 2.6.16.

Added in version 3.12.

더 보기

The *setns()* function.

Flags to the `unshare()` function, if the implementation supports them. See `unshare(2)` in the Linux manual for their exact effect and availability.

os.**CLONE_FILES**

os.**CLONE_FS**

os.**CLONE_NEWCGROUP**

os.**CLONE_NEWIPC**

os.**CLONE_NEWNET**

os.**CLONE_NEWNS**

os.**CLONE_NEWPID**

os.**CLONE_NEWTIME**

os.**CLONE_NEWUSER**

os.**CLONE_NEWUTS**

os.**CLONE_SIGHAND**

os.**CLONE_SYSVSEM**

os.**CLONE_THREAD**

os.**CLONE_VM**

16.1.4 파일 객체 생성

이 함수들은 새로운 파일 객체를 만듭니다. (파일 기술자를 여는 것에 관해서는 `open()`를 참조하십시오.)

os.**fdopen**(*fd*, **args*, ***kwargs*)

파일 기술자 *fd*에 연결된 열린 파일 객체를 반환합니다. 이것은 `open()` 내장 함수의 별칭이며 같은 인자를 받아들입니다. 유일한 차이점은 `fdopen()`의 첫 번째 인자는 항상 정수여야 한다는 것입니다.

16.1.5 파일 기술자 연산

이 함수들은 파일 기술자를 사용하여 참조된 I/O 스트림에 작용합니다.

파일 기술자는 현재 프로세스에 의해 열린 파일에 대응하는 작은 정수입니다. 예를 들어, 표준 입력은 보통 파일 기술자 0이고, 표준 출력은 1이며, 표준 에러는 2입니다. 프로세스에 의해 열린 추가 파일은 3, 4, 5 등으로 지정됩니다. "파일 기술자"라는 이름은 약간 기만적입니다; 유닉스 플랫폼에서, 소켓과 파이프도 파일 기술자에 의해 참조됩니다.

`fileno()` 메서드는 필요할 때 파일 객체와 연관된 파일 기술자를 얻는 데 사용될 수 있습니다. 파일 기술자를 직접 사용하면 파일 객체 메서드를 거치지 않아서, 데이터의 내부 버퍼링과 같은 측면을 무시하게 되는 것에 유의하십시오.

os.**close**(*fd*)

파일 기술자 *fd*를 닫습니다.

> **참고**
>
> 이 함수는 저수준 I/O를 위한 것이며, `os.open()` 또는 `pipe()`에 의해 반환된 파일 기술자에 적용되어야 합니다. 내장 함수 `open()` 나 `popen()` 또는 `fdopen()`에 의해 반환된 "파일 객체"를 닫으려면, `close()` 메서드를 사용하십시오.

os.**closerange**(*fd_low*, *fd_high*, /)

에러는 무시하면서, *fd_low*(포함)부터 *fd_high*(제외)까지 모든 파일 기술자를 닫습니다. 다음과 동등합니다 (하지만 훨씬 빠릅니다):

```
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    except OSError:
        pass
```

os.**copy_file_range**(*src, dst, count, offset_src=None, offset_dst=None*)

Copy *count* bytes from file descriptor *src*, starting from offset *offset_src*, to file descriptor *dst*, starting from offset *offset_dst*. If *offset_src* is `None`, then *src* is read from the current position; respectively for *offset_dst*.

In Linux kernel older than 5.3, the files pointed to by *src* and *dst* must reside in the same filesystem, otherwise an `OSError` is raised with `errno` set to `errno.EXDEV`.

This copy is done without the additional cost of transferring data from the kernel to user space and then back into the kernel. Additionally, some filesystems could implement extra optimizations, such as the use of reflinks (i.e., two or more inodes that share pointers to the same copy-on-write disk blocks; supported file systems include btrfs and XFS) and server-side copy (in the case of NFS).

The function copies bytes between two file descriptors. Text options, like the encoding and the line ending, are ignored.

반환 값은 복사된 바이트의 양입니다. 이것은 요구된 양보다 적을 수 있습니다.

> **참고**
>
> On Linux, `os.copy_file_range()` should not be used for copying a range of a pseudo file from a special filesystem like procfs and sysfs. It will always copy no bytes and return 0 as if the file was empty because of a known Linux kernel issue.

가용성: Linux >= 4.5 with glibc >= 2.27.

Added in version 3.8.

os.**device_encoding**(*fd*)

fd 와 연관된 장치가 터미널에 연결되어 있을 때 인코딩을 설명하는 문자열을 반환합니다; 그렇지 않으면 `None`을 반환합니다.

On Unix, if the *Python UTF-8 Mode* is enabled, return `'UTF-8'` rather than the device encoding.

버전 3.10에서 변경: On Unix, the function now implements the Python UTF-8 Mode.

os.**dup**(*fd, /*)

파일 기술자 *fd* 의 복사본을 반환합니다. 새 파일 기술자는 상속 불가능합니다.

윈도우에서는, 표준 스트림(0: stdin, 1: stdout, 2: stderr)을 복제할 때, 새 파일 기술자가 상속 가능합니다.

가용성: not WASI.

버전 3.4에서 변경: 새로운 파일 기술자는 이제 상속 불가능합니다.

os.**dup2**(*fd, fd2, inheritable=True*)

파일 기술자 *fd* 를 *fd2*에 복제하고, 필요하면 먼저 후자를 닫습니다. *fd2*를 반환합니다. 새로운 파일 기술자는 기본적으로 상속 가능하고, *inheritable* 이 `False`면 상속 불가능합니다.

가용성: not WASI.

버전 3.4에서 변경: 선택적 *inheritable* 매개 변수를 추가했습니다.

버전 3.7에서 변경: 성공하면 *fd2* 를 반환합니다. 이전에는 항상 `None`을 반환했습니다.

os.**fchmod**(*fd, mode*)

fd 에 의해 주어진 파일의 모드를 숫자 *mode* 로 변경합니다. *mode*의 가능한 값은 `chmod()` 문서를 참조하십시오. 파이썬 3.3부터는, `os.chmod(fd, mode)`와 같습니다.

`path`, `mode`, `dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.chmod`를 발생시킵니다.

가용성: Unix, Windows.

The function is limited on WASI, see 웹어셈블리 플랫폼 for more information.

버전 3.13에서 변경: Added support on Windows.

os.**fchown**(*fd*, *uid*, *gid*)

fd 에 의해 주어진 파일의 소유자와 그룹 id를 숫자 *uid* 와 *gid*로 변경합니다. ID 중 하나를 변경하지 않으려면, 그것을 -1로 설정하십시오. `chown()`를 참조하십시오. 파이썬 3.3부터는, `os.chown(fd, uid, gid)` 와 같습니다.

`path`, `uid`, `gid`, `dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.chown`을 발생시킵니다.

가용성: Unix.

The function is limited on WASI, see 웹어셈블리 플랫폼 for more information.

os.**fdatasync**(*fd*)

파일 기술자 *fd* 로 주어진 파일을 디스크에 쓰도록 강제합니다. 메타 데이터를 갱신하도록 강제하지 않습니다.

가용성: Unix.

> **참고**
>
> 이 함수는 MacOS에서는 사용할 수 없습니다.

os.**fpathconf**(*fd*, *name*, /)

열린 파일과 관련된 시스템 구성 정보를 반환합니다. *name* 은 조회할 구성 값을 지정합니다; 정의된 시스템 값의 이름인 문자열일 수 있습니다; 이 이름은 여러 표준(POSIX.1, 유닉스 95, 유닉스 98 및 기타)에서 지정됩니다. 일부 플랫폼은 추가 이름도 정의합니다. 호스트 운영 체제에 알려진 이름은 `pathconf_names` 딕셔너리에서 제공됩니다. 이 매핑에 포함되지 않은 구성 변수의 경우, *name*에 정수를 전달하는 것도 허용됩니다.

name 이 문자열이고 알 수 없으면, `ValueError`가 발생합니다. *name*에 대한 특정 값이 호스트 시스템에서 지원되지 않으면, `pathconf_names`에 포함되어 있어도, 에러 번호가 `errno.EINVAL`인 `OSError`가 발생합니다.

파이썬 3.3부터, `os.pathconf(fd, name)` 과 같습니다.

가용성: Unix.

os.**fstat**(*fd*)

파일 기술자 *fd* 의 상태를 가져옵니다. `stat_result` 객체를 반환합니다.

파이썬 3.3부터는, `os.stat(fd)` 와 같습니다.

> **더 보기**
>
> `stat()` 함수.

os.**fstatvfs**(*fd*, /)

`statvfs()` 처럼, 파일 기술자 *fd* 와 연관된 파일을 포함하는 파일 시스템에 대한 정보를 반환합니다. 파이썬 3.3부터는, `os.statvfs(fd)` 와 같습니다.

가용성: Unix.

os.**fsync**(*fd*)

Force write of file with filedescriptor *fd* to disk. On Unix, this calls the native `fsync()` function; on Windows, the MS `_commit()` function.

버퍼링 된 파이썬 파일 객체 *f*로 시작하는 경우, *f* 와 연관된 모든 내부 버퍼가 디스크에 기록되게 하려면, 먼저 `f.flush()` 를 수행한 다음 `os.fsync(f.fileno())` 를 하십시오.

가용성: Unix, Windows.

os.**ftruncate**(*fd*, *length*, /)

파일 기술자 *fd*에 해당하는 파일을 잘라내어 최대 *length* 바이트가 되도록 만듭니다. 파이썬 3.3부터는, `os.truncate(fd, length)`와 같습니다.

`fd`, `length`를 인자로 감사 이벤트(*auditing event*) `os.truncate`를 발생시킵니다.

가용성: Unix, Windows.

버전 3.5에서 변경: 윈도우 지원 추가

os.**get_blocking**(*fd*, /)

파일 기술자의 블로킹 모드를 얻어옵니다: *O_NONBLOCK* 플래그가 설정되었으면 `False`, 플래그가 지워졌으면 `True`.

set_blocking() 및 *socket.socket.setblocking()*도 참조하십시오.

가용성: Unix, Windows.

The function is limited on WASI, see 웹어셈블리 플랫폼 for more information.

On Windows, this function is limited to pipes.

Added in version 3.5.

버전 3.12에서 변경: Added support for pipes on Windows.

os.**grantpt**(*fd*, /)

Grant access to the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the C standard library function `grantpt()`.

가용성: Unix, not WASI.

Added in version 3.13.

os.**isatty**(*fd*, /)

파일 기술자 *fd*가 열려 있고 tty(류의) 장치에 연결되어 있으면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

os.**lockf**(*fd*, *cmd*, *len*, /)

열린 파일 기술자에 POSIX 록을 적용, 검사 또는 제거합니다. *fd* 는 열린 파일 기술자입니다. *cmd* 는 사용할 명령을 지정합니다 - *F_LOCK*, *F_TLOCK*, *F_ULOCK* 또는 *F_TEST* 중 하나. *len* 은 잠글 파일의 영역을 지정합니다.

`fd`, `cmd`, `len`을 인자로 감사 이벤트(*auditing event*) `os.lockf`를 발생시킵니다.

가용성: Unix.

Added in version 3.3.

os.**F_LOCK**

os.**F_TLOCK**

os.**F_ULOCK**

os.**F_TEST**

*lockf()*가 취할 조치를 지정하는 플래그.

가용성: Unix.

Added in version 3.3.

os.**login_tty**(*fd*, /)

Prepare the tty of which fd is a file descriptor for a new login session. Make the calling process a session leader; make the tty the controlling tty, the stdin, the stdout, and the stderr of the calling process; close fd.

가용성: Unix, not WASI.

Added in version 3.11.

os.**lseek**(*fd*, *pos*, *whence*, /)

Set the current position of file descriptor *fd* to position *pos*, modified by *whence*, and return the new position in bytes relative to the start of the file. Valid values for *whence* are:

- `SEEK_SET` or `0` – set *pos* relative to the beginning of the file
- `SEEK_CUR` or `1` – set *pos* relative to the current file position
- `SEEK_END` or `2` – set *pos* relative to the end of the file
- `SEEK_HOLE` – set *pos* to the next data location, relative to *pos*
- `SEEK_DATA` – set *pos* to the next data hole, relative to *pos*

버전 3.3에서 변경: Add support for `SEEK_HOLE` and `SEEK_DATA`.

os.**SEEK_SET**

os.**SEEK_CUR**

os.**SEEK_END**

Parameters to the `lseek()` function and the `seek()` method on *file-like objects*, for whence to adjust the file position indicator.

SEEK_SET
Adjust the file position relative to the beginning of the file.

SEEK_CUR
Adjust the file position relative to the current file position.

SEEK_END
Adjust the file position relative to the end of the file.

Their values are 0, 1, and 2, respectively.

os.**SEEK_HOLE**

os.**SEEK_DATA**

Parameters to the `lseek()` function and the `seek()` method on *file-like objects*, for seeking file data and holes on sparsely allocated files.

SEEK_DATA
Adjust the file offset to the next location containing data, relative to the seek position.

SEEK_HOLE
Adjust the file offset to the next location containing a hole, relative to the seek position. A hole is defined as a sequence of zeros.

> **참고**
>
> These operations only make sense for filesystems that support them.

가용성: Linux >= 3.1, macOS, Unix

Added in version 3.3.

os.**open**(*path*, *flags*, *mode=0o777*, **, dir_fd=None*)

파일 *path*를 열고 *flags* 에 따른 다양한 플래그와 때로 *mode* 따른 모드를 설정합니다. *mode*를 계산할 때, 현재 umask 값으로 먼저 마스킹합니다. 새롭게 열린 파일의 파일 기술자를 돌려줍니다. 새 파일 기술자는 상속 불가능합니다.

플래그와 모드 값에 대한 설명은, C 런타임 설명서를 참조하십시오; 플래그 상수(`O_RDONLY` 와 `O_WRONLY`와 같은)는 `os` 모듈에 정의되어 있습니다. 특히, 윈도우에서 바이너리 모드로 파일을 열려면 `O_BINARY`를 추가해야 합니다.

이 함수는 *dir_fd* 매개 변수로 디렉터리 기술자에 상대적인 경로를 지원할 수 있습니다.

path, mode, flags를 인자로 감사 이벤트(*auditing event*) open을 발생시킵니다.

버전 3.4에서 변경: 새로운 파일 기술자는 이제 상속 불가능합니다.

> **참고**
>
> 이 함수는 저수준 I/O를 위한 것입니다. 일반적인 사용을 위해서는 내장 함수 *open()*을 사용하십시오, 이 함수는 read() 및 write() 메서드(와 더 많은 메서드)가있는 파일 객체를 반환합니다. 파일 기술자를 파일 객체로 싸려면, *fdopen()*을 사용하십시오.

버전 3.3에서 변경: *dir_fd* 매개 변수가 추가되었습니다.

버전 3.5에서 변경: 시스템 호출이 인터럽트 되고 시그널 처리기가 예외를 발생시키지 않으면, 함수는 이제 *InterruptedError* 예외를 일으키는 대신 시스템 호출을 재시도합니다 (이유는 **PEP 475**를 참조하세요).

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

다음 상수는 *open()* 함수에 대한 *flags* 매개 변수의 옵션입니다. 비트별 OR 연산자 | 를 사용하여 결합할 수 있습니다. 일부는 모든 플랫폼에서 사용할 수는 없습니다. 가용성과 사용에 대한 설명은 유닉스의 *open(2)* 매뉴얼 페이지 또는 윈도우의 MSDN을 참조하십시오.

os.**O_RDONLY**

os.**O_WRONLY**

os.**O_RDWR**

os.**O_APPEND**

os.**O_CREAT**

os.**O_EXCL**

os.**O_TRUNC**

위의 상수는 유닉스 및 윈도우에서 사용할 수 있습니다.

os.**O_DSYNC**

os.**O_RSYNC**

os.**O_SYNC**

os.**O_NDELAY**

os.**O_NONBLOCK**

os.**O_NOCTTY**

os.**O_CLOEXEC**

위의 상수는 유닉스에서만 사용할 수 있습니다.

버전 3.3에서 변경: *O_CLOEXEC* 상수를 추가합니다.

os.**O_BINARY**

os.**O_NOINHERIT**

os.**O_SHORT_LIVED**

os.**O_TEMPORARY**

os.**O_RANDOM**

os.**O_SEQUENTIAL**

os.**O_TEXT**

위의 상수는 윈도우에서만 사용할 수 있습니다.

os.**O_EVTONLY**

os.**O_FSYNC**

os.**O_SYMLINK**

os.**O_NOFOLLOW_ANY**

The above constants are only available on macOS.

버전 3.10에서 변경: Add *O_EVTONLY*, *O_FSYNC*, *O_SYMLINK* and *O_NOFOLLOW_ANY* constants.

os.**O_ASYNC**

os.**O_DIRECT**

os.**O_DIRECTORY**

os.**O_NOFOLLOW**

os.**O_NOATIME**

os.**O_PATH**

os.**O_TMPFILE**

os.**O_SHLOCK**

os.**O_EXLOCK**

위의 상수는 확장이며 C 라이브러리에서 정의하지 않으면 존재하지 않습니다.

버전 3.4에서 변경: 지원하는 시스템에 *O_PATH*를 추가합니다. 리눅스 커널 3.11 이상에서만 사용 가능한 *O_TMPFILE*를 추가합니다.

os.**openpty**()

새로운 의사 터미널 쌍을 엽니다. 파일 기술자의 쌍 `(master, slave)` 를 반환하는데, 각각 pty와 tty 입니다. 새 파일 기술자는 상속 불가능합니다. (약간) 더 이식성 있는 접근 방식을 사용하려면, *pty* 모듈을 사용하십시오.

가용성: Unix, not WASI.

버전 3.4에서 변경: 새로운 파일 기술자는 이제 상속 불가능합니다.

os.**pipe**()

파이프를 만듭니다. 파일 기술자 쌍 `(r, w)` 를 반환하는데, 각각 읽기와 쓰기에 사용할 수 있습니다. 새 파일 기술자는 상속 불가능합니다.

가용성: Unix, Windows.

버전 3.4에서 변경: 새로운 파일 기술자는 이제 상속 불가능합니다.

os.**pipe2**(*flags*, /)

flags 가 원자적으로 설정된 파이프를 만듭니다. *flags* 는 다음과 같은 값들을 하나 이상 OR 해서 만들 수 있습니다: *O_NONBLOCK*, *O_CLOEXEC*. 파일 기술자 쌍 `(r, w)` 를 반환하는데, 각각 읽기와 쓰기에 사용할 수 있습니다.

가용성: Unix, not WASI.

Added in version 3.3.

os.**posix_fallocate**(*fd*, *offset*, *len*, /)

*fd*로 지정된 파일이 *offset* 에서 시작하여 *len* 바이트 동안 계속되도록 충분한 디스크 공간을 할당합니다.

가용성: Unix.

Added in version 3.3.

os.**posix_fadvise**(*fd*, *offset*, *len*, *advice*, /)

특정 패턴으로 데이터에 액세스하려는 의도를 알려 커널이 최적화할 수 있도록 합니다. 조언(advice)은 *fd*에 의해 지정된 파일의 *offset* 에서 시작하여 *len* 바이트 동안 계속되는 영역에 적용됩니다. *advice* 는 *POSIX_FADV_NORMAL*, *POSIX_FADV_SEQUENTIAL*, *POSIX_FADV_RANDOM*, *POSIX_FADV_NOREUSE*, *POSIX_FADV_WILLNEED* 또는 *POSIX_FADV_DONTNEED* 중 하나입니다.

가용성: Unix.

Added in version 3.3.

os.**POSIX_FADV_NORMAL**

os.**POSIX_FADV_SEQUENTIAL**

os.**POSIX_FADV_RANDOM**

os.**POSIX_FADV_NOREUSE**

os.**POSIX_FADV_WILLNEED**

os.**POSIX_FADV_DONTNEED**

사용 가능성이 큰 액세스 패턴을 지정하는 `posix_fadvise()`의 *advice* 에 사용될 수 있는 플래그.

가용성: Unix.

Added in version 3.3.

os.**pread**(*fd, n, offset, /*)

파일 기술자 *fd*에서 *offset* 의 위치부터 최대 *n* 바이트를 읽어 들이고, 파일 오프셋은 변경되지 않은 채로 남겨 둡니다.

읽어 들인 바이트를 포함하는 바이트열을 돌려줍니다. *fd* 에 의해 참조된 파일의 끝에 도달하면, 빈 바이트열 객체가 반환됩니다.

가용성: Unix.

Added in version 3.3.

os.**posix_openpt**(*oflag, /*)

Open and return a file descriptor for a master pseudo-terminal device.

Calls the C standard library function `posix_openpt()`. The *oflag* argument is used to set file status flags and file access modes as specified in the manual page of `posix_openpt()` of your system.

The returned file descriptor is *non-inheritable*. If the value `O_CLOEXEC` is available on the system, it is added to *oflag*.

가용성: Unix, not WASI.

Added in version 3.13.

os.**preadv**(*fd, buffers, offset, flags=0, /*)

파일 기술자 *fd*에서 *offset* 위치부터 가변 바이트열류 객체들 *buffers* 로 읽어 들이고, 파일 오프셋은 변경되지 않은 채로 남겨 둡니다. 데이터가 가득 찰 때까지 각 버퍼로 데이터를 전송한 다음 나머지 데이터를 보관하기 위해 시퀀스의 다음 버퍼로 이동합니다.

flags 인자는 다음 플래그 중 0개 이상의 비트별 OR를 포함합니다:

- `RWF_HIPRI`
- `RWF_NOWAIT`

실제로 읽힌 총 바이트 수를 반환합니다. 이 값은 모든 객체의 총 용량보다 작을 수 있습니다.

운영 체제는 사용할 수 있는 버퍼 수에 한계(`sysconf()` 값 `'SC_IOV_MAX'`)를 설정할 수 있습니다.

`os.readv()` 와 `os.pread()`의 기능을 결합합니다.

가용성: Linux >= 2.6.30, FreeBSD >= 6.0, OpenBSD >= 2.7, AIX >= 7.1.

Using flags requires Linux >= 4.6.

Added in version 3.7.

os.**RWF_NOWAIT**

즉시 사용할 수 없는 데이터를 기다리지 않습니다. 이 플래그를 지정하면, 하부 저장 장치에서 데이터를 읽어야 하거나 록을 기다려야 할 때 즉시 시스템 호출이 반환됩니다.

If some data was successfully read, it will return the number of bytes read. If no bytes were read, it will return `-1` and set errno to `errno.EAGAIN`.

가용성: Linux >= 4.14.

Added in version 3.7.

os.**RWF_HIPRI**

우선순위가 높은 읽기/쓰기. 블록 기반 파일 시스템이 장치의 폴링을 사용할 수 있게 하여, 지연은 짧아 지지만, 추가 자원을 사용할 수 있습니다.

현재, 리눅스에서, 이 기능은 *O_DIRECT* 플래그를 사용하여 열린 파일 기술자에만 사용할 수 있습니다.

가용성: Linux >= 4.6.

Added in version 3.7.

os.**ptsname**(*fd*, /)

Return the name of the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the reentrant C standard library function `ptsname_r()` if it is available; otherwise, the C standard library function `ptsname()`, which is not guaranteed to be thread-safe, is called.

가용성: Unix, not WASI.

Added in version 3.13.

os.**pwrite**(*fd*, *str*, *offset*, /)

파일 기술자 *fd*의 *offset* 위치에 *str* 바이트열을 쓰고, 파일 오프셋은 변경되지 않은 채로 남겨 둡니다.

실제로 쓴 바이트 수를 반환합니다.

가용성: Unix.

Added in version 3.3.

os.**pwritev**(*fd*, *buffers*, *offset*, *flags=0*, /)

Write the *buffers* contents to file descriptor *fd* at an offset *offset*, leaving the file offset unchanged. *buffers* must be a sequence of *bytes-like objects*. Buffers are processed in array order. Entire contents of the first buffer is written before proceeding to the second, and so on.

flags 인자는 다음 플래그 중 0개 이상의 비트별 OR를 포함합니다:

- *RWF_DSYNC*
- *RWF_SYNC*
- *RWF_APPEND*

실제로 쓴 총 바이트 수를 반환합니다.

운영 체제는 사용할 수 있는 버퍼 수에 한계(*sysconf()* 값 `'SC_IOV_MAX'`)를 설정할 수 있습니다.

os.writev() 와 *os.pwrite()*의 기능을 결합합니다.

가용성: Linux >= 2.6.30, FreeBSD >= 6.0, OpenBSD >= 2.7, AIX >= 7.1.

Using flags requires Linux >= 4.6.

Added in version 3.7.

os.**RWF_DSYNC**

Provide a per-write equivalent of the *O_DSYNC* *os.open()* flag. This flag effect applies only to the data range written by the system call.

가용성: Linux >= 4.7.

Added in version 3.7.

os.**RWF_SYNC**

Provide a per-write equivalent of the *O_SYNC* *os.open()* flag. This flag effect applies only to the data range written by the system call.

가용성: Linux >= 4.7.

Added in version 3.7.

os.**RWF_APPEND**

Provide a per-write equivalent of the `O_APPEND` `os.open()` flag. This flag is meaningful only for `os.pwritev()`, and its effect applies only to the data range written by the system call. The *offset* argument does not affect the write operation; the data is always appended to the end of the file. However, if the *offset* argument is `-1`, the current file *offset* is updated.

가용성: Linux >= 4.16.

Added in version 3.10.

os.**read**(*fd*, *n*, /)

파일 기술자 *fd*에서 최대 *n* 바이트를 읽습니다.

읽어 들인 바이트를 포함하는 바이트열을 돌려줍니다. *fd* 에 의해 참조된 파일의 끝에 도달하면, 빈 바이트열 객체가 반환됩니다.

> **참고**
>
> 이 함수는 저수준 I/O를 위한 것이며 `os.open()` 이나 `pipe()` 에 의해 반환된 파일 기술자에 적용되어야 합니다. 내장 함수 `open()` 이나 `popen()` 또는 `fdopen()` 에 의해 반환된 "파일 객체" 나 `sys.stdin`을 읽으려면, 그것의 `read()` 나 `readline()` 메서드를 사용하십시오.

버전 3.5에서 변경: 시스템 호출이 인터럽트 되고 시그널 처리기가 예외를 발생시키지 않으면, 함수는 이제 `InterruptedError` 예외를 일으키는 대신 시스템 호출을 재시도합니다 (이유는 **PEP 475**를 참조하세요).

os.**readinto**(*fd*, *buffer*, /)

Read from a file descriptor *fd* into a mutable buffer object *buffer*.

The *buffer* should be mutable and *bytes-like*. On success, returns the number of bytes read. Less bytes may be read than the size of the buffer. The underlying system call will be retried when interrupted by a signal, unless the signal handler raises an exception. Other errors will not be retried and an error will be raised.

Returns 0 if *fd* is at end of file or if the provided *buffer* has length 0 (which can be used to check for errors without reading data). Never returns negative.

> **참고**
>
> This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `os.pipe()`. To read a "file object" returned by the built-in function `open()`, or `sys.stdin`, use its member functions, for example `io.BufferedIOBase.readinto()`, `io.BufferedIOBase.read()`, or `io.TextIOBase.read()`

Added in version 3.14.

os.**sendfile**(*out_fd*, *in_fd*, *offset*, *count*)

os.**sendfile**(*out_fd*, *in_fd*, *offset*, *count*, *headers*=(), *trailers*=(), *flags*=*0*)

파일 기술자 *in_fd*에서 파일 기술자 *out_fd* 로 *offset*에서 시작하여 *count* 바이트를 복사합니다. 전송된 바이트 수를 반환합니다. EOF에 도달하면 `0`을 반환합니다.

첫 번째 함수 서명은 `sendfile()`를 정의하는 모든 플랫폼에서 지원됩니다.

리눅스에서, *offset*이 `None`으로 주어지면, *in_fd*의 현재 위치에서 바이트를 읽고 *in_fd*의 위치가 갱신됩니다.

The second case may be used on macOS and FreeBSD where *headers* and *trailers* are arbitrary sequences of buffers that are written before and after the data from *in_fd* is written. It returns the same as the first case.

On macOS and FreeBSD, a value of `0` for *count* specifies to send until the end of *in_fd* is reached.

모든 플랫폼은 *out_fd* 파일 기술자로 소켓을 지원하고, 일부 플랫폼은 다른 유형(예를 들어 일반 파일, 파이프)들도 허락합니다.

이기종 플랫폼 응용 프로그램은 *headers*, *trailers* 및 *flags* 인자를 사용해서는 안 됩니다.

가용성: Unix, not WASI.

> **참고**
>
> `sendfile()`의 고수준 래퍼는, `socket.socket.sendfile()`을 보십시오.

Added in version 3.3.

버전 3.9에서 변경: 매개 변수 *out*과 *in*의 이름이 *out_fd*와 *in_fd*로 변경되었습니다.

os.**SF_NODISKIO**

os.**SF_MNOWAIT**

os.**SF_SYNC**

구현이 지원하는 경우, `sendfile()` 함수에 대한 매개 변수입니다.

가용성: Unix, not WASI.

Added in version 3.3.

os.**SF_NOCACHE**

Parameter to the `sendfile()` function, if the implementation supports it. The data won' t be cached in the virtual memory and will be freed afterwards.

가용성: Unix, not WASI.

Added in version 3.11.

os.**set_blocking**(*fd*, *blocking*, /)

지정된 파일 기술자의 블로킹 모드를 설정합니다. blocking이 `False`면 `O_NONBLOCK` 플래그를 설정하고, 그렇지 않으면 플래그를 지웁니다.

`get_blocking()`과 `socket.socket.setblocking()`도 참조하십시오.

가용성: Unix, Windows.

The function is limited on WASI, see 웹어셈블리 플랫폼 for more information.

On Windows, this function is limited to pipes.

Added in version 3.5.

버전 3.12에서 변경: Added support for pipes on Windows.

os.**splice**(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*, *flags=0*)

Transfer *count* bytes from file descriptor *src*, starting from offset *offset_src*, to file descriptor *dst*, starting from offset *offset_dst*.

The splicing behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the `|` operator):

- If `SPLICE_F_MOVE` is specified, the kernel is asked to move pages instead of copying, but pages may still be copied if the kernel cannot move the pages from the pipe.
- If `SPLICE_F_NONBLOCK` is specified, the kernel is asked to not block on I/O. This makes the splice pipe operations nonblocking, but splice may nevertheless block because the spliced file descriptors may block.
- If `SPLICE_F_MORE` is specified, it hints to the kernel that more data will be coming in a subsequent splice.

At least one of the file descriptors must refer to a pipe. If *offset_src* is `None`, then *src* is read from the current position; respectively for *offset_dst*. The offset associated to the file descriptor that refers to a pipe must be `None`. The files pointed to by *src* and *dst* must reside in the same filesystem, otherwise an `OSError` is raised with `errno` set to `errno.EXDEV`.

이 복사는 커널에서 사용자 공간으로 데이터를 전송한 다음 다시 커널로 전송하는 추가 비용 없이 수행됩니다. 또한, 일부 파일 시스템은 추가 최적화를 구현할 수 있습니다. 두 파일이 바이너리로 열린 것처럼 복사가 수행됩니다.

Upon successful completion, returns the number of bytes spliced to or from the pipe. A return value of 0 means end of input. If *src* refers to a pipe, then this means that there was no data to transfer, and it would not make sense to block because there are no writers connected to the write end of the pipe.

> **더 보기**
>
> The `splice(2)` man page.

가용성: Linux >= 2.6.17 with glibc >= 2.5

Added in version 3.10.

os.**SPLICE_F_MOVE**

os.**SPLICE_F_NONBLOCK**

os.**SPLICE_F_MORE**

Added in version 3.10.

os.**readv**(*fd*, *buffers*, /)

파일 기술자 *fd*에서 여러 가변 바이트열류 객체 *buffers*로 읽어 들입니다. 데이터가 가득 찰 때까지 각 버퍼로 데이터를 전송한 다음 나머지 데이터를 보관하기 위해 시퀀스의 다음 버퍼로 이동합니다.

실제로 읽힌 총 바이트 수를 반환합니다. 이 값은 모든 객체의 총 용량보다 작을 수 있습니다.

운영 체제는 사용할 수 있는 버퍼 수에 한계(`sysconf()` 값 `'SC_IOV_MAX'`)를 설정할 수 있습니다.

가용성: Unix.

Added in version 3.3.

os.**tcgetpgrp**(*fd*, /)

fd(`os.open()`에 의해 반환된 것과 같은 열린 파일 기술자)에 의해 주어진 터미널과 관련된 프로세스 그룹을 반환합니다.

가용성: Unix, not WASI.

os.**tcsetpgrp**(*fd*, *pg*, /)

fd(`os.open()`에 의해 반환된 것과 같은 열린 파일 기술자)에 의해 주어진 터미널과 관련된 프로세스 그룹을 *pg*로 설정합니다.

가용성: Unix, not WASI.

os.**ttyname**(*fd*, /)

파일 기술자 *fd*와 관련된 터미널 장치를 나타내는 문자열을 돌려줍니다. *fd* 가 터미널 장치와 연관되어 있지 않으면, 예외가 발생합니다.

가용성: Unix.

os.**unlockpt**(*fd*, /)

Unlock the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the C standard library function `unlockpt()`.

가용성: Unix, not WASI.

Added in version 3.13.

os.**write**(*fd*, *str*, /)

str 바이트열을 파일 기술자 *fd* 에 씁니다.

실제로 쓴 바이트 수를 반환합니다.

> **참고**
>
> 이 함수는 저수준 I/O를 위한 것이며 os.open() 이나 pipe() 에 의해 반환된 파일 기술자에 적용되어야 합니다. 내장 함수 open() 이나 popen() 또는 fdopen() 에 의해 반환된 "파일 객체" 나 sys.stdout 또는 sys.stderr에 쓰려면, 그것의 write() 메서드를 사용하십시오.

버전 3.5에서 변경: 시스템 호출이 인터럽트 되고 시그널 처리기가 예외를 발생시키지 않으면, 함수는 이제 InterruptedError 예외를 일으키는 대신 시스템 호출을 재시도합니다 (이유는 **PEP 475**를 참조하세요).

os.**writev**(*fd*, *buffers*, /)

buffers 내용을 파일 기술자 *fd*에 씁니다. *buffers* 는 바이트열류 객체의 시퀀스 여야 합니다. 버퍼는 배열 순서로 처리됩니다. 첫 번째 버퍼의 전체 내용은 두 번째 버퍼로 진행하기 전에 기록되고, 같은 식으로 계속 진행합니다.

실제로 쓴 총 바이트 수를 반환합니다.

운영 체제는 사용할 수 있는 버퍼 수에 한계(sysconf() 값 'SC_IOV_MAX')를 설정할 수 있습니다.

가용성: Unix.

Added in version 3.3.

터미널의 크기 조회하기

Added in version 3.3.

os.**get_terminal_size**(*fd=STDOUT_FILENO*, /)

터미널 창의 크기를 (columns, lines) 로 반환하는데, terminal_size 형의 튜플입니다.

선택적 인자 fd(기본값 STDOUT_FILENO, 즉 표준 출력)는 조회할 파일 기술자를 지정합니다.

파일 기술자가 터미널에 연결되어 있지 않으면, OSError가 발생합니다.

shutil.get_terminal_size()가 일반적으로 사용해야 하는 고수준 함수이며, os.get_terminal_size는 저수준 구현입니다.

가용성: Unix, Windows.

class os.**terminal_size**

터미널 창 크기 (columns, lines) 를 저장하는 튜플의 서브 클래스.

columns

문자 단위의 터미널 창의 너비.

lines

문자 단위의 터미널 창의 높이.

파일 기술자의 상속

Added in version 3.4.

파일 기술자는 자식 프로세스가 파일 기술자를 상속받을 수 있는지를 나타내는 "상속 가능" 플래그를 가지고 있습니다. 파이썬 3.4부터, 파이썬에 의해 생성된 파일 기술자는 기본적으로 상속 불가능합니다.

유닉스에서는, 상속 불가능한 파일 기술자는 새 프로그램 실행 시 자식 프로세스에서 닫히고, 다른 파일 기술자는 상속됩니다.

윈도우에서는, 항상 상속되는 표준 스트림(파일 기술자 0, 1, 2: stdin, stdout, stderr)을 제외하고, 상속 불가능한 핸들 및 파일 기술자는 자식 프로세스에서 닫힙니다. spawn* 함수를 사용하면, 상속 가능한 모든

핸들과 상속 가능한 모든 파일 기술자가 상속됩니다. `subprocess` 모듈을 사용하면, 표준 스트림을 제외한 모든 파일 기술자가 닫히고, 상속 가능한 핸들은 *close_fds* 매개 변수가 `False` 일 때만 상속됩니다.

On WebAssembly platforms, the file descriptor cannot be modified.

`os.`**`get_inheritable`**(*fd*, /)

지정된 파일 기술자의 "상속 가능" 플래그를 가져옵니다 (논릿값).

`os.`**`set_inheritable`**(*fd*, *inheritable*, /)

지정된 파일 기술자의 "상속 가능(inheritable)" 플래그를 설정합니다.

`os.`**`get_handle_inheritable`**(*handle*, /)

지정된 핸들의 "상속 가능" 플래그를 가져옵니다 (논릿값).

가용성: Windows.

`os.`**`set_handle_inheritable`**(*handle*, *inheritable*, /)

지정된 핸들의 "상속 가능(inheritable)" 플래그를 설정합니다.

가용성: Windows.

16.1.6 파일과 디렉터리

일부 유닉스 플랫폼에서, 이 함수 중 많은 것들이 다음 기능 중 하나 이상을 지원합니다:

- **specifying a file descriptor:** Normally the *path* argument provided to functions in the `os` module must be a string specifying a file path. However, some functions now alternatively accept an open file descriptor for their *path* argument. The function will then operate on the file referred to by the descriptor. For POSIX systems, Python will call the variant of the function prefixed with `f` (e.g. call `fchdir` instead of `chdir`).

  `os.supports_fd`를 사용하여, 여러분의 플랫폼에서 특정 함수에 파일 기술자로 *path*를 지정할 수 있는지를 확인할 수 있습니다. 사용할 수 없을 때, 사용하면 `NotImplementedError`를 발생시킵니다.

  함수가 *dir_fd* 나 *follow_symlinks* 인자도 지원하면, *path* 에 파일 기술자를 제공할 때, 이 중 하나를 지정하는 것은 에러입니다.

- **paths relative to directory descriptors:** If *dir_fd* is not `None`, it should be a file descriptor referring to a directory, and the path to operate on should be relative; path will then be relative to that directory. If the path is absolute, *dir_fd* is ignored. For POSIX systems, Python will call the variant of the function with an `at` suffix and possibly prefixed with `f` (e.g. call `faccessat` instead of `access`).

  `os.supports_dir_fd`를 사용하여, 여러분의 플랫폼에서 특정 함수에 *dir_fd*가 지원되는지를 확인할 수 있습니다. 사용할 수 없을 때, 사용하면 `NotImplementedError`를 발생시킵니다.

- **not following symlinks:** If *follow_symlinks* is `False`, and the last element of the path to operate on is a symbolic link, the function will operate on the symbolic link itself rather than the file pointed to by the link. For POSIX systems, Python will call the `l...` variant of the function.

  `os.supports_follow_symlinks`를 사용하여, 여러분의 플랫폼에서 특정 함수에 *follow_symlinks*가 지원되는지를 확인할 수 있습니다. 사용할 수 없을 때, 사용하면 `NotImplementedError`를 발생시킵니다.

`os.`**`access`**(*path*, *mode*, *, *dir_fd=None*, *effective_ids=False*, *follow_symlinks=True*)

실제(real) uid/gid를 사용해서 *path*를 액세스할 수 있는지 검사합니다. 대부분의 연산은 유효한(effective) uid/gid를 사용할 것이므로, 이 함수는 suid/sgid 환경에서 호출하는 사용자가 지정된 *path* 에 대한 액세스 권한이 있는지 검사하는데 사용할 수 있습니다. *path*가 존재하는지를 검사하려면 *mode* 는 `F_OK` 여야 하며, 권한을 검사하려면 하나 이상의 `R_OK`, `W_OK` 및 `X_OK`를 OR 값일 수 있습니다. 액세스가 허용되면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다. 더 자세한 정보는 유닉스 매뉴얼 페이지 `access(2)`를 참조하십시오.

이 함수는 디렉터리 기술자에 상대적인 경로 와 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

effective_ids 가 `True` 면, `access()`는 실제(real) uid/gid 대신 유효한(effective) uid/gid를 사용하여 액세스 검사를 수행합니다. *effective_ids* 는 플랫폼에서 지원되지 않을 수 있습니다; `os.supports_effective_ids`를 사용하여, 사용할 수 있는지를 확인할 수 있습니다. 사용할 수 없을 때, 사용하면 `NotImplementedError`를 발생시킵니다.

> **참고**
>
> 예를 들어, 실제로 `open()`를 사용하여 파일을 열기 전에, `access()`를 사용하여 파일을 여는 권한이 있는지 확인하는 것은 보안 구멍을 만듭니다. 사용자가 파일을 확인하고 조작을 위해 열기 사이의 짧은 시간 간격을 악용할 수 있기 때문입니다. *EAFP* 기법을 사용하는 것이 좋습니다. 예를 들면:
>
> ```
> if os.access("myfile", os.R_OK):
>  with open("myfile") as fp:
>  return fp.read()
> return "some default data"
> ```
>
> 는 다음과 같이 쓰는 것이 더 좋습니다:
>
> ```
> try:
>  fp = open("myfile")
> except PermissionError:
>  return "some default data"
> else:
>  with fp:
>  return fp.read()
> ```

> **참고**
>
> `access()`가 성공할 것임을 알릴 때도, I/O 연산이 실패할 수 있습니다. 특히 일반적인 POSIX 권한 비트 모델을 넘어서는 권한 의미가 있을 수 있는 네트워크 파일 시스템에 대한 연산에서 그럴 수 있습니다.

버전 3.3에서 변경: *dir_fd*, *effective_ids* 및 *follow_symlinks* 매개 변수를 추가했습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**F_OK**

os.**R_OK**

os.**W_OK**

os.**X_OK**

path 의 존재 여부, 읽기 가능성, 쓰기 가능성 및 실행 가능성을 검사하기 위해, `access()`의 *mode* 매개 변수로 전달할 값입니다.

os.**chdir**(*path*)

현재 작업 디렉터리를 *path*로 변경합니다.

이 함수는 파일 기술자 지정을 지원할 수 있습니다. 기술자는 열려있는 파일이 아니라, 열려있는 디렉터리를 참조해야 합니다.

이 함수는 `OSError`와 `FileNotFoundError`, `PermissionError` 및 `NotADirectoryError`와 같은 서브 클래스를 발생시킬 수 있습니다.

`path`를 인자로 감사 이벤트(*auditing event*) `os.chdir`을 발생시킵니다.

버전 3.3에서 변경: 일부 플랫폼에서 *path* 를 파일 기술자로 지정하는 지원이 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**chflags**(*path*, *flags*, **, *follow_symlinks=True*)

path 의 플래그를 숫자 *flags*로 설정합니다. *flags*는 다음 값들(`stat` 모듈에 정의된 대로)의 조합(비트별 OR)을 취할 수 있습니다:

- `stat.UF_NODUMP`
- `stat.UF_IMMUTABLE`

- stat.UF_APPEND
- stat.UF_OPAQUE
- stat.UF_NOUNLINK
- stat.UF_COMPRESSED
- stat.UF_HIDDEN
- stat.SF_ARCHIVED
- stat.SF_IMMUTABLE
- stat.SF_APPEND
- stat.SF_NOUNLINK
- stat.SF_SNAPSHOT

이 함수는 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

path, flags를 인자로 감사 이벤트(*auditing event*) os.chflags를 발생시킵니다.

가용성: Unix, not WASI.

버전 3.3에서 변경: Added the *follow_symlinks* parameter.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**chmod**(*path*, *mode*, **, *dir_fd=None*, *follow_symlinks=True*)

*path*의 모드를 숫자 *mode*로 변경합니다. *mode* 는 다음 값들(stat 모듈에 정의된 대로)이나 이들의 비트별 OR 조합을 취할 수 있습니다:

- stat.S_ISUID
- stat.S_ISGID
- stat.S_ENFMT
- stat.S_ISVTX
- stat.S_IREAD
- stat.S_IWRITE
- stat.S_IEXEC
- stat.S_IRWXU
- stat.S_IRUSR
- stat.S_IWUSR
- stat.S_IXUSR
- stat.S_IRWXG
- stat.S_IRGRP
- stat.S_IWGRP
- stat.S_IXGRP
- stat.S_IRWXO
- stat.S_IROTH
- stat.S_IWOTH
- stat.S_IXOTH

이 함수는 파일 기술자 지정, 디렉터리 기술자에 상대적인 경로 및 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

> **참고**
>
> Although Windows supports `chmod()`, you can only set the file' s read-only flag with it (via the `stat.S_IWRITE` and `stat.S_IREAD` constants or a corresponding integer value). All other bits are ignored. The default value of *follow_symlinks* is `False` on Windows.
>
> The function is limited on WASI, see 웹어셈블리 플랫폼 for more information.

`path`, `mode`, `dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.chmod`를 발생시킵니다.

버전 3.3에서 변경: *path*를 열린 파일 기술자로 지정하는 지원과 *dir_fd* 및 *follow_symlinks* 인자가 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.13에서 변경: Added support for a file descriptor and the *follow_symlinks* argument on Windows.

`os.`**`chown`**(*path*, *uid*, *gid*, **, dir_fd=None, follow_symlinks=True*)

path 의 소유자와 그룹 ID를 숫자 *uid* 와 *gid*로 변경합니다. ID 중 하나를 변경하지 않으려면, 그것을 -1로 설정하십시오.

이 함수는 파일 기술자 지정, 디렉터리 기술자에 상대적인 경로 및 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

숫자 ID 이외에 이름을 허용하는 고수준 함수는 `shutil.chown()`를 참조하십시오.

`path`, `uid`, `gid`, `dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.chown`을 발생시킵니다.

가용성: Unix.

The function is limited on WASI, see 웹어셈블리 플랫폼 for more information.

버전 3.3에서 변경: *path*를 열린 파일 기술자로 지정하는 지원과 *dir_fd* 및 *follow_symlinks* 인자가 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 지원합니다.

`os.`**`chroot`**(*path*)

현재 프로세스의 루트 디렉터리를 *path*로 변경합니다.

가용성: Unix, not WASI, not Android.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

`os.`**`fchdir`**(*fd*)

현재 작업 디렉터리를 파일 기술자 *fd*가 나타내는 디렉터리로 변경합니다. 기술자는 열려있는 파일이 아니라 열려있는 디렉터리를 참조해야 합니다. 파이썬 3.3부터는, `os.chdir(fd)` 와 같습니다.

`path`를 인자로 감사 이벤트(*auditing event*) `os.chdir`을 발생시킵니다.

가용성: Unix.

`os.`**`getcwd`**()

현재 작업 디렉터리를 나타내는 문자열을 반환합니다.

`os.`**`getcwdb`**()

현재 작업 디렉터리를 나타내는 바이트열을 반환합니다.

버전 3.8에서 변경: 이 함수는 이제 윈도우에서 ANSI 코드 페이지가 아닌 UTF-8 인코딩을 사용합니다: 이유는 **PEP 529**를 참조하십시오. 이 함수는 윈도우에서 더는 폐지되지 않습니다.

`os.`**`lchflags`**(*path*, *flags*)

path 의 플래그를, `chflags()` 처럼, 숫자 *flags*로 설정하지만, 심볼릭 링크를 따르지 않습니다. 파이썬 3.3부터는, `os.chflags(path, flags, follow_symlinks=False)` 와 같습니다.

`path`, `flags`를 인자로 감사 이벤트(*auditing event*) `os.chflags`를 발생시킵니다.

가용성: Unix, not WASI.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**lchmod**(*path*, *mode*)

path 모드를 숫자 *mode*로 변경합니다. path가 심볼릭 링크면, 이 함수는 타깃이 아닌 심볼릭 링크에 영향을 미칩니다. *mode*의 가능한 값은 `chmod()` 문서를 참조하십시오. 파이썬 3.3부터는, `os.chmod(path, mode, follow_symlinks=False)`와 같습니다.

`lchmod()` is not part of POSIX, but Unix implementations may have it if changing the mode of symbolic links is supported.

`path`, `mode`, `dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.chmod`를 발생시킵니다.

가용성: Unix, Windows, not Linux, FreeBSD >= 1.3, NetBSD >= 1.3, not OpenBSD

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.13에서 변경: Added support on Windows.

os.**lchown**(*path*, *uid*, *gid*)

*path*의 소유자와 그룹 ID를 숫자 *uid*와 *gid*로 변경합니다. 이 함수는 심볼릭 링크를 따르지 않습니다. 파이썬 3.3부터는, `os.chown(path, uid, gid, follow_symlinks=False)`와 같습니다.

`path`, `uid`, `gid`, `dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.chown`을 발생시킵니다.

가용성: Unix.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**link**(*src*, *dst*, **, *src_dir_fd=None*, *dst_dir_fd=None*, *follow_symlinks=True*)

*src*를 가리키는 *dst* 라는 이름의 하드 링크를 만듭니다.

This function can support specifying *src_dir_fd* and/or *dst_dir_fd* to supply *paths relative to directory descriptors*, and *not following symlinks*. The default value of *follow_symlinks* is `False` on Windows.

`src`, `dst`, `src_dir_fd`, `dst_dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.link`를 발생시킵니다.

가용성: Unix, Windows.

버전 3.2에서 변경: 윈도우 지원이 추가되었습니다.

버전 3.3에서 변경: Added the *src_dir_fd*, *dst_dir_fd*, and *follow_symlinks* parameters.

버전 3.6에서 변경: *src* 및 *dst*로 경로류 객체를 받아들입니다.

os.**listdir**(*path='.'*)

*path*에 의해 주어진 디렉터리에 있는 항목들의 이름을 담고 있는 리스트를 반환합니다. 리스트는 임의의 순서로 나열되며, 디렉터리에 존재하더라도 특수 항목 `'.'` 과 `'..'`는 포함하지 않습니다. 이 함수 호출 중에 디렉터리에서 파일이 제거되거나 추가되면, 해당 파일의 이름이 포함되는지는 지정되지 않습니다.

*path*는 경로류 객체 일 수 있습니다. *path* 가 `bytes` 형이면 (직접 또는 `PathLike` 인터페이스를 통해 간접적으로), 반환되는 파일명도 `bytes` 형입니다; 다른 모든 상황에서는 형 `str`이 됩니다.

이 함수는 또한 파일 기술자 지정을 지원할 수 있습니다; 파일 기술자는 디렉터리를 참조해야 합니다.

`path`를 인자로 감사 이벤트(*auditing event*) `os.listdir`을 발생시킵니다.

> **참고**
>
> `str` 파일명을 `bytes`로 인코딩하려면, `fsencode()`를 사용하십시오.

> **더 보기**
>
> `scandir()` 함수는 파일 어트리뷰트 정보와 함께 디렉터리 항목을 반환하므로, 많은 일반적인 사용 사례에서 더 나은 성능을 제공합니다.

버전 3.2에서 변경: *path* 매개 변수는 선택 사항이 되었습니다.

버전 3.3에서 변경: *path*에 열린 파일 기술자를 지정하는 지원이 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**listdrives**()

Return a list containing the names of drives on a Windows system.

A drive name typically looks like `'C:\\'`. Not every drive name will be associated with a volume, and some may be inaccessible for a variety of reasons, including permissions, network connectivity or missing media. This function does not test for access.

May raise `OSError` if an error occurs collecting the drive names.

Raises an *auditing event* `os.listdrives` with no arguments.

가용성: Windows

Added in version 3.12.

os.**listmounts**(*volume*)

Return a list containing the mount points for a volume on a Windows system.

volume must be represented as a GUID path, like those returned by `os.listvolumes()`. Volumes may be mounted in multiple locations or not at all. In the latter case, the list will be empty. Mount points that are not associated with a volume will not be returned by this function.

The mount points return by this function will be absolute paths, and may be longer than the drive name.

Raises `OSError` if the volume is not recognized or if an error occurs collecting the paths.

Raises an *auditing event* `os.listmounts` with argument `volume`.

가용성: Windows

Added in version 3.12.

os.**listvolumes**()

Return a list containing the volumes in the system.

Volumes are typically represented as a GUID path that looks like `\\?\Volume{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}\`. Files can usually be accessed through a GUID path, permissions allowing. However, users are generally not familiar with them, and so the recommended use of this function is to retrieve mount points using `os.listmounts()`.

May raise `OSError` if an error occurs collecting the volumes.

Raises an *auditing event* `os.listvolumes` with no arguments.

가용성: Windows

Added in version 3.12.

os.**lstat**(*path*, *, *dir_fd=None*)

Perform the equivalent of an `lstat()` system call on the given path. Similar to `stat()`, but does not follow symbolic links. Return a `stat_result` object.

심볼릭 링크를 지원하지 않는 플랫폼에서, 이 함수는 `stat()`의 별칭입니다.

파이썬 3.3부터는, `os.stat(path, dir_fd=dir_fd, follow_symlinks=False)`와 같습니다.

이 기능은 디렉터리 기술자에 상대적인 경로도 지원할 수 있습니다.

더 보기

`stat()` 함수.

버전 3.2에서 변경: 윈도우 6.0 (Vista) 심볼릭 링크에 대한 지원이 추가되었습니다.

버전 3.3에서 변경: *dir_fd* 매개 변수가 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.8에서 변경: 윈도우에서, 이제 심볼릭 링크와 디렉터리 정션(directory junction)을 포함하는 다른 경로를 나타내는 재해석 지점(reparse point, 이름 서로게이트)을 엽니다. 다른 유형의 재해석 지점은 운영 체제에서 `stat()`에서 처럼 결정됩니다.

`os.`**`mkdir`**(*path*, *mode=0o777*, **, dir_fd=None*)

숫자 모드 *mode*로 *path* 라는 디렉터리를 만듭니다.

If the directory already exists, `FileExistsError` is raised. If a parent directory in the path does not exist, `FileNotFoundError` is raised.

일부 시스템에서는, *mode*가 무시됩니다. 모드가 사용될 때, 현재 umask 값으로 먼저 마스킹합니다. 마지막 9비트 (즉, *mode* 의 8진 표현의 마지막 3자리 수) 이외의 비트가 설정되면, 그 의미는 플랫폼에 따라 다릅니다. 일부 플랫폼에서는, 이것들이 무시되며, 설정하려면 명시적으로 `chmod()`를 호출해야 합니다.

On Windows, a *mode* of `0o700` is specifically handled to apply access control to the new directory such that only the current user and administrators have access. Other values of *mode* are ignored.

이 기능은 디렉터리 기술자에 상대적인 경로도 지원할 수 있습니다.

임시 디렉터리를 만들 수도 있습니다; `tempfile` 모듈의 `tempfile.mkdtemp()` 함수를 참조하십시오.

`path`, `mode`, `dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.mkdir`을 발생시킵니다.

버전 3.3에서 변경: *dir_fd* 매개 변수가 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.13에서 변경: Windows now handles a *mode* of `0o700`.

`os.`**`makedirs`**(*name*, *mode=0o777*, *exist_ok=False*)

재귀적 디렉터리 생성 함수. `mkdir()`와 비슷하지만, 말단 디렉터리를 포함하는 데 필요한 모든 중간 수준 디렉터리들을 만듭니다.

The *mode* parameter is passed to `mkdir()` for creating the leaf directory; see *the mkdir() description* for how it is interpreted. To set the file permission bits of any newly created parent directories you can set the umask before invoking `makedirs()`. The file permission bits of existing parent directories are not changed.

If *exist_ok* is `False` (the default), a `FileExistsError` is raised if the target directory already exists.

> **참고**
>
> `makedirs()`는 생성할 경로 요소에 `pardir`(예를 들어, 유닉스 시스템의 경우 "..")이 포함되어 있으면 혼란해 할 수 있습니다.

이 함수는 UNC 경로를 올바르게 처리합니다.

`path`, `mode`, `dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.mkdir`을 발생시킵니다.

버전 3.2에서 변경: Added the *exist_ok* parameter.

버전 3.4.1에서 변경: 파이썬 3.4.1 이전에는, *exist_ok* 가 `True`이고 디렉터리가 존재한다면, *mode* 가 기존 디렉터리의 모드와 일치하지 않을 때, `makedirs()`는 여전히 에러를 발생시킵니다. 이 동작은 안전하게 구현할 수 없으므로, 파이썬 3.4.1에서 제거되었습니다. bpo-21082를 참조하십시오.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.7에서 변경: The *mode* argument no longer affects the file permission bits of newly created intermediate-level directories.

os.**mkfifo**(*path*, *mode=0o666*, **, dir_fd=None*)

숫자 모드 *mode*로 *path* 라는 이름의 FIFO(이름있는 파이프)를 만듭니다. 현재 umask 값으로 먼저 모드를 마스킹합니다.

이 기능은 디렉터리 기술자에 상대적인 경로도 지원할 수 있습니다.

FIFO는 일반 파일처럼 액세스할 수 있는 파이프입니다. FIFO는 삭제될 때까지 존재합니다 (예를 들어 `os.unlink()`로). 일반적으로, FIFO는 "클라이언트"와 "서버" 유형 프로세스 사이에서 랑데부로 사용됩니다: 서버는 FIFO를 읽기 용도로 열고, 클라이언트는 쓰기 용도로 엽니다. `mkfifo()`가 FIFO를 열지는 않는다는 점에 유의하십시오 — 단지 랑데부 포인트를 생성합니다.

가용성: Unix, not WASI.

버전 3.3에서 변경: *dir_fd* 매개 변수가 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**mknod**(*path*, *mode=0o600*, *device=0*, **, dir_fd=None*)

path 라는 이름의 파일 시스템 노드(파일, 장치 특수 파일 또는 이름있는 파이프)를 만듭니다. *mode* 는 사용 권한과 생성될 노드의 유형을 모두 지정하며, `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK` 및 `stat.S_IFIFO` 중 하나와 결합(비트별 OR) 합니다 (이 상수들은 `stat`에 있습니다). `stat.S_IFCHR`와 `stat.S_IFBLK`의 경우, *device* 는 새로 만들어지는 장치 특수 파일(아마도 `os.makedev()`를 사용해서)을 정의합니다, 그렇지 않으면 무시됩니다.

이 기능은 디렉터리 기술자에 상대적인 경로도 지원할 수 있습니다.

가용성: Unix, not WASI.

버전 3.3에서 변경: *dir_fd* 매개 변수가 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**major**(*device*, */*)

Extract the device major number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

os.**minor**(*device*, */*)

Extract the device minor number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

os.**makedev**(*major*, *minor*, */*)

주 장치 번호와 부 장치 번호로 원시 장치 번호를 조립합니다.

os.**pathconf**(*path*, *name*)

이름있는 파일과 관련된 시스템 구성 정보를 반환합니다. *name* 은 조회할 구성 값을 지정합니다; 정의된 시스템 값의 이름인 문자열일 수 있습니다; 이 이름은 여러 표준(POSIX.1, 유닉스 95, 유닉스 98 및 기타)에서 지정됩니다. 일부 플랫폼은 추가적인 이름도 정의합니다. 호스트 운영 체제에 알려진 이름은 `pathconf_names` 딕셔너리에서 제공됩니다. 이 매핑에 포함되지 않은 구성 변수를 위해, *name*에 정수를 전달하는 것도 허용됩니다.

name 이 문자열이고 알 수 없으면, `ValueError`가 발생합니다. *name*에 대한 특정 값이 호스트 시스템에서 지원되지 않으면, `pathconf_names`에 포함되어 있어도, 에러 번호가 `errno.EINVAL`인 `OSError`가 발생합니다.

이 함수는 파일 기술자 지정을 지원할 수 있습니다.

가용성: Unix.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**pathconf_names**

`pathconf()`와 `fpathconf()`가 받아들이는 이름을 호스트 운영 체제에서 해당 이름에 대해 정의된 정숫값으로 매핑하는 딕셔너리. 이것은 시스템에 알려진 이름 집합을 판별하는 데 사용될 수 있습니다.

가용성: Unix.

os.**readlink**(*path*, *, *dir_fd=None*)

심볼릭 링크가 가리키는 경로를 나타내는 문자열을 반환합니다. 결과는 절대 또는 상대 경로명일 수 있습니다; 상대 경로이면 `os.path.join(os.path.dirname(path), result)`를 사용하여 절대 경로명으로 변환할 수 있습니다.

path 가 (직접 또는 `PathLike` 인터페이스를 통해 간접적으로) 문자열 객체면, 결과도 문자열 객체가 되고, 호출은 UnicodeDecodeError를 발생시킬 수 있습니다. *path* 가 (직접 또는 간접적으로) 바이트열 객체면, 결과는 바이트열 객체가 됩니다.

이 기능은 디렉터리 기술자에 상대적인 경로도 지원할 수 있습니다.

링크를 포함할 수 있는 경로를 결정(resolve)하려고 할 때, `realpath()`를 사용하여 재귀와 플랫폼 차이를 올바르게 처리하십시오.

가용성: Unix, Windows.

버전 3.2에서 변경: 윈도우 6.0 (Vista) 심볼릭 링크에 대한 지원이 추가되었습니다.

버전 3.3에서 변경: *dir_fd* 매개 변수가 추가되었습니다.

버전 3.6에서 변경: 유닉스에서 경로류 객체를 받아들입니다.

버전 3.8에서 변경: 윈도우에서 경로류 객체와 바이트열 객체를 받아들입니다.

디렉터리 정션(directory junction)에 대한 지원이 추가되었고, 이전에 반환되던 선택적 "print name" 필드 대신 치환 경로(일반적으로 `\\?\` 접두사를 포함합니다)를 반환하도록 변경되었습니다.

os.**remove**(*path*, *, *dir_fd=None*)

Remove (delete) the file *path*. If *path* is a directory, an `OSError` is raised. Use `rmdir()` to remove directories. If the file does not exist, a `FileNotFoundError` is raised.

이 함수는 디렉터리 기술자에 상대적인 경로를 지원할 수 있습니다.

윈도우에서, 사용 중인 파일을 제거하려고 시도하면 예외가 발생합니다; 유닉스에서는 디렉터리 항목이 제거되지만, 원본 파일이 더는 사용되지 않을 때까지 파일에 할당된 저장 공간을 사용할 수 없습니다.

이 함수는 의미 적으로 `unlink()`와 같습니다.

`path`, `dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.remove`를 발생시킵니다.

버전 3.3에서 변경: *dir_fd* 매개 변수가 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**removedirs**(*name*)

재귀적으로 디렉터리를 제거합니다. `rmdir()` 처럼 동작하는데 다음과 같은 차이가 있습니다. 말단 디렉터리가 성공적으로 제거되면, `removedirs()`는 에러가 발생할 때까지 *path*에 언급된 모든 상위 디렉터리를 연속적으로 제거하려고 합니다 (에러는 무시되는데, 이는 일반적으로 부모 디렉터리가 비어 있음을 뜻하기 때문입니다). 예를 들어, `os.removedirs('foo/bar/baz')`는 먼저 `'foo/bar/baz'` 디렉터리를 제거한 다음, `'foo/bar'` 및 `'foo'`가 비어 있으면 제거합니다. 말단 디렉터리를 성공적으로 제거할 수 없으면, `OSError`를 발생시킵니다.

`path`, `dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.remove`를 발생시킵니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**rename**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)

파일 또는 디렉터리 *src*의 이름을 *dst*로 바꿉니다. *dst*가 존재하면, 많은 경우에 `OSError` 서브 클래스로 연산이 실패합니다:

On Windows, if *dst* exists a `FileExistsError` is always raised. The operation may fail if *src* and *dst* are on different filesystems. Use `shutil.move()` to support moves to a different filesystem.

On Unix, if *src* is a file and *dst* is a directory or vice-versa, an `IsADirectoryError` or a `NotADirectoryError` will be raised respectively. If both are directories and *dst* is empty, *dst* will be silently replaced. If *dst* is a non-empty directory, an `OSError` is raised. If both are files, *dst* will be replaced silently if

the user has permission. The operation may fail on some Unix flavors if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

이 함수는 디렉터리 기술자에 상대적인 경로를 제공하도록 *src_dir_fd* 와/나 *dst_dir_fd* 를 지정하는 것을 지원할 수 있습니다.

플랫폼에 무관하게 대상을 덮어쓰길 원하면, `replace()`를 사용하십시오.

`src`, `dst`, `src_dir_fd`, `dst_dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.rename`을 발생시킵니다.

버전 3.3에서 변경: Added the *src_dir_fd* and *dst_dir_fd* parameters.

버전 3.6에서 변경: *src* 및 *dst*로 경로류 객체를 받아들입니다.

`os.`**`renames`**(*old*, *new*)

재귀적 디렉터리 또는 파일 이름 바꾸기 함수. `rename()`처럼 작동하지만, 새 경로명이 유효하도록 만들기 위해 먼저 필요한 중간 디렉터리를 만드는 점이 다릅니다. 이름을 변경한 후에는, 이전 이름의 가장 오른쪽 경로 세그먼트에 해당하는 디렉터리를 `removedirs()`를 사용하여 제거합니다.

> **참고**
>
> 이 함수는 말단 디렉터리나 파일을 제거하는 데 필요한 권한이 없을 때, 새 디렉터리 구조를 만든 상태에서 실패할 수 있습니다.

`src`, `dst`, `src_dir_fd`, `dst_dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.rename`을 발생시킵니다.

버전 3.6에서 변경: *old* 와 *new* 에 경로류 객체를 받아들입니다.

`os.`**`replace`**(*src*, *dst*, **, src_dir_fd=None, dst_dir_fd=None*)

Rename the file or directory *src* to *dst*. If *dst* is a non-empty directory, `OSError` will be raised. If *dst* exists and is a file, it will be replaced silently if the user has permission. The operation may fail if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

이 함수는 디렉터리 기술자에 상대적인 경로를 제공하도록 *src_dir_fd* 와/나 *dst_dir_fd* 를 지정하는 것을 지원할 수 있습니다.

`src`, `dst`, `src_dir_fd`, `dst_dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.rename`을 발생시킵니다.

Added in version 3.3.

버전 3.6에서 변경: *src* 및 *dst*로 경로류 객체를 받아들입니다.

`os.`**`rmdir`**(*path*, **, dir_fd=None*)

Remove (delete) the directory *path*. If the directory does not exist or is not empty, a `FileNotFoundError` or an `OSError` is raised respectively. In order to remove whole directory trees, `shutil.rmtree()` can be used.

이 함수는 디렉터리 기술자에 상대적인 경로를 지원할 수 있습니다.

`path`, `dir_fd`를 인자로 감사 이벤트(*auditing event*) `os.rmdir`을 발생시킵니다.

버전 3.3에서 변경: *dir_fd* 매개 변수가 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

`os.`**`scandir`**(*path='.'*)

*path*로 지정된 디렉터리 내의 항목에 대응하는 `os.DirEntry` 객체의 이터레이터를 돌려줍니다. 항목은 임의의 순서로 제공되며, 특수 항목 `'.'` 및 `'..'`는 포함되지 않습니다. 이터레이터를 만든 후에 디렉터리에서 파일이 제거되거나 추가되면, 해당 파일의 항목이 포함되는지는 지정되지 않습니다.

`listdir()` 대신 `scandir()`를 사용하면, 디렉터리를 검색할 때 운영 체제가 제공한다면 `os.DirEntry` 객체가 파일 유형과 파일 어트리뷰트 정보를 제공하기 때문에, 이것들이 필요한 코드의 성능을 크게 개선할 수 있습니다. 모든 `os.DirEntry` 메서드가 시스템 호출을 수행할 수 있지만,

일반적으로 *is_dir()* 및 *is_file()*는 심볼릭 링크에 대해서만 시스템 호출을 요구합니다; *os.DirEntry.stat()*는 유닉스에서 항상 시스템 호출을 요구하지만 윈도우에서는 심볼릭 링크에 대해서만 시스템 호출을 요구합니다.

path 는 경로류 객체 일 수 있습니다. *path* 가 (직접 또는 *PathLike* 인터페이스를 통해 간접적으로) bytes 형이면, 각 *os.DirEntry*의 *name* 및 *path* 어트리뷰트의 형은 bytes 입니다. 다른 모든 상황에서는 형 str 이 됩니다.

이 함수는 또한 파일 기술자 지정을 지원할 수 있습니다; 파일 기술자는 디렉터리를 참조해야 합니다.

path를 인자로 감사 이벤트(*auditing event*) os.scandir을 발생시킵니다.

scandir() 이터레이터는 컨텍스트 관리자 프로토콜을 지원하고 다음과 같은 메서드를 제공합니다:

scandir.**close**()

이터레이터를 닫고 확보한 자원을 반납합니다.

이터레이터가 소진되거나 가비지 수집될 때 또는 이터레이션 중에 에러가 발생하면 자동으로 호출됩니다. 하지만 명시적으로 호출하거나 with 문을 사용하는 것이 좋습니다.

Added in version 3.6.

다음 예제는 주어진 *path*의 '.'로 시작하지 않는 모든 파일(디렉터리 제외)을 표시하기 위한 *scandir()*의 간단한 사용을 보여줍니다. entry.is_file() 호출은 일반적으로 추가 시스템 호출을 하지 않습니다:

```
with os.scandir(path) as it:
    for entry in it:
        if not entry.name.startswith('.') and entry.is_file():
            print(entry.name)
```

참고

On Unix-based systems, *scandir()* uses the system' s opendir() and readdir() functions. On Windows, it uses the Win32 FindFirstFileW and FindNextFileW functions.

Added in version 3.5.

버전 3.6에서 변경: Added support for the *context manager* protocol and the *close()* method. If a *scandir()* iterator is neither exhausted nor explicitly closed a *ResourceWarning* will be emitted in its destructor.

이 함수는 경로류 객체를 받아들입니다.

버전 3.7에서 변경: 유닉스에서 파일 기술자에 대한 지원이 추가되었습니다.

class os.**DirEntry**

디렉터리 항목의 파일 경로와 다른 파일 어트리뷰트를 노출하기 위해 *scandir()*에 의해 산출되는 객체.

*scandir()*는 추가 시스템 호출 없이 가능한 많은 정보를 제공합니다. stat() 또는 lstat() 시스템 호출이 이루어지면, os.DirEntry 객체는 결과를 캐시 합니다.

os.DirEntry 인스턴스는 수명이 긴 데이터 구조에 저장하는 용도가 아닙니다; 파일 메타 데이터가 변경되었거나 *scandir()*를 호출한 후 오랜 시간이 지났음을 안다면, os.stat(entry.path) 를 호출하여 최신 정보를 가져오십시오.

os.DirEntry 메서드는 운영 체제 시스템 호출을 할 수 있으므로, *OSError*를 일으킬 수도 있습니다. 에러에 대해 매우 세부적인 제어가 필요하면, os.DirEntry 메서드 중 하나를 호출할 때 *OSError*를 잡은 후 적절하게 처리할 수 있습니다.

경로류 객체로 직접 사용할 수 있도록, os.DirEntry는 *PathLike* 인터페이스를 구현합니다.

os.DirEntry 인스턴스의 어트리뷰트 및 메서드는 다음과 같습니다:

name

scandir() *path* 인자에 상대적인, 항목의 기본(base) 파일명.

name 어트리뷰트는 *scandir()* *path* 인자가 bytes 형이면 bytes 고, 그렇지 않으면 str 입니다. 바이트열 파일명을 디코딩하려면 *fsdecode()*를 사용하십시오.

path

항목의 전체 경로명: os.path.join(scandir_path, entry.name) 과 같습니다. 여기서 *scandir_path* 는 *scandir()* *path* 인자입니다. 경로는 *scandir()* *path* 인자가 절대 경로일 때만 절대 경로입니다. *scandir()* *path* 인자가 파일 기술자면, *path* 어트리뷰트는 *name* 어트리뷰트와 같습니다.

path 어트리뷰트는 *scandir()* *path* 인자가 bytes 형이면 bytes고, 그렇지 않으면 str 입니다. 바이트열 파일명을 디코딩하려면 *fsdecode()*를 사용하십시오.

inode()

항목의 아이노드(inode) 번호를 반환합니다.

결과는 os.DirEntry 객체에 캐시 됩니다. 최신 정보를 가져오려면 os.stat(entry.path, follow_symlinks=False).st_ino를 사용하십시오.

최초의 캐시 되지 않은 호출에서, 윈도우 에서는 시스템 호출이 필요하지만, 유닉스에서는 그렇지 않습니다.

is_dir(*, *follow_symlinks=True*)

이 항목이 디렉터리 또는 디렉터리를 가리키는 심볼릭 링크면 True를 반환합니다; 항목이 다른 종류의 파일이거나 다른 종류의 파일을 가리키면, 또는 더는 존재하지 않으면 False를 반환합니다.

follow_symlinks 가 False면, 이 항목이 디렉터리일 때만 (심볼릭 링크를 따르지 않고) True를 반환합니다; 항목이 다른 종류의 파일이거나 더는 존재하지 않으면 False를 반환합니다.

결과는 *follow_symlinks* 가 True 및 False일 때에 대해 별도로 os.DirEntry 객체에 캐시 됩니다. 최신 정보를 가져오려면, *stat.S_ISDIR()*로 *os.stat()*을 호출하십시오.

최초의 캐시 되지 않은 호출에서, 대부분 시스템 호출이 필요하지 않습니다. 특히, 심볼릭 링크가 아니면, 윈도우나 유닉스 모두 시스템 호출이 필요하지 않은데, 네트워크 파일 시스템과 같이 dirent.d_type == DT_UNKNOWN를 반환하는 특정 유닉스 파일 시스템은 예외입니다. 항목이 심볼릭 링크면, *follow_symlinks* 가 False가 아닌 이상, 심볼릭 링크를 따르기 위해 시스템 호출이 필요합니다.

이 메서드는, *PermissionError*와 같은, *OSError*를 발생시킬 수 있지만, *FileNotFoundError*는 잡혀서 발생하지 않습니다.

is_file(*, *follow_symlinks=True*)

이 항목이 파일이나 파일을 가리키는 심볼릭 링크면 True를 반환합니다; 항목이 디렉터리 또는 다른 비 파일 항목이거나, 그런 것을 가리키거나, 더는 존재하지 않으면 False를 반환합니다.

follow_symlinks 가 False면, 이 항목이 파일일 때만 (심볼릭 링크를 따르지 않고) True를 반환합니다; 항목이 디렉터리 나 다른 비 파일 항목이거나 더는 존재하지 않으면 False를 반환합니다.

결과는 os.DirEntry 객체에 캐시 됩니다. 캐싱, 시스템 호출, 예외 발생은 *is_dir()*과 같습니다.

is_symlink()

이 항목이 심볼릭 링크면 (망가졌다 하더라도) True를 반환합니다; 항목이 디렉터리 나 어떤 종류의 파일이거나 더는 존재하지 않으면 False를 반환합니다.

결과는 os.DirEntry 객체에 캐시 됩니다. 최신 정보를 가져오려면 *os.path.islink()*를 호출하십시오.

첫 번째, 캐시 되지 않은 호출에서는 시스템 호출이 필요하지 않습니다. 특히 윈도우 나 유닉스는 dirent.d_type == DT_UNKNOWN를 반환하는 특정 유닉스 파일 시스템 (예 : 네트워크 파일 시스템)을 제외하고는 시스템 호출이 필요하지 않습니다.

이 메서드는, *PermissionError*와 같은, *OSError*를 발생시킬 수 있지만, *FileNotFoundError*는 잡혀서 발생하지 않습니다.

is_junction()

Return `True` if this entry is a junction (even if broken); return `False` if the entry points to a regular directory, any kind of file, a symlink, or if it doesn' t exist anymore.

The result is cached on the `os.DirEntry` object. Call *os.path.isjunction()* to fetch up-to-date information.

Added in version 3.12.

stat(*, *follow_symlinks=True*)

이 항목의 *stat_result* 객체를 돌려줍니다. 이 메서드는 기본적으로 심볼릭 링크를 따릅니다; 심볼릭 링크를 stat 하려면, `follow_symlinks=False` 인자를 추가하십시오.

유닉스에서, 이 메서드는 항상 시스템 호출을 요구합니다. 윈도우에서, *follow_symlinks* 가 `True` 이고 항목이 재해석 지점(reparse point, 예를 들어, 심볼릭 링크나 디렉터리 정션(directory junction)) 일 때만 시스템 호출이 필요합니다.

윈도우에서, *stat_result*의 `st_ino`, `st_dev` 및 `st_nlink` 어트리뷰트는 항상 0으로 설정됩니다. 이러한 어트리뷰트를 얻으려면 *os.stat()*을 호출하십시오.

결과는 *follow_symlinks* 가 `True` 및 `False` 일 때에 대해 별도로 `os.DirEntry` 객체에 캐시 됩니다. 최신 정보를 가져오려면, *os.stat()*을 호출하십시오.

Note that there is a nice correspondence between several attributes and methods of `os.DirEntry` and of *pathlib.Path*. In particular, the `name` attribute has the same meaning, as do the `is_dir()`, `is_file()`, `is_symlink()`, `is_junction()`, and `stat()` methods.

Added in version 3.5.

버전 3.6에서 변경: *PathLike* 인터페이스에 대한 지원이 추가되었습니다. 윈도우에서 *bytes* 경로에 대한 지원이 추가되었습니다.

버전 3.12에서 변경: The `st_ctime` attribute of a stat result is deprecated on Windows. The file creation time is properly available as `st_birthtime`, and in the future `st_ctime` may be changed to return zero or the metadata change time, if available.

`os.`**stat**(*path*, *, *dir_fd=None*, *follow_symlinks=True*)

파일 또는 파일 기술자의 상태를 가져옵니다. 주어진 경로에 대해 `stat()` 시스템 호출과 같은 작업을 수행합니다. *path* 는 문자열이나 바이트열 – 직접 또는 *PathLike* 인터페이스를 통해 간접적으로 – 또는 열린 파일 기술자로 지정될 수 있습니다. *stat_result* 객체를 반환합니다.

이 함수는 일반적으로 심볼릭 링크를 따릅니다; 심볼릭 링크를 stat 하려면, 인자 `follow_symlinks=False`를 추가하거나 *lstat()*를 사용하십시오.

이 함수는 파일 기술자 지정 및 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

윈도우에서, `follow_symlinks=False`를 전달하면 심볼릭 링크와 디렉터리 정션(directory junction)을 포함하는 모든 이름 서로게이트(name-surrogate) 재해석 지점(reparse point)을 따라가지 않습니다. 링크처럼 보이지 않거나 운영 체제에서 따라갈 수 없는 다른 유형의 재해석 지점은 직접 열립니다. 여러 링크 체인을 따라갈 때, 전체 탐색을 방해하는 비 링크 대신 원래 링크가 반환될 수 있습니다. 이 경우 최종 경로에 대한 stat 결과를 얻으려면, *os.path.realpath()* 함수를 사용하여 가능한 한 멀리 간 경로 이름을 확인한 다음 드 결과에 대해 *lstat()*을 호출하십시오. 매달린(dangling) 심볼릭 링크나 정션 지점에는 적용되지 않고, 일반적인 예외가 발생합니다.

예:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
os.stat_result(st_mode=33188, st_ino=7876932, st_dev=234881026,
st_nlink=1, st_uid=501, st_gid=501, st_size=264, st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> statinfo.st_size
264
```

더 보기

fstat() 및 *lstat()* 함수.

버전 3.3에서 변경: Added the *dir_fd* and *follow_symlinks* parameters, specifying a file descriptor instead of a path.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.8에서 변경: 윈도우에서, 이제 운영 체제가 결정할 수 있는 모든 재해석 지점을 따라가고, `follow_symlinks=False`를 전달하면 모든 이름 서로게이트 재해석 지점을 따라가지 않습니다. 운영 체제가 따라갈 수 없는 재해석 지점에 도달하면, *stat*은 이제 에러를 발생시키는 대신 `follow_symlinks=False`가 지정된 것처럼 원래 경로에 대한 정보를 반환합니다.

class os.**stat_result**

Object whose attributes correspond roughly to the members of the `stat` structure. It is used for the result of *os.stat()*, *os.fstat()* and *os.lstat()*.

어트리뷰트:

st_mode

파일 모드: 파일 유형 및 파일 모드 비트 (사용 권한).

st_ino

플랫폼에 따라 다르지만, 0이 아니면, 지정된 값의 `st_dev`은 파일을 고유하게 식별합니다. 일반적으로:

- 유닉스의 아이노드 번호,
- 윈도우의 파일 인덱스

st_dev

이 파일이 있는 장치의 식별자.

st_nlink

하드 링크 수.

st_uid

파일 소유자의 사용자 식별자.

st_gid

파일 소유자의 그룹 식별자.

st_size

일반 파일 또는 심볼릭 링크면, 바이트 단위의 파일의 크기. 심볼릭 링크의 크기는 포함하고 있는 경로명의 길이이며, 끝나는 널 바이트는 포함하지 않습니다.

타임스탬프:

st_atime

초 단위의 가장 최근의 액세스 시간.

st_mtime

초 단위의 가장 최근의 내용 수정 시간.

st_ctime

Time of most recent metadata change expressed in seconds.

버전 3.12에서 변경: `st_ctime` is deprecated on Windows. Use `st_birthtime` for the file creation time. In the future, `st_ctime` will contain the time of the most recent metadata change, as for other platforms.

st_atime_ns

나노초 정수 단위의 가장 최근의 액세스 시간.

Added in version 3.3.

st_mtime_ns

나노초 정수 단위의 가장 최근의 내용 수정 시간.

Added in version 3.3.

st_ctime_ns

Time of most recent metadata change expressed in nanoseconds as an integer.

Added in version 3.3.

버전 3.12에서 변경: `st_ctime_ns` is deprecated on Windows. Use `st_birthtime_ns` for the file creation time. In the future, `st_ctime` will contain the time of the most recent metadata change, as for other platforms.

st_birthtime

Time of file creation expressed in seconds. This attribute is not always available, and may raise *AttributeError*.

버전 3.12에서 변경: `st_birthtime` is now available on Windows.

st_birthtime_ns

Time of file creation expressed in nanoseconds as an integer. This attribute is not always available, and may raise *AttributeError*.

Added in version 3.12.

> **참고**
>
> The exact meaning and resolution of the *st_atime*, *st_mtime*, *st_ctime* and *st_birthtime* attributes depend on the operating system and the file system. For example, on Windows systems using the FAT32 file systems, *st_mtime* has 2-second resolution, and *st_atime* has only 1-day resolution. See your operating system documentation for details.
>
> Similarly, although *st_atime_ns*, *st_mtime_ns*, *st_ctime_ns* and *st_birthtime_ns* are always expressed in nanoseconds, many systems do not provide nanosecond precision. On systems that do provide nanosecond precision, the floating-point object used to store *st_atime*, *st_mtime*, *st_ctime* and *st_birthtime* cannot preserve all of it, and as such will be slightly inexact. If you need the exact timestamps you should always use *st_atime_ns*, *st_mtime_ns*, *st_ctime_ns* and *st_birthtime_ns*.

(리눅스와 같은) 일부 유닉스 시스템에서는, 다음 어트리뷰트도 사용할 수 있습니다:

st_blocks

파일에 할당된 512-바이트 블록 수. 파일에 구멍이 있으면 *st_size*/512보다 작을 수 있습니다.

st_blksize

효율적인 파일 시스템 I/O를 위해 "선호되는" 블록 크기. 더 작은 크기로 파일에 기록하면 비효율적인 읽기-수정-다시 쓰기가 발생할 수 있습니다.

st_rdev

아이노드 장치면 장치 유형.

st_flags

파일에 대한 사용자 정의 플래그.

(FreeBSD와 같은) 다른 유닉스 시스템에서는, 다음 어트리뷰트를 사용할 수 있습니다 (그러나 root가 사용하려고 할 때만 채워질 수 있습니다):

st_gen

파일 생성 번호.

Solaris 및 파생 상품에서, 다음 어트리뷰트도 사용할 수 있습니다:

st_fstype

파일을 포함하는 파일 시스템의 유형을 고유하게 식별하는 문자열.

On macOS systems, the following attributes may also be available:

st_rsize

파일의 실제 크기.

st_creator

파일의 생성자.

st_type

파일 유형.

윈도우 시스템에서는, 다음 어트리뷰트도 사용할 수 있습니다:

st_file_attributes

Windows file attributes: `dwFileAttributes` member of the `BY_HANDLE_FILE_INFORMATION` structure returned by `GetFileInformationByHandle()`. See the `FILE_ATTRIBUTE_* <stat.FILE_ATTRIBUTE_ARCHIVE>` constants in the *stat* module.

Added in version 3.5.

st_reparse_tag

When *st_file_attributes* has the *FILE_ATTRIBUTE_REPARSE_POINT* set, this field contains the tag identifying the type of reparse point. See the *IO_REPARSE_TAG_** constants in the *stat* module.

The standard module *stat* defines functions and constants that are useful for extracting information from a `stat` structure. (On Windows, some items are filled with dummy values.)

For backward compatibility, a *stat_result* instance is also accessible as a tuple of at least 10 integers giving the most important (and portable) members of the `stat` structure, in the order *st_mode*, *st_ino*, *st_dev*, *st_nlink*, *st_uid*, *st_gid*, *st_size*, *st_atime*, *st_mtime*, *st_ctime*. More items may be added at the end by some implementations. For compatibility with older Python versions, accessing *stat_result* as a tuple always returns integers.

버전 3.5에서 변경: 윈도우는 이제 사용 가능할 때 파일 인덱스를 *st_ino*로 반환합니다.

버전 3.7에서 변경: Solaris/파생 제품에 *st_fstype* 멤버를 추가했습니다.

버전 3.8에서 변경: 윈도우에서 *st_reparse_tag* 멤버를 추가했습니다.

버전 3.8에서 변경: 윈도우에서, *st_mode* 멤버는 이제 특수 파일을 `S_IFCHR`, `S_IFIFO` 또는 `S_IFBLK`로 적절히 식별합니다.

버전 3.12에서 변경: On Windows, *st_ctime* is deprecated. Eventually, it will contain the last metadata change time, for consistency with other platforms, but for now still contains creation time. Use *st_birthtime* for the creation time.

On Windows, *st_ino* may now be up to 128 bits, depending on the file system. Previously it would not be above 64 bits, and larger file identifiers would be arbitrarily packed.

On Windows, *st_rdev* no longer returns a value. Previously it would contain the same as *st_dev*, which was incorrect.

Added the *st_birthtime* member on Windows.

os.**statvfs**(*path*)

Perform a statvfs() system call on the given path. The return value is an object whose attributes describe the filesystem on the given path, and correspond to the members of the statvfs structure, namely: f_bsize, f_frsize, f_blocks, f_bfree, f_bavail, f_files, f_ffree, f_favail, f_flag, f_namemax, f_fsid.

f_flag 어트리뷰트의 비트 플래그에 대해 두 개의 모듈 수준 상수가 정의됩니다: ST_RDONLY가 설정되면, 파일 시스템은 읽기 전용으로 마운트되었고, ST_NOSUID가 설정되면, setuid/setgid 비트의 의미가 비활성화되었거나 지원되지 않습니다.

추가적인 모듈 수준 상수가 GNU/glibc 기반 시스템에 대해 정의됩니다. 이들은 ST_NODEV (장치 특수 파일에 대한 액세스 금지), ST_NOEXEC (프로그램 실행 금지), ST_SYNCHRONOUS (한 번에 쓰기 동기화), ST_MANDLOCK (FS에 필수 잠금 허용), ST_WRITE (파일/디렉터리/심볼릭 링크 쓰기), ST_APPEND (덧붙이기 전용 파일), ST_IMMUTABLE (불변 파일), ST_NOATIME (액세스 시간을 갱신하지 않음), ST_NODIRATIME (디렉터리 액세스 시간을 갱신하지 않음), ST_RELATIME (mtime/ctime에 상대적으로 atime을 갱신).

이 함수는 파일 기술자 지정을 지원할 수 있습니다.

가용성: Unix.

버전 3.2에서 변경: ST_RDONLY 및 ST_NOSUID 상수가 추가되었습니다.

버전 3.3에서 변경: *path*에 열린 파일 기술자를 지정하는 지원이 추가되었습니다.

버전 3.4에서 변경: ST_NODEV, ST_NOEXEC, ST_SYNCHRONOUS, ST_MANDLOCK, ST_WRITE, ST_APPEND, ST_IMMUTABLE, ST_NOATIME, ST_NODIRATIME 및 ST_RELATIME 상수가 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.7에서 변경: Added the f_fsid attribute.

os.**supports_dir_fd**

os 모듈의 어떤 함수가 *dir_fd* 매개 변수로 열린 파일 기술자를 받아들이는지를 나타내는 set 객체. 플랫폼마다 다른 기능을 제공하며, 파이썬이 *dir_fd* 매개 변수를 구현하는 데 사용하는 하부 기능이 파이썬이 지원하는 모든 플랫폼에서 제공되지는 않습니다. 일관성을 위해, *dir_fd*를 지원할 수도 있는 함수는 항상 매개 변수를 지정할 수 있도록 하지만, 로컬에서 사용할 수 없을 때, 기능을 사용하면 예외를 발생시킵니다. (*dir_fd*에 None을 지정하는 것은 모든 플랫폼에서 항상 지원됩니다.)

특정 함수가 *dir_fd* 매개 변수로 열린 파일 기술자를 받아들이는지 확인하려면, supports_dir_fd에 in 연산자를 사용하십시오. 예를 들어, 이 표현식은 로컬 플랫폼에서 os.stat()이 *dir_fd* 매개 변수로 열린 파일 기술자를 받아들이면 True로 평가됩니다:

```
os.stat in os.supports_dir_fd
```

현재 *dir_fd* 매개 변수는 유닉스 플랫폼에서만 작동합니다; 어느 것도 윈도우에서 작동하지 않습니다.

Added in version 3.3.

os.**supports_effective_ids**

os.access()가 로컬 플랫폼에서 *effective_ids* 매개 변수에 True를 지정하는 것을 허용하는지를 나타내는 set 객체. (*effective_ids*에 False를 지정하는 것은 모든 플랫폼에서 항상 지원됩니다.) 로컬 플랫폼이 지원하면, 컬렉션에 os.access()가 포함됩니다; 그렇지 않으면 비어있게 됩니다.

이 표현식은 로컬 플랫폼에서 os.access()가 effective_ids=True를 지원하면 True로 평가됩니다:

```
os.access in os.supports_effective_ids
```

현재 *effective_ids*는 유닉스 플랫폼에서만 지원됩니다; 윈도우에서는 작동하지 않습니다.

Added in version 3.3.

os.**supports_fd**

os 모듈의 어떤 함수가 로컬 플랫폼에서 자신의 *path* 매개 변수에 열린 파일 기술자를 지정하는 것을 허용하는지를 나타내는 set 객체. 플랫폼마다 다른 기능을 제공하며, 파이썬이 *path*로 열린 파일 기술자를 받아들이는 데 사용하는 하부 기능이 파이썬이 지원하는 모든 플랫폼에서 제공되지는 않습니다.

특정 함수가 *path* 매개 변수에 열린 파일 기술자를 지정할 수 있도록 허용하는지를 판단하려면, supports_fd에 in 연산자를 사용하십시오. 예를 들어, 이 표현식은 로컬 플랫폼에서 os.chdir() 가 *path*로 열린 파일 기술자를 받아들이면 True로 평가됩니다:

```
os.chdir in os.supports_fd
```

Added in version 3.3.

os.**supports_follow_symlinks**

os 모듈의 어떤 함수가 *follow_symlinks* 매개 변수로 False를 받아들이는지를 나타내는 set 객체. 플랫폼마다 다른 기능을 제공하며, 파이썬이 *follow_symlinks*를 구현하는 데 사용하는 하부 기능이 파이썬이 지원하는 모든 플랫폼에서 제공되지는 않습니다. 일관성을 위해, *follow_symlinks* 를 지원할 수도 있는 함수는 항상 매개 변수를 지정할 수 있도록 하지만, 로컬에서 사용할 수 없을 때, 기능이 사용되면 예외를 발생시킵니다. (*follow_symlinks*에 None을 지정하는 것은 모든 플랫폼에서 항상 지원됩니다.)

특정 함수가 *follow_symlinks* 매개 변수로 False를 받아들이는지 확인하려면, supports_follow_symlinks에 in 연산자를 사용하십시오. 예를 들어, 이 표현식은 로컬 플랫폼에서 os.stat()을 호출할 때 follow_symlinks=False를 지정할 수 있으면 True로 평가됩니다:

```
os.stat in os.supports_follow_symlinks
```

Added in version 3.3.

os.**symlink**(*src*, *dst*, *target_is_directory=False*, **, dir_fd=None*)

*src*를 가리키는 *dst* 라는 이름의 심볼릭 링크를 만듭니다.

윈도우에서, 심볼릭 링크는 파일이나 디렉터리를 나타내며, 동적으로 대상에 맞춰 변형되지 않습니다. 대상이 있으면, 일치하도록 심볼릭 링크의 유형이 만들어집니다. 그렇지 않으면, *target_is_directory* 가 True 면 심볼릭 링크가 디렉터리로 만들어지고, 그렇지 않으면 파일 심볼릭 링크(기본값)가 만들어집니다. 비 윈도우 플랫폼에서는 *target_is_directory* 가 무시됩니다.

이 함수는 디렉터리 기술자에 상대적인 경로를 지원할 수 있습니다.

> **참고**
>
> 최신 버전의 윈도우 10에서, 개발자 모드가 활성화되면, 권한이 없는 계정이 심볼릭 링크를 만들 수 있습니다. 개발자 모드를 사용할 수 없거나 활성화되어 있지 않으면, *SeCreateSymbolicLinkPrivilege* 권한이 필요하거나, 프로세스를 관리자로 실행해야 합니다.
>
> 권한이 없는 사용자가 함수를 호출하면 OSError가 발생합니다.

src, dst, dir_fd를 인자로 감사 이벤트(*auditing event*) os.symlink를 발생시킵니다.

가용성: Unix, Windows.

The function is limited on WASI, see 웹어셈블리 플랫폼 for more information.

버전 3.2에서 변경: 윈도우 6.0 (Vista) 심볼릭 링크에 대한 지원이 추가되었습니다.

버전 3.3에서 변경: Added the *dir_fd* parameter, and now allow *target_is_directory* on non-Windows platforms.

버전 3.6에서 변경: *src* 및 *dst*로 경로류 객체를 받아들입니다.

버전 3.8에서 변경: 개발자 모드가 있는 윈도우에서 권한 상승 없는(unelevated) 심볼릭 링크에 대한 지원이 추가되었습니다.

os.**sync**()

디스크에 모든 것을 쓰도록 강제합니다.

가용성: Unix.

Added in version 3.3.

os.**truncate**(*path*, *length*)

최대 *length* 바이트가 되도록 *path*에 해당하는 파일을 자릅니다.

이 함수는 파일 기술자 지정을 지원할 수 있습니다.

path, length를 인자로 감사 이벤트(*auditing event*) os.truncate를 발생시킵니다.

가용성: Unix, Windows.

Added in version 3.3.

버전 3.5에서 변경: 윈도우 지원 추가

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**unlink**(*path*, *, *dir_fd=None*)

파일 *path*를 제거(삭제)합니다. 이 함수는 의미상 `remove()`와 같습니다; unlink 라는 이름은 전통적인 유닉스 이름입니다. 자세한 내용은 `remove()` 설명서를 참조하십시오.

path, dir_fd를 인자로 감사 이벤트(*auditing event*) os.remove를 발생시킵니다.

버전 3.3에서 변경: *dir_fd* 매개 변수가 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**utime**(*path*, *times=None*, *, [*ns*,]*dir_fd=None*, *follow_symlinks=True*)

*path*로 지정된 파일의 액세스 및 수정 시간을 설정합니다.

`utime()`은 *times* 과 *ns* 라는 두 개의 선택적 매개 변수를 취합니다. *path*에 설정할 시간을 지정하며 다음과 같이 사용됩니다:

- *ns* 가 지정되면, (atime_ns, mtime_ns) 형식의 2-튜플이어야 하며, 각 멤버는 나노초를 나타내는 int입니다.
- *times* 가 None이 아니면, (atime, mtime) 형식의 2-튜플이어야 하며, 각 멤버는 초를 나타내는 int 또는 float입니다.
- *times* 가 None이고 *ns* 가 지정되지 않으면, ns=(atime_ns, mtime_ns)를 지정하는 것과 같은데, 두 시간 모두 현재 시각입니다.

times 와 *ns*에 모두 튜플을 지정하는 것은 에러입니다.

Note that the exact times you set here may not be returned by a subsequent `stat()` call, depending on the resolution with which your operating system records access and modification times; see `stat()`. The best way to preserve exact times is to use the *st_atime_ns* and *st_mtime_ns* fields from the `os.stat()` result object with the *ns* parameter to `utime()`.

이 함수는 파일 기술자 지정, 디렉터리 기술자에 상대적인 경로 및 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

path, times, ns, dir_fd를 인자로 감사 이벤트(*auditing event*) os.utime을 발생시킵니다.

버전 3.3에서 변경: *path*에 열린 파일 기술자를 지정하는 것과 *dir_fd* , *follow_symlinks* 및 *ns* 매개 변수 지원이 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**walk**(*top*, *topdown=True*, *onerror=None*, *followlinks=False*)

트리를 하향식 또는 상향식으로 탐색하여 디렉터리 트리에 있는 파일명을 생성합니다. 디렉터리 *top*을 루트로 하는 트리의 디렉터리(*top* 자체를 포함합니다)마다, 3-튜플 (dirpath, dirnames, filenames)를 산출합니다.

dirpath is a string, the path to the directory. *dirnames* is a list of the names of the subdirectories in *dirpath* (including symlinks to directories, and excluding `'.'` and `'..'`). *filenames* is a list of the names of the non-directory files in *dirpath*. Note that the names in the lists contain no path components. To get a full path (which begins with *top*) to a file or directory in *dirpath*, do `os.path.join(dirpath, name)`. Whether or not the lists are sorted depends on the file system. If a file is removed from or added to the *dirpath* directory during generating the lists, whether a name for that file be included is unspecified.

선택적 인자 *topdown* 이 `True` 이거나 지정되지 않으면, 디렉터리에 대한 3-튜플은 하위 디렉터리에 대한 3-튜플이 생성되기 전에 생성됩니다 (디렉터리는 하향식으로 생성됩니다). *topdown* 이 `False` 면, 모든 하위 디렉터리에 대한 3-튜플 다음에 디렉터리에 대한 3-튜플이 생성됩니다 (디렉터리가 상향식으로 생성됨). *topdown* 의 값에 상관없이, 디렉터리와 해당 하위 디렉터리의 튜플이 생성되기 전에 하위 디렉터리 목록이 조회됩니다.

topdown 이 `True` 일 때, 호출자는 (아마도 `del` 또는 슬라이스 대입을 사용하여) *dirnames* 리스트를 수정할 수 있으며, *walk()* 는 이름이 *dirnames* 남아있는 하위 디렉터리로만 재귀합니다; 검색을 가지치기하거나, 특정 방문 순서를 지정하거나, 심지어 *walk()* 가 다시 시작하기 전에 호출자가 새로 만들거나 이름을 바꾼 디렉터리에 대해 *walk()* 에 알릴 때도 사용할 수 있습니다. *topdown* 이 `False` 일 때 *dirnames*를 수정하는 것은 walk의 동작에 영향을 주지 못하는데, 상향식 모드에서 *dirnames*의 디렉터리는 *dirpath* 자체가 생성되기 전에 생성되기 때문입니다.

기본적으로, *scandir()* 호출의 에러는 무시됩니다. 선택적 인자 *onerror* 가 지정되면, 함수여야 합니다; 하나의 인자 *OSError* 인스턴스로 호출됩니다. 에러를 보고하고 walk를 계속하도록 하거나, 예외를 발생시켜 walk를 중단할 수 있습니다. 파일명은 예외 객체의 `filename` 어트리뷰트로 제공됩니다.

기본적으로, *walk()* 는 디렉터리로 해석되는 심볼릭 링크로 이동하지 않습니다. 지원하는 시스템에서, 심볼릭 링크가 가리키는 디렉터리를 방문하려면, *followlinks*를 `True` 로 설정하십시오.

> **참고**
>
> 심볼릭 링크가 자신의 부모 디렉터리를 가리킬 때, *followlinks*를 `True` 로 설정하면 무한 재귀가 발생할 수 있음에 주의해야 합니다. *walk()* 는 이미 방문한 디렉터리를 추적하지 않습니다.

> **참고**
>
> 상대 경로명을 전달할 때는, *walk()* 가 실행되는 도중 현재 작업 디렉터리를 변경하지 마십시오. *walk()* 는 현재 디렉터리를 절대로 변경하지 않으며, 호출자도 마찬가지라고 가정합니다.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn' t look under any `__pycache__` subdirectory:

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/xml'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if '__pycache__' in dirs:
        dirs.remove('__pycache__')  # don't visit __pycache__ directories
```

다음 예(*shutil.rmtree()* 의 간단한 구현)에서는, 트리를 상향식으로 탐색하는 것이 필수적입니다, *rmdir()* 는 비어 있지 않은 디렉터리를 삭제할 수 없습니다:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
os.rmdir(top)
```

top, topdown, onerror, followlinks를 인자로 감사 이벤트(*auditing event*) os.walk를 발생시킵니다.

버전 3.5에서 변경: 이 함수는 이제 *os.listdir()* 대신 *os.scandir()*를 호출하기 때문에, *os.stat()* 호출 수를 줄여 더 빨라졌습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**fwalk**(*top='.'*, *topdown=True*, *onerror=None*, ***, *follow_symlinks=False*, *dir_fd=None*)

이 함수는 *walk()*와 똑같이 동작합니다. 단, 4-튜플 (dirpath, dirnames, filenames, dirfd)를 산출하고 dir_fd를 지원합니다.

dirpath , *dirnames* 및 *filenames* 은 *walk()* 출력과 같고, *dirfd* 는 *dirpath* 디렉터리를 가리키는 파일 기술자입니다.

이 함수는 항상 디렉터리 기술자에 상대적인 경로 및 심볼릭 링크를 따르지 않음을 지원합니다. 하지만, 다른 함수와는 달리, *follow_symlinks*에 대한 *fwalk()*의 기본값은 False임에 주의하십시오.

> **참고**
>
> *fwalk()*는 다음 이터레이션 단계까지만 유효한 파일 기술자를 산출하기 때문에, 더 오래 유지하려면 복제해야 합니다 (예를 들어, *dup()*로).

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn' t look under any __pycache__ subdirectory:

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/xml'):
    print(root, "consumes", end=" ")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name in files]),
          end=" ")
    print("bytes in", len(files), "non-directory files")
    if '__pycache__' in dirs:
        dirs.remove('__pycache__')  # don't visit __pycache__ directories
```

다음 예에서는, 트리를 상향식으로 탐색하는 것이 필수적입니다: *rmdir()*는 비어 있지 않은 디렉터리를 삭제할 수 없습니다:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files, rootfd in os.fwalk(top, topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

top, topdown, onerror, follow_symlinks, dir_fd를 인자로 감사 이벤트(*auditing event*) os.fwalk를 발생시킵니다.

가용성: Unix.

Added in version 3.3.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.7에서 변경: `bytes` 경로에 대한 지원이 추가되었습니다.

os.**memfd_create**(*name*[, *flags=os.MFD_CLOEXEC*])

익명 파일을 만들고 이를 가리키는 파일 기술자를 반환합니다. *flags*는 시스템에서 사용할 수 있는 `os.MFD_*` 상수(또는 이들의 비트별 OR 조합) 중 하나여야 합니다. 기본적으로, 새 파일 기술자는 상속 불가능합니다.

*name*에 제공된 이름은 파일명으로 사용되며 해당 심볼릭 링크의 대상으로 `/proc/self/fd/` 디렉터리에 표시됩니다. 표시된 이름에는 항상 `memfd:` 접두어가 붙으며 디버깅 목적으로만 사용됩니다. 이름은 파일 기술자의 동작에 영향을 미치지 않고, 여러 파일이 부작용 없이 같은 이름을 가질 수 있습니다.

가용성: Linux >= 3.17 with glibc >= 2.27.

Added in version 3.8.

os.**MFD_CLOEXEC**

os.**MFD_ALLOW_SEALING**

os.**MFD_HUGETLB**

os.**MFD_HUGE_SHIFT**

os.**MFD_HUGE_MASK**

os.**MFD_HUGE_64KB**

os.**MFD_HUGE_512KB**

os.**MFD_HUGE_1MB**

os.**MFD_HUGE_2MB**

os.**MFD_HUGE_8MB**

os.**MFD_HUGE_16MB**

os.**MFD_HUGE_32MB**

os.**MFD_HUGE_256MB**

os.**MFD_HUGE_512MB**

os.**MFD_HUGE_1GB**

os.**MFD_HUGE_2GB**

os.**MFD_HUGE_16GB**

이 플래그들은 `memfd_create()`로 전달될 수 있습니다.

가용성: Linux >= 3.17 with glibc >= 2.27

The `MFD_HUGE*` flags are only available since Linux 4.14.

Added in version 3.8.

os.**eventfd**(*initval*[, *flags=os.EFD_CLOEXEC*])

Create and return an event file descriptor. The file descriptors supports raw `read()` and `write()` with a buffer size of 8, `select()`, `poll()` and similar. See man page `eventfd(2)` for more information. By default, the new file descriptor is *non-inheritable*.

initval is the initial value of the event counter. The initial value must be a 32 bit unsigned integer. Please note that the initial value is limited to a 32 bit unsigned int although the event counter is an unsigned 64 bit integer with a maximum value of 2^{64}-2.

flags can be constructed from `EFD_CLOEXEC`, `EFD_NONBLOCK`, and `EFD_SEMAPHORE`.

If `EFD_SEMAPHORE` is specified and the event counter is non-zero, `eventfd_read()` returns 1 and decrements the counter by one.

If *EFD_SEMAPHORE* is not specified and the event counter is non-zero, *eventfd_read()* returns the current event counter value and resets the counter to zero.

If the event counter is zero and *EFD_NONBLOCK* is not specified, *eventfd_read()* blocks.

eventfd_write() increments the event counter. Write blocks if the write operation would increment the counter to a value larger than 2^{64}-2.

예:

```
import os

# semaphore with start value '1'
fd = os.eventfd(1, os.EFD_SEMAPHORE | os.EFC_CLOEXEC)
try:
    # acquire semaphore
    v = os.eventfd_read(fd)
    try:
        do_work()
    finally:
        # release semaphore
        os.eventfd_write(fd, v)
finally:
    os.close(fd)
```

가용성: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.10.

os.**eventfd_read**(*fd*)

Read value from an *eventfd()* file descriptor and return a 64 bit unsigned int. The function does not verify that *fd* is an *eventfd()*.

가용성: Linux >= 2.6.27

Added in version 3.10.

os.**eventfd_write**(*fd*, *value*)

Add value to an *eventfd()* file descriptor. *value* must be a 64 bit unsigned int. The function does not verify that *fd* is an *eventfd()*.

가용성: Linux >= 2.6.27

Added in version 3.10.

os.**EFD_CLOEXEC**

Set close-on-exec flag for new *eventfd()* file descriptor.

가용성: Linux >= 2.6.27

Added in version 3.10.

os.**EFD_NONBLOCK**

Set *O_NONBLOCK* status flag for new *eventfd()* file descriptor.

가용성: Linux >= 2.6.27

Added in version 3.10.

os.**EFD_SEMAPHORE**

Provide semaphore-like semantics for reads from an *eventfd()* file descriptor. On read the internal counter is decremented by one.

가용성: Linux >= 2.6.30

Added in version 3.10.

Timer File Descriptors

Added in version 3.13.

These functions provide support for Linux' s *timer file descriptor* API. Naturally, they are all only available on Linux.

os.**timerfd_create**(*clockid*, /, *, *flags=0*)

Create and return a timer file descriptor (*timerfd*).

The file descriptor returned by `timerfd_create()` supports:

- `read()`
- `select()`
- `poll()`

The file descriptor' s `read()` method can be called with a buffer size of 8. If the timer has already expired one or more times, `read()` returns the number of expirations with the host' s endianness, which may be converted to an `int` by `int.from_bytes(x, byteorder=sys.byteorder)`.

`select()` and `poll()` can be used to wait until timer expires and the file descriptor is readable.

clockid must be a valid *clock ID*, as defined in the `time` module:

- `time.CLOCK_REALTIME`
- `time.CLOCK_MONOTONIC`
- `time.CLOCK_BOOTTIME` (Since Linux 3.15 for timerfd_create)

If *clockid* is `time.CLOCK_REALTIME`, a settable system-wide real-time clock is used. If system clock is changed, timer setting need to be updated. To cancel timer when system clock is changed, see `TFD_TIMER_CANCEL_ON_SET`.

If *clockid* is `time.CLOCK_MONOTONIC`, a non-settable monotonically increasing clock is used. Even if the system clock is changed, the timer setting will not be affected.

If *clockid* is `time.CLOCK_BOOTTIME`, same as `time.CLOCK_MONOTONIC` except it includes any time that the system is suspended.

The file descriptor' s behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the `|` operator):

- `TFD_NONBLOCK`
- `TFD_CLOEXEC`

If `TFD_NONBLOCK` is not set as a flag, `read()` blocks until the timer expires. If it is set as a flag, `read()` doesn' t block, but If there hasn' t been an expiration since the last call to read, `read()` raises `OSError` with `errno` is set to `errno.EAGAIN`.

`TFD_CLOEXEC` is always set by Python automatically.

The file descriptor must be closed with `os.close()` when it is no longer needed, or else the file descriptor will be leaked.

> **더 보기**
>
> The `timerfd_create(2)` man page.

가용성: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**timerfd_settime**(*fd*, /, *, *flags=flags*, *initial=0.0*, *interval=0.0*)

Alter a timer file descriptor' s internal timer. This function operates the same interval timer as `timerfd_settime_ns()`.

fd must be a valid timer file descriptor.

The timer' s behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the `|` operator):

- `TFD_TIMER_ABSTIME`
- `TFD_TIMER_CANCEL_ON_SET`

The timer is disabled by setting *initial* to zero (`0`). If *initial* is equal to or greater than zero, the timer is enabled. If *initial* is less than zero, it raises an `OSError` exception with `errno` set to `errno.EINVAL`

By default the timer will fire when *initial* seconds have elapsed. (If *initial* is zero, timer will fire immediately.)

However, if the `TFD_TIMER_ABSTIME` flag is set, the timer will fire when the timer' s clock (set by *clockid* in `timerfd_create()`) reaches *initial* seconds.

The timer' s interval is set by the *interval* `float`. If *interval* is zero, the timer only fires once, on the initial expiration. If *interval* is greater than zero, the timer fires every time *interval* seconds have elapsed since the previous expiration. If *interval* is less than zero, it raises `OSError` with `errno` set to `errno.EINVAL`

If the `TFD_TIMER_CANCEL_ON_SET` flag is set along with `TFD_TIMER_ABSTIME` and the clock for this timer is `time.CLOCK_REALTIME`, the timer is marked as cancelable if the real-time clock is changed discontinuously. Reading the descriptor is aborted with the error ECANCELED.

Linux manages system clock as UTC. A daylight-savings time transition is done by changing time offset only and doesn' t cause discontinuous system clock change.

Discontinuous system clock change will be caused by the following events:

- `settimeofday`
- `clock_settime`
- set the system date and time by `date` command

Return a two-item tuple of (`next_expiration`, `interval`) from the previous timer state, before this function executed.

> **더 보기**
>
> `timerfd_create(2)`, `timerfd_settime(2)`, `settimeofday(2)`, `clock_settime(2)`, and `date(1)`.

가용성: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**timerfd_settime_ns**(*fd*, /, *, *flags=0*, *initial=0*, *interval=0*)

Similar to `timerfd_settime()`, but use time as nanoseconds. This function operates the same interval timer as `timerfd_settime()`.

가용성: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**timerfd_gettime**(*fd*, /)

Return a two-item tuple of floats (`next_expiration`, `interval`).

`next_expiration` denotes the relative time until next the timer next fires, regardless of if the `TFD_TIMER_ABSTIME` flag is set.

`interval` denotes the timer' s interval. If zero, the timer will only fire once, after `next_expiration` seconds have elapsed.

> **더 보기**
>
> `timerfd_gettime(2)`

가용성: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**timerfd_gettime_ns**(*fd*, /)

Similar to `timerfd_gettime()`, but return time as nanoseconds.

가용성: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_NONBLOCK**

A flag for the `timerfd_create()` function, which sets the `O_NONBLOCK` status flag for the new timer file descriptor. If `TFD_NONBLOCK` is not set as a flag, `read()` blocks.

가용성: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_CLOEXEC**

A flag for the `timerfd_create()` function, If `TFD_CLOEXEC` is set as a flag, set close-on-exec flag for new file descriptor.

가용성: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_TIMER_ABSTIME**

A flag for the `timerfd_settime()` and `timerfd_settime_ns()` functions. If this flag is set, *initial* is interpreted as an absolute value on the timer' s clock (in UTC seconds or nanoseconds since the Unix Epoch).

가용성: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_TIMER_CANCEL_ON_SET**

A flag for the `timerfd_settime()` and `timerfd_settime_ns()` functions along with `TFD_TIMER_ABSTIME`. The timer is cancelled when the time of the underlying clock changes discontinuously.

가용성: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

리눅스 확장 어트리뷰트

Added in version 3.3.

이 함수들은 모두 리눅스에서만 사용 가능합니다.

os.**getxattr**(*path*, *attribute*, **, *follow_symlinks=True*)

*path*의 확장 파일 시스템 어트리뷰트 *attribute*의 값을 반환합니다. *attribute* 는 bytes 또는 str(직접 또는 `PathLike` 인터페이스를 통해 간접적으로)일 수 있습니다. str이면, 파일 시스템 인코딩으로 인코딩됩니다.

이 함수는 파일 기술자 지정 및 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

`path`, `attribute`를 인자로 감사 이벤트(*auditing event*) `os.getxattr`을 발생시킵니다.

버전 3.6에서 변경: *path* 및 *attribute*에 대해 경로류 객체를 받아들입니다.

os.**listxattr**(*path=None, *, follow_symlinks=True*)

path 의 확장 파일 시스템 어트리뷰트 목록을 반환합니다. 목록의 어트리뷰트는 파일 시스템 인코딩으로 디코딩된 문자열로 표시됩니다. *path* 가 None 이면, *listxattr()* 는 현재 디렉터리를 검사합니다.

이 함수는 파일 기술자 지정 및 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

path를 인자로 감사 이벤트(*auditing event*) os.listxattr을 발생시킵니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**removexattr**(*path, attribute, *, follow_symlinks=True*)

Removes the extended filesystem attribute *attribute* from *path*. *attribute* should be bytes or str (directly or indirectly through the *PathLike* interface). If it is a string, it is encoded with the *filesystem encoding and error handler*.

이 함수는 파일 기술자 지정 및 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

path, attribute를 인자로 감사 이벤트(*auditing event*) os.removexattr을 발생시킵니다.

버전 3.6에서 변경: *path* 및 *attribute*에 대해 경로류 객체를 받아들입니다.

os.**setxattr**(*path, attribute, value, flags=0, *, follow_symlinks=True*)

Set the extended filesystem attribute *attribute* on *path* to *value*. *attribute* must be a bytes or str with no embedded NULs (directly or indirectly through the *PathLike* interface). If it is a str, it is encoded with the *filesystem encoding and error handler*. *flags* may be *XATTR_REPLACE* or *XATTR_CREATE*. If *XATTR_REPLACE* is given and the attribute does not exist, ENODATA will be raised. If *XATTR_CREATE* is given and the attribute already exists, the attribute will not be created and EEXISTS will be raised.

이 함수는 파일 기술자 지정 및 심볼릭 링크를 따르지 않음을 지원할 수 있습니다.

> **참고**
>
> 리눅스 커널 버전 2.6.39 미만의 버그로 인해 flags 인자가 일부 파일 시스템에서 무시되었습니다.

path, attribute, value, flags를 인자로 감사 이벤트(*auditing event*) os.setxattr을 발생시킵니다.

버전 3.6에서 변경: *path* 및 *attribute*에 대해 경로류 객체를 받아들입니다.

os.**XATTR_SIZE_MAX**

확장 어트리뷰트 값의 최대 크기입니다. 현재, 리눅스에서 64 KiB입니다.

os.**XATTR_CREATE**

이것은 *setxattr()* 의 flags 인자를 위한 값입니다. 연산이 반드시 어트리뷰트를 새로 만들어야 함을 나타냅니다.

os.**XATTR_REPLACE**

이것은 *setxattr()* 의 flags 인자를 위한 값입니다. 연산이 반드시 기존 어트리뷰트를 대체해야 함을 나타냅니다.

16.1.7 프로세스 관리

이 함수들은 프로세스를 만들고 관리하는데 사용될 수 있습니다.

다양한 *exec** 함수는 프로세스로 로드되는 새 프로그램에 대한 인자 목록을 받아들입니다. 각각의 경우에, 첫 번째 인자는 사용자가 명령 줄에 입력할 수 있는 인자가 아닌 프로그램 자체의 이름으로 새 프로그램에 전달됩니다. C 프로그래머에게, 이것은 프로그램의 main() 에 전달된 argv[0] 입니다. 예를 들어, os.execv('/bin/echo', ['foo', 'bar']) 는 표준 출력에 bar만 인쇄합니다; foo는 무시되는 것처럼 보이게 됩니다.

os.**abort**()

현재 프로세스에 SIGABRT 시그널을 생성합니다. 유닉스에서, 기본 동작은 코어 덤프를 생성하는

것입니다; 윈도우에서, 프로세스는 즉시 종료 코드 3을 반환합니다. 이 함수를 호출하면 *signal.signal()*를 사용하여 SIGABRT에 등록된 파이썬 시그널 처리기를 호출하지 않게 됨에 주의하시기 바랍니다.

os.**add_dll_directory**(*path*)

DLL 검색 경로에 path를 추가합니다.

This search path is used when resolving dependencies for imported extension modules (the module itself is resolved through *sys.path*), and also by *ctypes*.

반환된 객체의 **close()**를 호출하거나 반환된 객체를 with 문에서 사용하여 디렉터리를 제거하십시오.

DLL이 로드되는 방법에 대한 자세한 내용은 마이크로소프트 설명서를 참조하십시오.

path를 인자로 감사 이벤트(*auditing event*) os.add_dll_directory를 발생시킵니다.

가용성: Windows.

Added in version 3.8: 이전 버전의 CPython은 현재 프로세스의 기본 동작을 사용하여 DLL을 해결(resolve)합니다. 이로 인해 때때로 PATH나 현재 작업 디렉터리를 검색하거나, AddDllDirectory와 같은 OS 함수가 효과가 없게 되는 것과 같은 일관성 없는 결과를 낳습니다.

3.8에서는, 일관성을 보장하기 위해 이제 DLL이 로드되는 두 가지 기본 방법이 프로세스 전반의 동작을 명시적으로 재정의합니다. 라이브러리 갱신에 대한 정보는 이식 주의 사항을 참조하십시오.

os.**execl**(*path, arg0, arg1, ...*)

os.**execle**(*path, arg0, arg1, ..., env*)

os.**execlp**(*file, arg0, arg1, ...*)

os.**execlpe**(*file, arg0, arg1, ..., env*)

os.**execv**(*path, args*)

os.**execve**(*path, args, env*)

os.**execvp**(*file, args*)

os.**execvpe**(*file, args, env*)

이 함수들은 모두 현재 프로세스를 대체해서 새로운 프로그램을 실행합니다; 반환되지 않습니다. 유닉스에서, 새로운 실행 파일이 현재 프로세스에 로드되고, 호출자와 같은 프로세스 ID를 갖게 됩니다. 에러는 *OSError* 예외로 보고됩니다.

현재 프로세스가 즉시 교체됩니다. 열린 파일 객체와 기술자는 플러시 되지 않으므로, 이러한 열린 파일에 버퍼링 된 데이터가 있으면, *exec** 함수를 호출하기 전에 sys.stdout.flush() 또는 *os.fsync()*를 사용하여 플러시 해야 합니다.

The "l" and "v" variants of the *exec** functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the execl*() functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process should start with the name of the command being run, but this is not enforced.

The variants which include a "p" near the end (*execlp()*, *execlpe()*, *execvp()*, and *execvpe()*) will use the PATH environment variable to locate the program *file*. When the environment is being replaced (using one of the *exec*e* variants, discussed in the next paragraph), the new environment is used as the source of the PATH variable. The other variants, *execl()*, *execle()*, *execv()*, and *execve()*, will not use the PATH variable to locate the executable; *path* must contain an appropriate absolute or relative path. Relative paths must include at least one slash, even on Windows, as plain names will not be resolved.

execle(), *execlpe()*, *execve()*, *execvpe()*의 경우 (모두 "e"로 끝납니다), *env* 매개 변수는 새 프로세스의 환경 변수를 정의하는 데 사용되는 매핑이어야 합니다 (이것이 현재 프로세스의 환경 대신 사용됩니다); 함수 *execl()*, *execlp()*, *execv()* 및 *execvp()*는 모두 새 프로세스가 현재 프로세스의 환경을 상속하게 합니다.

일부 플랫폼에서 *execve()*의 경우, *path* 는 열린 파일 기술자로도 지정될 수 있습니다. 이 기능은 여러분의 플랫폼에서 지원되지 않을 수 있습니다; *os.supports_fd*를 사용하여 사용할 수 있는지를 확인할 수 있습니다. 사용할 수 없을 때, 이를 사용하면 *NotImplementedError*가 발생합니다.

path, args, env를 인자로 감사 이벤트(*auditing event*) os.exec를 발생시킵니다.

가용성: Unix, Windows, not WASI, not Android, not iOS.

버전 3.3에서 변경: *execve()*의 *path*에 열린 파일 기술자를 지정하는 지원이 추가되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

os.**_exit**(*n*)

상태 *n*으로 프로세스를 종료합니다. 클린업 처리기를 호출하거나, stdio 버퍼를 플러시 하거나 등등은 수행하지 않습니다.

> **참고**
>
> The standard way to exit is *sys.exit(n)*. _exit() should normally only be used in the child process after a *fork()*.

필수 조건은 아니지만, 다음 종료 코드가 정의되어 있으며 *_exit()*와 함께 사용할 수 있습니다. 이것은 메일 서버의 외부 명령 배달 프로그램과 같이 파이썬으로 작성된 시스템 프로그램에서 일반적으로 사용됩니다.

> **참고**
>
> 약간의 차이점이 있어서, 이들 중 일부는 모든 유닉스 플랫폼에서 사용하지는 못할 수 있습니다. 이 상수는 하부 플랫폼에서 정의될 때만 정의됩니다.

os.**EX_OK**

Exit code that means no error occurred. May be taken from the defined value of EXIT_SUCCESS on some platforms. Generally has a value of zero.

가용성: Unix, Windows.

os.**EX_USAGE**

잘못된 개수의 인자가 제공된 경우처럼, 명령이 잘못 사용되었음을 나타내는 종료 코드.

가용성: Unix, not WASI.

os.**EX_DATAERR**

입력 데이터가 잘못되었음을 나타내는 종료 코드.

가용성: Unix, not WASI.

os.**EX_NOINPUT**

입력 파일이 없거나 읽을 수 없음을 나타내는 종료 코드.

가용성: Unix, not WASI.

os.**EX_NOUSER**

지정된 사용자가 존재하지 않음을 나타내는 종료 코드.

가용성: Unix, not WASI.

os.**EX_NOHOST**

지정된 호스트가 존재하지 않음을 나타내는 종료 코드.

가용성: Unix, not WASI.

os.**EX_UNAVAILABLE**

필수 서비스를 사용할 수 없음을 나타내는 종료 코드.

가용성: Unix, not WASI.

os.**EX_SOFTWARE**

내부 소프트웨어 에러가 감지되었음을 나타내는 종료 코드.

가용성: Unix, not WASI.

os.**EX_OSERR**

포크 하거나 파이프를 만들 수 없는 등, 운영 체제 에러가 감지되었음을 나타내는 종료 코드.

가용성: Unix, not WASI.

os.**EX_OSFILE**

일부 시스템 파일이 없거나, 열 수 없거나, 다른 에러가 있음을 나타내는 종료 코드.

가용성: Unix, not WASI.

os.**EX_CANTCREAT**

사용자가 지정한 출력 파일을 만들 수 없음을 나타내는 종료 코드.

가용성: Unix, not WASI.

os.**EX_IOERR**

일부 파일에서 I/O를 수행하는 동안 에러가 발생했음을 나타내는 종료 코드.

가용성: Unix, not WASI.

os.**EX_TEMPFAIL**

임시 에러가 발생했음을 나타내는 종료 코드. 이는 재시도 가능한 작업 중에 만들 수 없었던 네트워크 연결과 같이 실제로는 에러가 아닐 수 있는 것을 나타냅니다.

가용성: Unix, not WASI.

os.**EX_PROTOCOL**

프로토콜 교환이 불법이거나 유효하지 않거나 이해되지 않았음을 나타내는 종료 코드.

가용성: Unix, not WASI.

os.**EX_NOPERM**

작업을 수행할 수 있는 권한이 충분하지 않음을 나타내는 종료 코드 (파일 시스템 문제에는 사용하지 않습니다).

가용성: Unix, not WASI.

os.**EX_CONFIG**

어떤 종류의 구성 에러가 발생했음을 나타내는 종료 코드.

가용성: Unix, not WASI.

os.**EX_NOTFOUND**

"항목을 찾을 수 없습니다" 와 같은 것을 의미하는 종료 코드.

가용성: Unix, not WASI.

os.**fork**()

자식 프로세스를 포크 합니다. 자식에서는 0을 반환하고, 부모에서는 자식의 프로세스 ID를 반환합니다. 에러가 발생하면 *OSError*를 일으킵니다.

FreeBSD <= 6.3 및 Cygwin을 포함한 일부 플랫폼은 스레드에서 `fork()` 를 사용할 때 알려진 문제점이 있습니다.

인자 없이 감사 이벤트(*auditing event*) `os.fork`를 발생시킵니다.

> **경고**
>
> If you use TLS sockets in an application calling `fork()`, see the warning in the *ssl* documentation.

> **경고**
>
> On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using `urllib.request`.

버전 3.8에서 변경: 서브 인터프리터에서 `fork()`를 호출하는 것은 더는 지원되지 않습니다 (`RuntimeError`가 발생합니다).

버전 3.12에서 변경: If Python is able to detect that your process has multiple threads, `os.fork()` now raises a `DeprecationWarning`.

We chose to surface this as a warning, when detectable, to better inform developers of a design problem that the POSIX platform specifically notes as not supported. Even in code that *appears* to work, it has never been safe to mix threading with `os.fork()` on POSIX platforms. The CPython runtime itself has always made API calls that are not safe for use in the child process when threads existed in the parent (such as `malloc` and `free`).

Users of macOS or users of libc or malloc implementations other than those typically found in glibc to date are among those already more likely to experience deadlocks running such code.

See this discussion on fork being incompatible with threads for technical details of why we' re surfacing this longstanding platform compatibility problem to developers.

가용성: POSIX, not WASI, not Android, not iOS.

`os.`**`forkpty`**`()`

새 의사 터미널을 자식의 제어 터미널로 사용하여 자식 프로세스를 포크 합니다. `(pid, fd)` 쌍을 반환하는데, 여기서 *pid* 는 자식에서 `0`이고, 부모에서는 새 자식의 프로세스 ID이고, *fd* 는 의사 터미널의 마스터 단의 파일 기술자입니다. 좀 더 이식성 있는 접근법을 사용하려면, `pty` 모듈을 사용하십시오. 에러가 발생하면 `OSError`를 일으킵니다.

인자 없이 감사 이벤트(*auditing event*) `os.forkpty`를 발생시킵니다.

> **경고**
>
> On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using `urllib.request`.

버전 3.8에서 변경: 서브 인터프리터에서 `forkpty()`를 호출하는 것은 더는 지원되지 않습니다 (`RuntimeError`가 발생합니다).

버전 3.12에서 변경: If Python is able to detect that your process has multiple threads, this now raises a `DeprecationWarning`. See the longer explanation on `os.fork()`.

가용성: Unix, not WASI, not Android, not iOS.

`os.`**`kill`**`(`*pid, sig, /*`)`

프로세스 *pid*에 시그널 *sig*를 보냅니다. 호스트 플랫폼에서 사용할 수 있는 구체적인 시그널에 대한 상수는 `signal` 모듈에 정의되어 있습니다.

Windows: The `signal.CTRL_C_EVENT` and `signal.CTRL_BREAK_EVENT` signals are special signals which can only be sent to console processes which share a common console window, e.g., some subprocesses. Any other value for *sig* will cause the process to be unconditionally killed by the TerminateProcess API, and the exit code will be set to *sig*.

`signal.pthread_kill()`도 참조하십시오.

`pid`, `sig`를 인자로 감사 이벤트(*auditing event*) `os.kill`을 발생시킵니다.

가용성: Unix, Windows, not WASI, not iOS.

버전 3.2에서 변경: 윈도우 지원이 추가되었습니다.

os.**killpg**(*pgid, sig, /*)

시그널 *sig* 를 프로세스 그룹 *pgid* 로 보냅니다.

pgid, sig를 인자로 감사 이벤트(*auditing event*) os.killpg를 발생시킵니다.

가용성: Unix, not WASI, not iOS.

os.**nice**(*increment, /*)

프로세스의 "우선도(niceness)" 에 *increment* 를 추가합니다. 새로운 우선도를 반환합니다.

가용성: Unix, not WASI.

os.**pidfd_open**(*pid, flags=0*)

Return a file descriptor referring to the process *pid* with *flags* set. This descriptor can be used to perform process management without races and signals.

자세한 내용은 *pidfd_open(2)* 매뉴얼 페이지를 참조하십시오.

가용성: Linux >= 5.3, Android >= *build-time* API level 31

Added in version 3.9.

os.**PIDFD_NONBLOCK**

This flag indicates that the file descriptor will be non-blocking. If the process referred to by the file descriptor has not yet terminated, then an attempt to wait on the file descriptor using *waitid(2)* will immediately return the error *EAGAIN* rather than blocking.

가용성: Linux >= 5.10

Added in version 3.12.

os.**plock**(*op, /*)

프로그램 세그먼트를 메모리에 잠급니다. (<sys/lock.h>에서 정의된) *op* 값은 잠기는 세그먼트를 판별합니다.

가용성: Unix, not WASI, not iOS.

os.**popen**(*cmd, mode='r', buffering=-1*)

Open a pipe to or from command *cmd*. The return value is an open file object connected to the pipe, which can be read or written depending on whether *mode* is 'r' (default) or 'w'. The *buffering* argument have the same meaning as the corresponding argument to the built-in *open()* function. The returned file object reads or writes text strings rather than bytes.

close 메서드는 자식 프로세스가 성공적으로 종료되면 *None*을 반환하고, 에러가 있으면 자식 프로세스가 반환한 코드를 반환합니다. POSIX 시스템에서, 반환 코드가 양수면, 프로세스의 반환 값을 1바이트 왼쪽으로 시프트 한 값을 나타냅니다. 반환 코드가 음수면, 음의 반환 코드로 주어진 시그널에 의해 강제 종료된 것입니다. 예를 들어, 자식 프로세스가 죽었을(kill) 때 반환 값은 - signal.SIGKILL 일 수 있습니다. 윈도우 시스템에서, 반환 값은 자식 프로세스의 부호 있는 정수 반환 코드를 포함합니다.

유닉스에서, *waitstatus_to_exitcode()*는 None이 아닐 때 close 메서드 결과(종료 상태)를 종료 코드로 변환하는 데 사용할 수 있습니다. 윈도우에서, close 메서드 결과는 직접 종료 코드(또는 None)입니다.

이것은 *subprocess.Popen*를 사용하여 구현됩니다; 자식 프로세스를 관리하고 통신하는 보다 강력한 방법에 대해서는 이 클래스의 설명서를 참조하십시오.

가용성: not WASI, not Android, not iOS.

> **참고**
>
> The *Python UTF-8 Mode* affects encodings used for *cmd* and pipe contents.
>
> *popen()* is a simple wrapper around *subprocess.Popen*. Use *subprocess.Popen* or *subprocess.run()* to control options like encodings.

버전 3.14부터 폐지됨: The function is *soft deprecated* and should no longer be used to write new code. The `subprocess` module is recommended instead.

os.**posix_spawn**(*path*, *argv*, *env*, **, file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

Wraps the `posix_spawn()` C library API for use from Python.

대부분 사용자는 `posix_spawn()` 대신 `subprocess.run()`을 사용해야 합니다.

The positional-only arguments *path*, *args*, and *env* are similar to `execve()`. *env* is allowed to be `None`, in which case current process' environment is used.

path 매개 변수는 실행 파일의 경로입니다. *path*에는 디렉터리가 있어야 합니다. 디렉터리 없이 실행 파일을 전달하려면 `posix_spawnp()`를 사용하십시오.

file_actions 인자는 C 라이브러리 구현의 `fork()`와 `exec()` 단계 사이의 자식 프로세스에서 특정 파일 기술자에 취할 동작을 설명하는 튜플의 시퀀스 일 수 있습니다. 각 튜플의 첫 번째 항목은 나머지 튜플 요소를 설명하는, 아래에 나열된 세 가지 형 지시자 중 하나여야 합니다:

os.**POSIX_SPAWN_OPEN**

(`os.POSIX_SPAWN_OPEN`, *fd*, *path*, *flags*, *mode*)

`os.dup2(os.open(path, flags, mode), fd)`를 수행합니다.

os.**POSIX_SPAWN_CLOSE**

(`os.POSIX_SPAWN_CLOSE`, *fd*)

`os.close(fd)`를 수행합니다.

os.**POSIX_SPAWN_DUP2**

(`os.POSIX_SPAWN_DUP2`, *fd*, *new_fd*)

`os.dup2(fd, new_fd)`를 수행합니다.

os.**POSIX_SPAWN_CLOSEFROM**

(`os.POSIX_SPAWN_CLOSEFROM`, *fd*)

Performs `os.closerange(fd, INF)`.

These tuples correspond to the C library `posix_spawn_file_actions_addopen()`, `posix_spawn_file_actions_addclose()`, `posix_spawn_file_actions_adddup2()`, and `posix_spawn_file_actions_addclosefrom_np()` API calls used to prepare for the `posix_spawn()` call itself.

The *setpgroup* argument will set the process group of the child to the value specified. If the value specified is 0, the child' s process group ID will be made the same as its process ID. If the value of *setpgroup* is not set, the child will inherit the parent' s process group ID. This argument corresponds to the C library `POSIX_SPAWN_SETPGROUP` flag.

If the *resetids* argument is `True` it will reset the effective UID and GID of the child to the real UID and GID of the parent process. If the argument is `False`, then the child retains the effective UID and GID of the parent. In either case, if the set-user-ID and set-group-ID permission bits are enabled on the executable file, their effect will override the setting of the effective UID and GID. This argument corresponds to the C library `POSIX_SPAWN_RESETIDS` flag.

If the *setsid* argument is `True`, it will create a new session ID for `posix_spawn`. *setsid* requires `POSIX_SPAWN_SETSID` or `POSIX_SPAWN_SETSID_NP` flag. Otherwise, `NotImplementedError` is raised.

The *setsigmask* argument will set the signal mask to the signal set specified. If the parameter is not used, then the child inherits the parent' s signal mask. This argument corresponds to the C library `POSIX_SPAWN_SETSIGMASK` flag.

The *sigdef* argument will reset the disposition of all signals in the set specified. This argument corresponds to the C library `POSIX_SPAWN_SETSIGDEF` flag.

The *scheduler* argument must be a tuple containing the (optional) scheduler policy and an instance of `sched_param` with the scheduler parameters. A value of `None` in the place of the scheduler policy indicates

that is not being provided. This argument is a combination of the C library `POSIX_SPAWN_SETSCHEDPARAM` and `POSIX_SPAWN_SETSCHEDULER` flags.

`path`, `argv`, `env`를 인자로 감사 이벤트(*auditing event*) `os.posix_spawn`을 발생시킵니다.

Added in version 3.8.

버전 3.13에서 변경: *env* parameter accepts `None`. `os.POSIX_SPAWN_CLOSEFROM` is available on platforms where `posix_spawn_file_actions_addclosefrom_np()` exists.

가용성: Unix, not WASI, not Android, not iOS.

`os.`**`posix_spawnp`**(*path, argv, env, *, file_actions=None, setpgroup=None, resetids=False, setsid=False, setsigmask=(), setsigdef=(), scheduler=None*)

Wraps the `posix_spawnp()` C library API for use from Python.

시스템이 (`execvp(3)`과 같은 방식으로) `PATH` 환경 변수에 의해 지정된 디렉터리 목록에서 실행 파일을 검색한다는 점을 제외하고는 *`posix_spawn()`*과 유사합니다.

`path`, `argv`, `env`를 인자로 감사 이벤트(*auditing event*) `os.posix_spawn`을 발생시킵니다.

Added in version 3.8.

가용성: POSIX, not WASI, not Android, not iOS.

See *`posix_spawn()`* documentation.

`os.`**`register_at_fork`**(**, before=None, after_in_parent=None, after_in_child=None*)

`os.fork()` 또는 유사한 프로세스 복제 API를 사용하여 새 자식 프로세스가 포크 될 때 실행될 콜러블들을 등록합니다. 매개 변수는 선택적이며 키워드 전용입니다. 각각은 다른 호출 지점을 지정합니다.

- *before* 는 자식 프로세스를 포크 하기 전에 호출되는 함수입니다.
- *after_in_parent* 는 자식 프로세스를 포크 한 후에 부모 프로세스에서 호출되는 함수입니다.
- *after_in_child* 는 자식 프로세스에서 호출되는 함수입니다.

이러한 호출은 제거가 파이썬 인터프리터로 반환될 것으로 예상되는 경우에만 수행됩니다. 일반적인 *`subprocess`* 실행은 자식이 인터프리터로 재진입하지 않기 때문에, 이 호출들이 일어나지 않습니다.

포크 이전에 실행되도록 등록된 함수는 등록 역순으로 실행됩니다. 포크 후에 실행되도록 등록된 함수(부모나 자식 모두)는 등록 순서로 호출됩니다.

제삼자 C 코드에 의한 `fork()` 호출은, 그것이 명시적으로 `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` 및 `PyOS_AfterFork_Child()`를 호출하지 않는 한, 이 함수들을 호출하지 않습니다.

함수 등록을 취소할 방법은 없습니다.

가용성: Unix, not WASI, not Android, not iOS.

Added in version 3.7.

`os.`**`spawnl`**(*mode, path, ...*)

`os.`**`spawnle`**(*mode, path, ..., env*)

`os.`**`spawnlp`**(*mode, file, ...*)

`os.`**`spawnlpe`**(*mode, file, ..., env*)

`os.`**`spawnv`**(*mode, path, args*)

`os.`**`spawnve`**(*mode, path, args, env*)

`os.`**`spawnvp`**(*mode, file, args*)

`os.`**`spawnvpe`**(*mode, file, args, env*)

새 프로세스에서 프로그램 *path* 를 실행합니다.

(*`subprocess`* 모듈은 새 프로세스를 생성하고 결과를 조회하는데, 더욱 강력한 기능을 제공합니다; 이 모듈을 사용하는 것이 이 함수들을 사용하는 것보다 더 바람직합니다. 특히 이전 함수를 *subprocess* 모듈로 교체하기 섹션을 확인하십시오.)

mode 가 `P_NOWAIT`면, 이 함수는 새 프로세스의 프로세스 ID를 반환합니다; *mode*가 `P_WAIT`면, 종료 코드(정상적으로 종료했을 때)나 `-signal`(*signal*은 프로세스를 죽인 시그널입니다)을 반환합니다. 윈도우에서, 프로세스 ID는 실제로 프로세스 핸들이므로, `waitpid()` 함수에 사용할 수 있습니다.

VxWorks에서, 이 함수는 새로운 프로세스가 죽을(kill) 때 `-signal`을 반환하지 않습니다. 대신 OSError 예외가 발생합니다.

The "l" and "v" variants of the `spawn*` functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the `spawnl*()` functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process must start with the name of the command being run.

끝 근처에 두 번째 "p"가 포함된 변형(`spawnlp()`, `spawnlpe()`, `spawnvp()` 및 `spawnvpe()`)은 `PATH` 환경 변수를 사용하여 프로그램 *file*을 찾습니다. 환경이 대체 될 때 (다음 단락에서 설명할 `spawn*e` 변형 중 하나를 사용하여), 새 환경이 `PATH` 변수의 소스로 사용됩니다. 다른 변형 `spawnl()`, `spawnle()`, `spawnv()` 및 `spawnve()`는 `PATH` 변수를 사용하여 실행 파일을 찾지 않습니다; *path* 에는 반드시 적절한 절대 또는 상대 경로가 있어야 합니다.

`spawnle()`, `spawnlpe()`, `spawnve()` 및 `spawnvpe()`의 경우 (모두 "e"로 끝납니다), *env* 매개 변수는 새 프로세스의 환경 변수를 정의하는 데 사용되는 매핑이어야 합니다 (이것이 현재 프로세스의 환경 대신 사용됩니다); 함수 `spawnl()`, `spawnlp()`, `spawnv()` 및 `spawnvp()`는 모두 새 프로세스가 현재 프로세스의 환경을 상속하게 합니다. *env* 딕셔너리의 키와 값은 반드시 문자열이어야 함에 주의하십시오; 잘못된 키나 값은 반환 값 `127`로 함수가 실패하게 합니다.

예를 들어, `spawnlp()` 및 `spawnvpe()`에 대한 다음 호출은 동등합니다:

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

`mode`, `path`, `args`, `env`를 인자로 감사 이벤트(*auditing event*) `os.spawn`을 발생시킵니다.

가용성: Unix, Windows, not WASI, not Android, not iOS.

`spawnlp()`, `spawnlpe()`, `spawnvp()` and `spawnvpe()` are not available on Windows. `spawnle()` and `spawnve()` are not thread-safe on Windows; we advise you to use the `subprocess` module instead.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.14부터 폐지됨: These functions are *soft deprecated* and should no longer be used to write new code. The `subprocess` module is recommended instead.

`os.`**`P_NOWAIT`**

`os.`**`P_NOWAITO`**

Possible values for the *mode* parameter to the `spawn*` family of functions. If either of these values is given, the `spawn*` functions will return as soon as the new process has been created, with the process id as the return value.

가용성: Unix, Windows.

`os.`**`P_WAIT`**

Possible value for the *mode* parameter to the `spawn*` family of functions. If this is given as *mode*, the `spawn*` functions will not return until the new process has run to completion and will return the exit code of the process the run is successful, or `-signal` if a signal kills the process.

가용성: Unix, Windows.

`os.`**`P_DETACH`**

os.**P_OVERLAY**

*spawn** 계열 함수의 *mode* 매개 변수에 사용할 수 있는 값. 이들은 위에 나열된 것보다 이식성이 낮습니다. *P_DETACH*는 *P_NOWAIT*와 비슷하지만, 새 프로세스는 호출 프로세스의 콘솔에서 분리됩니다. *P_OVERLAY*가 사용되면, 현재 프로세스가 대체됩니다; *spawn** 함수가 반환되지 않습니다.

가용성: Windows.

os.**startfile**(*path*[, *operation*][, *arguments*][, *cwd*][, *show_cmd*])

연관된 응용 프로그램으로 파일을 시작합니다.

When *operation* is not specified, this acts like double-clicking the file in Windows Explorer, or giving the file name as an argument to the **start** command from the interactive command shell: the file is opened with whatever application (if any) its extension is associated.

When another *operation* is given, it must be a "command verb" that specifies what should be done with the file. Common verbs documented by Microsoft are `'open'`, `'print'` and `'edit'` (to be used on files) as well as `'explore'` and `'find'` (to be used on directories).

When launching an application, specify *arguments* to be passed as a single string. This argument may have no effect when using this function to launch a document.

The default working directory is inherited, but may be overridden by the *cwd* argument. This should be an absolute path. A relative *path* will be resolved against this argument.

Use *show_cmd* to override the default window style. Whether this has any effect will depend on the application being launched. Values are integers as supported by the Win32 `ShellExecute()` function.

startfile() returns as soon as the associated application is launched. There is no option to wait for the application to close, and no way to retrieve the application' s exit status. The *path* parameter is relative to the current directory or *cwd*. If you want to use an absolute path, make sure the first character is not a slash (`'/'`) Use *pathlib* or the *os.path.normpath()* function to ensure that paths are properly encoded for Win32.

To reduce interpreter startup overhead, the Win32 `ShellExecute()` function is not resolved until this function is first called. If the function cannot be resolved, *NotImplementedError* will be raised.

`path`, `operation`을 인자로 감사 이벤트(*auditing event*) `os.startfile`을 발생시킵니다.

Raises an *auditing event* `os.startfile/2` with arguments `path`, `operation`, `arguments`, `cwd`, `show_cmd`.

가용성: Windows.

버전 3.10에서 변경: Added the *arguments*, *cwd* and *show_cmd* arguments, and the `os.startfile/2` audit event.

os.**system**(*command*)

Execute the command (a string) in a subshell. This is implemented by calling the Standard C function `system()`, and has the same limitations. Changes to *sys.stdin*, etc. are not reflected in the environment of the executed command. If *command* generates any output, it will be sent to the interpreter standard output stream. The C standard does not specify the meaning of the return value of the C function, so the return value of the Python function is system-dependent.

On Unix, the return value is the exit status of the process encoded in the format specified for *wait()*.

윈도우에서, 반환 값은 *command*를 실행한 후 시스템 셸에서 반환한 값입니다. 셸은 윈도우 환경 변수 `COMSPEC`에 의해 제공됩니다: 보통 **cmd.exe**인데, 명령 실행의 종료 상태를 반환합니다; 기본이 아닌 셸을 사용하는 시스템에서는 셸 설명서를 참조하십시오.

The *subprocess* module provides more powerful facilities for spawning new processes and retrieving their results; using that module is recommended to using this function. See the 이전 함수를 *subprocess* 모듈로 교체하기 section in the *subprocess* documentation for some helpful recipes.

유닉스에서, *waitstatus_to_exitcode()*를 사용하여 결과(종료 상태)를 종료 코드로 변환할 수 있습니다. 윈도우에서, 결과는 직접 종료 코드입니다.

`command`를 인자로 감사 이벤트(*auditing event*) `os.system`을 발생시킵니다.

가용성: Unix, Windows, not WASI, not Android, not iOS.

os.**times**()

현재 전역 프로세스 시간을 반환합니다. 반환 값은 5가지 어트리뷰트를 가진 객체입니다:

- user - user time
- system - system time
- children_user - user time of all child processes
- children_system - system time of all child processes
- elapsed - elapsed real time since a fixed point in the past

For backwards compatibility, this object also behaves like a five-tuple containing user, system, children_user, children_system, and elapsed in that order.

See the Unix manual page *times(2)* and times(3) manual page on Unix or the GetProcessTimes MSDN on Windows. On Windows, only user and system are known; the other attributes are zero.

가용성: Unix, Windows.

버전 3.3에서 변경: 반환형이 튜플에서 이름이 지정된 어트리뷰트를 가진 튜플류 객체로 변경되었습니다.

os.**wait**()

자식 프로세스가 완료될 때까지 기다렸다가, pid 및 종료 상태 표시를 포함하는 튜플을 반환합니다: 종료 상태 표시는 16비트 숫자인데, 하위 바이트가 프로세스를 죽인 시그널 번호이고, 상위 바이트가 종료 상태(시그널 번호가 0이면) 입니다; 코어 파일이 생성되면 하위 바이트의 상위 비트가 설정됩니다.

If there are no children that could be waited for, *ChildProcessError* is raised.

*waitstatus_to_exitcode()*를 사용하여 종료 상태를 종료 코드로 변환할 수 있습니다.

가용성: Unix, not WASI, not Android, not iOS.

> **더 보기**
>
> The other wait*() functions documented below can be used to wait for the completion of a specific child process and have more options. *waitpid()* is the only one also available on Windows.

os.**waitid**(*idtype, id, options, /*)

Wait for the completion of a child process.

idtype can be *P_PID*, *P_PGID*, *P_ALL*, or (on Linux) *P_PIDFD*. The interpretation of *id* depends on it; see their individual descriptions.

options is an OR combination of flags. At least one of *WEXITED*, *WSTOPPED* or *WCONTINUED* is required; *WNOHANG* and *WNOWAIT* are additional optional flags.

The return value is an object representing the data contained in the siginfo_t structure with the following attributes:

- si_pid (process ID)
- si_uid (real user ID of the child)
- si_signo (always *SIGCHLD*)
- si_status (the exit status or signal number, depending on si_code)
- si_code (see *CLD_EXITED* for possible values)

If *WNOHANG* is specified and there are no matching children in the requested state, None is returned. Otherwise, if there are no matching children that could be waited for, *ChildProcessError* is raised.

가용성: Unix, not WASI, not Android, not iOS.

Added in version 3.3.

버전 3.13에서 변경: This function is now available on macOS as well.

os.**waitpid**(*pid, options, /*)

이 함수의 세부 사항은 유닉스 및 윈도우에서 다릅니다.

유닉스에서: 프로세스 ID *pid*에 의해 주어진 자식 프로세스의 완료를 기다리고, 프로세스 ID와 종료 상태 표시(`wait()`처럼 인코딩됨)를 포함하는 튜플을 반환합니다. 호출의 의미는 정수 *options* 의 값에 영향을 받는데, 일반 작업의 경우 `0` 이어야 합니다.

pid 가 `0`보다 크면, `waitpid()`는 해당 프로세스에 대한 상태 정보를 요청합니다. *pid* 가 `0`이면, 현재 프로세스의 프로세스 그룹에 있는 모든 자식의 상태를 요청합니다. *pid* 가 `-1`이면, 현재 프로세스의 모든 자식의 상태를 요청합니다. *pid* 가 `-1`보다 작으면, 프로세스 그룹 `-pid`(*pid* 의 절댓값)에 있는 모든 프로세스의 상태를 요청합니다.

options is an OR combination of flags. If it contains `WNOHANG` and there are no matching children in the requested state, `(0, 0)` is returned. Otherwise, if there are no matching children that could be waited for, `ChildProcessError` is raised. Other options that can be used are `WUNTRACED` and `WCONTINUED`.

윈도우에서: 프로세스 핸들 *pid*로 지정된 프로세스가 완료될 때까지 기다리고, *pid*와 종료 상태를 8 비트 왼쪽으로 시프트 한 값을 포함하는 튜플을 반환합니다 (시프팅이 함수를 더 이식성 있게 만듭니다). `0`보다 작거나 같은 *pid* 는 윈도우에서 특별한 의미가 없고 예외가 발생합니다. 정수 *options* 의 값은 아무 효과가 없습니다. *pid* 는 id가 알려진 모든 프로세스를 가리킬 수 있습니다, 반드시 자식 프로세스일 필요는 없습니다. `P_NOWAIT`로 호출된 `spawn*` 함수는 적절한 프로세스 핸들을 반환합니다.

`waitstatus_to_exitcode()`를 사용하여 종료 상태를 종료 코드로 변환할 수 있습니다.

가용성: Unix, Windows, not WASI, not Android, not iOS.

버전 3.5에서 변경: 시스템 호출이 인터럽트 되고 시그널 처리기가 예외를 발생시키지 않으면, 함수는 이제 `InterruptedError` 예외를 일으키는 대신 시스템 호출을 재시도합니다 (이유는 **PEP 475**를 참조하세요).

os.**wait3**(*options*)

Similar to `waitpid()`, except no process id argument is given and a 3-element tuple containing the child's process id, exit status indication, and resource usage information is returned. Refer to `resource.getrusage()` for details on resource usage information. The *options* argument is the same as that provided to `waitpid()` and `wait4()`.

`waitstatus_to_exitcode()`를 사용하여 종료 상태를 종료 코드로 변환할 수 있습니다.

가용성: Unix, not WASI, not Android, not iOS.

os.**wait4**(*pid, options*)

Similar to `waitpid()`, except a 3-element tuple, containing the child's process id, exit status indication, and resource usage information is returned. Refer to `resource.getrusage()` for details on resource usage information. The arguments to `wait4()` are the same as those provided to `waitpid()`.

`waitstatus_to_exitcode()`를 사용하여 종료 상태를 종료 코드로 변환할 수 있습니다.

가용성: Unix, not WASI, not Android, not iOS.

os.**P_PID**

os.**P_PGID**

os.**P_ALL**

os.**P_PIDFD**

These are the possible values for *idtype* in `waitid()`. They affect how *id* is interpreted:

- `P_PID` - wait for the child whose PID is *id*.
- `P_PGID` - wait for any child whose progress group ID is *id*.
- `P_ALL` - wait for any child; *id* is ignored.

- `P_PIDFD` - wait for the child identified by the file descriptor *id* (a process file descriptor created with `pidfd_open()`).

가용성: Unix, not WASI, not Android, not iOS.

> **참고**
>
> `P_PIDFD` is only available on Linux >= 5.4.

Added in version 3.3.

Added in version 3.9: The `P_PIDFD` constant.

os.**WCONTINUED**

This *options* flag for `waitpid()`, `wait3()`, `wait4()`, and `waitid()` causes child processes to be reported if they have been continued from a job control stop since they were last reported.

가용성: Unix, not WASI, not Android, not iOS.

os.**WEXITED**

This *options* flag for `waitid()` causes child processes that have terminated to be reported.

The other `wait*` functions always report children that have terminated, so this option is not available for them.

가용성: Unix, not WASI, not Android, not iOS.

Added in version 3.3.

os.**WSTOPPED**

This *options* flag for `waitid()` causes child processes that have been stopped by the delivery of a signal to be reported.

This option is not available for the other `wait*` functions.

가용성: Unix, not WASI, not Android, not iOS.

Added in version 3.3.

os.**WUNTRACED**

This *options* flag for `waitpid()`, `wait3()`, and `wait4()` causes child processes to also be reported if they have been stopped but their current state has not been reported since they were stopped.

This option is not available for `waitid()`.

가용성: Unix, not WASI, not Android, not iOS.

os.**WNOHANG**

This *options* flag causes `waitpid()`, `wait3()`, `wait4()`, and `waitid()` to return right away if no child process status is available immediately.

가용성: Unix, not WASI, not Android, not iOS.

os.**WNOWAIT**

This *options* flag causes `waitid()` to leave the child in a waitable state, so that a later `wait*()` call can be used to retrieve the child status information again.

This option is not available for the other `wait*` functions.

가용성: Unix, not WASI, not Android, not iOS.

os.**CLD_EXITED**

os.**CLD_KILLED**

os.**CLD_DUMPED**

os.**CLD_TRAPPED**

os.**CLD_STOPPED**

os.**CLD_CONTINUED**

These are the possible values for `si_code` in the result returned by *waitid()*.

가용성: Unix, not WASI, not Android, not iOS.

Added in version 3.3.

버전 3.9에서 변경: *CLD_KILLED*와 *CLD_STOPPED* 값을 추가했습니다.

os.**waitstatus_to_exitcode**(*status*)

대기 상태(wait status)를 종료 코드로 변환합니다.

유닉스에서:

- 프로세스가 정상적으로 종료되면 (`WIFEXITED(status)` 가 참이면), 프로세스 종료 상태를 반환합니다 (`WEXITSTATUS(status)` 를 반환합니다): 결과는 0보다 크거나 같습니다.
- 프로세스가 시그널에 의해 종료되면 (`WIFSIGNALED(status)` 가 참이면), `-signum`을 반환합니다, 여기서 *signum*은 프로세스를 종료시킨 시그널 번호입니다 (`-WTERMSIG(status)` 를 반환합니다): 결과는 0보다 작습니다.
- 그렇지 않으면, *ValueError*를 발생시킵니다.

윈도우에서, 8비트만큼 오른쪽으로 시프트된 *status*를 반환합니다.

유닉스에서, 프로세스가 추적되고 있거나 *waitpid()*가 *WUNTRACED* 옵션으로 호출되었으면, 호출자는 먼저 `WIFSTOPPED(status)` 가 참인지 확인해야 합니다. `WIFSTOPPED(status)` 가 참이면 이 함수를 호출하면 안 됩니다.

더 보기

WIFEXITED(), *WEXITSTATUS()*, *WIFSIGNALED()*, *WTERMSIG()*, *WIFSTOPPED()*, *WSTOPSIG()* 함수.

가용성: Unix, Windows, not WASI, not Android, not iOS.

Added in version 3.9.

다음 함수들은 *system()*, *wait()* 또는 *waitpid()*에 의해 반환된 프로세스 상태 코드를 매개 변수로 받아들입니다. 이것들은 프로세스의 처리를 결정하는 데 사용될 수 있습니다.

os.**WCOREDUMP**(*status*, /)

프로세스에 대해 코어 덤프가 생성되었으면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

이 함수는 *WIFSIGNALED()*가 참일 때만 사용해야 합니다.

가용성: Unix, not WASI, not Android, not iOS.

os.**WIFCONTINUED**(*status*)

Return `True` if a stopped child has been resumed by delivery of *SIGCONT* (if the process has been continued from a job control stop), otherwise return `False`.

WCONTINUED 옵션을 참조하십시오.

가용성: Unix, not WASI, not Android, not iOS.

os.**WIFSTOPPED**(*status*)

시그널의 전달로 인해 프로세스가 중지되었으면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

*WIFSTOPPED()*는 *WUNTRACED* 옵션을 사용하여 *waitpid()*을 호출했거나 프로세스가 추적되고 있을 때만 `True`를 반환합니다 (*ptrace(2)*를 참조하십시오).

가용성: Unix, not WASI, not Android, not iOS.

os.**WIFSIGNALED**(*status*)

시그널로 인해 프로세스가 종료되었으면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

가용성: Unix, not WASI, not Android, not iOS.

os.**WIFEXITED**(*status*)

프로세스가 정상 종료했으면 `True`를 반환합니다, 즉 `exit()` 나 `_exit()` 를 호출했거나, `main()` 에서 반환하여; 그렇지 않으면 `False`를 반환합니다.

가용성: Unix, not WASI, not Android, not iOS.

os.**WEXITSTATUS**(*status*)

프로세스 종료 상태를 반환합니다.

이 함수는 *WIFEXITED()* 가 참일 때만 사용해야 합니다.

가용성: Unix, not WASI, not Android, not iOS.

os.**WSTOPSIG**(*status*)

프로세스를 멈추게 한 시그널을 반환합니다.

이 함수는 *WIFSTOPPED()* 가 참일 때만 사용해야 합니다.

가용성: Unix, not WASI, not Android, not iOS.

os.**WTERMSIG**(*status*)

프로세스를 종료시킨 시그널의 번호를 반환합니다.

이 함수는 *WIFSIGNALED()* 가 참일 때만 사용해야 합니다.

가용성: Unix, not WASI, not Android, not iOS.

16.1.8 스케줄러에 대한 인터페이스

이 함수들은 운영 체제가 프로세스에 CPU 시간을 할당하는 방법을 제어합니다. 일부 유닉스 플랫폼에서만 사용할 수 있습니다. 자세한 내용은 유닉스 매뉴얼 페이지를 참조하십시오.

Added in version 3.3.

다음 스케줄 정책은 운영 체제에서 지원하는 경우 공개됩니다.

os.**SCHED_OTHER**

기본 스케줄 정책.

os.**SCHED_BATCH**

컴퓨터의 나머지 부분에서 반응성을 유지하려고 하는 CPU 집약적인 프로세스를 위한 스케줄 정책.

os.**SCHED_DEADLINE**

Scheduling policy for tasks with deadline constraints.

Added in version 3.14.

os.**SCHED_IDLE**

매우 낮은 우선순위의 배경 작업에 대한 스케줄 정책.

os.**SCHED_NORMAL**

Alias for *SCHED_OTHER*.

Added in version 3.14.

os.**SCHED_SPORADIC**

간헐적인 서버 프로그램을 위한 스케줄 정책.

os.**SCHED_FIFO**

선입 선출 (First In First Out) 스케줄 정책.

os.**SCHED_RR**

라운드 로빈 스케줄 정책.

os.**SCHED_RESET_ON_FORK**

이 플래그는 다른 스케줄 정책과 OR 될 수 있습니다. 이 플래그가 설정되어있는 프로세스가 포크 할 때, 자식의 스케줄링 정책 및 우선순위가 기본값으로 재설정됩니다.

class os.**sched_param**(*sched_priority*)

이 클래스는 `sched_setparam()`, `sched_setscheduler()`, 및 `sched_getparam()` 에서 사용되는 튜닝 가능한 스케줄 파라미터를 나타냅니다. 불변입니다.

현재, 가능한 매개 변수는 하나뿐입니다:

sched_priority

스케줄 정책의 스케줄 우선순위.

os.**sched_get_priority_min**(*policy*)

policy 의 최소 우선순위 값을 가져옵니다. *policy* 는 위의 스케줄 정책 상수 중 하나입니다.

os.**sched_get_priority_max**(*policy*)

policy 의 최대 우선순위 값을 가져옵니다. *policy* 는 위의 스케줄 정책 상수 중 하나입니다.

os.**sched_setscheduler**(*pid*, *policy*, *param*, /)

PID 가 *pid* 인 프로세스의 스케줄 정책을 설정합니다. *pid* 가 0 이면, 호출하는 프로세스를 의미합니다. *policy* 는 위의 스케줄 정책 상수 중 하나입니다. *param* 은 `sched_param` 인스턴스입니다.

os.**sched_getscheduler**(*pid*, /)

PID 가 *pid* 인 프로세스의 스케줄 정책을 반환합니다. *pid* 가 0 이면, 호출하는 프로세스를 의미합니다. 결과는 위의 스케줄 정책 상수 중 하나입니다.

os.**sched_setparam**(*pid*, *param*, /)

Set the scheduling parameters for the process with PID *pid*. A *pid* of 0 means the calling process. *param* is a `sched_param` instance.

os.**sched_getparam**(*pid*, /)

PID 가 *pid* 인 프로세스의 스케줄 매개 변수를 `sched_param` 인스턴스로 반환합니다. *pid* 가 0 이면 호출하는 프로세스를 의미합니다.

os.**sched_rr_get_interval**(*pid*, /)

PID 가 *pid* 인 프로세스의 라운드 로빈 퀀텀을 초 단위로 반환합니다. *pid* 가 0 이면 호출하는 프로세스를 의미합니다.

os.**sched_yield**()

Voluntarily relinquish the CPU. See `sched_yield(2)` for details.

os.**sched_setaffinity**(*pid*, *mask*, /)

PID 가 *pid* 인 프로세스(또는 0 이면 현재 프로세스)를 CPU 집합으로 제한합니다. *mask* 는 프로세스가 제한되어야 하는 CPU 집합을 나타내는 정수의 이터러블입니다.

os.**sched_getaffinity**(*pid*, /)

Return the set of CPUs the process with PID *pid* is restricted to.

If *pid* is zero, return the set of CPUs the calling thread of the current process is restricted to.

See also the `process_cpu_count()` function.

16.1.9 기타 시스템 정보

os.**confstr**(*name*, /)

문자열 값 시스템 구성 값을 반환합니다. *name* 은 조회할 구성 값을 지정합니다; 정의된 시스템 값의 이름인 문자열일 수 있습니다; 이 이름은 여러 표준(POSIX, 유닉스 95, 유닉스 98 및 기타)에서 지정됩니다. 일부 플랫폼은 추가 이름도 정의합니다. 호스트 운영 체제에 알려진 이름은 `confstr_names`

딕셔너리의 키로 제공됩니다. 해당 매핑에 포함되지 않은 구성 변수를 위해, *name*에 정수를 전달하는 것도 허용됩니다.

name 으로 지정된 구성 값이 정의되어 있지 않으면, `None`이 반환됩니다.

name 이 문자열이고 알 수 없으면, `ValueError`가 발생합니다. *name*에 대한 특정 값이 호스트 시스템에서 지원되지 않으면, `confstr_names`에 포함되어 있어도, 에러 번호 `errno.EINVAL`로 `OSError`가 발생합니다.

가용성: Unix.

`os.`**`confstr_names`**

`confstr()`에서 허용하는 이름을 호스트 운영 체제가 해당 이름에 대해 정의한 정숫값으로 매핑하는 딕셔너리입니다. 이것은 시스템에 알려진 이름 집합을 판별하는 데 사용될 수 있습니다.

가용성: Unix.

`os.`**`cpu_count`**`()`

Return the number of logical CPUs in the **system**. Returns `None` if undetermined.

The `process_cpu_count()` function can be used to get the number of logical CPUs usable by the calling thread of the **current process**.

Added in version 3.4.

버전 3.13에서 변경: If `-X cpu_count` is given or `PYTHON_CPU_COUNT` is set, `cpu_count()` returns the overridden value *n*.

`os.`**`getloadavg`**`()`

마지막 1, 5, 15분에 걸쳐 평균한 시스템 실행 대기열의 프로세스 수를 반환하거나, 로드 평균을 얻을 수 없으면, `OSError`를 발생시킵니다.

가용성: Unix.

`os.`**`process_cpu_count`**`()`

Get the number of logical CPUs usable by the calling thread of the **current process**. Returns `None` if undetermined. It can be less than `cpu_count()` depending on the CPU affinity.

The `cpu_count()` function can be used to get the number of logical CPUs in the **system**.

If `-X cpu_count` is given or `PYTHON_CPU_COUNT` is set, `process_cpu_count()` returns the overridden value *n*.

See also the `sched_getaffinity()` function.

Added in version 3.13.

`os.`**`sysconf`**`(`*name*, /`)`

정숫값 시스템 구성 값을 반환합니다. *name* 으로 지정된 구성 값이 정의되어 있지 않으면, `-1`이 반환됩니다. `confstr()`의 *name* 매개 변수에 관한 주석은 여기에도 적용됩니다; 알려진 이름에 대한 정보를 제공하는 딕셔너리는 `sysconf_names`에 의해 제공됩니다.

가용성: Unix.

`os.`**`sysconf_names`**

`sysconf()`에서 허용하는 이름을 호스트 운영 체제가 해당 이름에 대해 정의한 정숫값으로 매핑하는 딕셔너리입니다. 이것은 시스템에 알려진 이름 집합을 판별하는 데 사용될 수 있습니다.

가용성: Unix.

버전 3.11에서 변경: Add `'SC_MINSIGSTKSZ'` name.

다음 데이터값들은 경로 조작 연산을 지원하는 데 사용됩니다. 이는 모든 플랫폼에서 정의됩니다.

경로명에 대한 고수준 연산은 `os.path` 모듈에서 정의됩니다.

os.**curdir**

현재 디렉터리를 가리키기 위해 운영 체제에서 사용하는 상수 문자열. 이것은 윈도우 및 POSIX의 경우 '.'입니다. *os.path*를 통해서도 제공됩니다.

os.**pardir**

부모 디렉터리를 가리키기 위해 운영 체제에서 사용하는 상수 문자열입니다. 이것은 윈도우 및 POSIX의 경우 '..'입니다. *os.path*를 통해서도 제공됩니다.

os.**sep**

경로명 구성 요소를 분리하기 위해 운영 체제에서 사용하는 문자. 이것은 POSIX의 경우 '/'이고, 윈도우의 경우 '\\'입니다. 이것을 아는 것만으로는 경로명을 구문 분석하거나 이어붙일 수는 없습니다만 — *os.path.split()*와 *os.path.join()*를 사용하세요 — 가끔 유용합니다. *os.path*를 통해서도 제공됩니다.

os.**altsep**

경로명 구성 요소를 분리하기 위해 운영 체제에서 사용하는 대체 문자이거나, 단 하나의 구분 문자만 있는 경우 None입니다. sep가 백 슬래시인 윈도우 시스템에서는 '/'로 설정됩니다. *os.path*를 통해서도 제공됩니다.

os.**extsep**

기본 파일명과 확장자를 구분하는 문자; 예를 들어, os.py에서 '.'. *os.path*를 통해서도 제공됩니다.

os.**pathsep**

검색 경로 구성 요소(PATH에서와 같이)를 분리하기 위해 운영 체제에서 관습적으로 사용하는 문자, 가령 POSIX의 ':' 또는 윈도우의 ';'. *os.path*를 통해서도 제공됩니다.

os.**defpath**

환경에 'PATH' 키가 없을 때, *exec*p** 및 *spawn*p**에서 사용하는 기본 검색 경로. *os.path*를 통해서도 제공됩니다.

os.**linesep**

현재 플랫폼에서 행을 분리(또는 종료)하는 데 사용되는 문자열. 이는 POSIX의 '\n'와 같은 단일 문자이거나, 윈도우의 '\r\n'와 같은 여러 문자일 수 있습니다. 텍스트 모드로 열린(기본값) 파일에 쓸 때 줄 종결자로 *os.linesep*를 사용하지 마십시오; 대신 모든 플랫폼에서 단일 '\n'를 사용하십시오.

os.**devnull**

널(null) 장치의 파일 경로. 예를 들어: POSIX의 경우 '/dev/null', 윈도우의 경우 'nul'. *os.path*를 통해서도 제공됩니다.

os.**RTLD_LAZY**

os.**RTLD_NOW**

os.**RTLD_GLOBAL**

os.**RTLD_LOCAL**

os.**RTLD_NODELETE**

os.**RTLD_NOLOAD**

os.**RTLD_DEEPBIND**

setdlopenflags() 및 *getdlopenflags()* 함수에 사용하는 플래그. 각 플래그가 의미하는 바는 유닉스 매뉴얼 페이지 *dlopen(3)*를 참조하십시오.

Added in version 3.3.

16.1.10 난수

os.**getrandom**(*size*, *flags=0*)

최대 *size* 크기의 난수 바이트열을 업습니다. 이 함수는 요청한 것보다 짧은 바이트열를 반환할 수 있습니다.

이 바이트열은 사용자 공간 난수 발생기를 시드 하거나 암호화 목적으로 사용할 수 있습니다.

getrandom() 는 장치 드라이버 및 기타 환경 소음원에서 수집한 엔트로피에 의존합니다. 대량의 데이터를 불필요하게 읽는 것은 /dev/random 및 /dev/urandom 장치의 다른 사용자에게 부정적인 영향을 미칩니다.

The flags argument is a bit mask that can contain zero or more of the following values ORed together: *os.GRND_RANDOM* and *GRND_NONBLOCK*.

See also the Linux getrandom() manual page.

가용성: Linux >= 3.17.

Added in version 3.6.

os.**urandom**(*size*, /)

Return a bytestring of *size* random bytes suitable for cryptographic use.

이 함수는 OS 종속적인 임의성 소스에서 난수 바이트열을 반환합니다. 반환된 데이터는 암호화 응용에 충분하도록 예측할 수 없어야 하지만, 정확한 품질은 OS 구현에 따라 달라집니다.

리눅스에서, getrandom() 시스템 호출을 사용할 수 있으면, 블로킹 모드로 사용됩니다: 시스템의 urandom 엔트로피 풀이 초기화될 때까지 블록 됩니다 (커널이 128 비트의 엔트로피를 수집합니다). 이유는 **PEP 524**를 참조하십시오. 리눅스에서, *getrandom()* 함수는 (*GRND_NONBLOCK* 플래그를 사용하여) 비 블로킹 모드로 난수 바이트열을 얻거나, 시스템 urandom 엔트로피 풀이 초기화될 때까지 폴링 할 수 있습니다.

유닉스류 시스템에서, /dev/urandom 장치에서 난수 바이트열을 읽습니다. /dev/urandom 장치를 사용할 수 없거나 읽을 수 없으면, *NotImplementedError* 예외가 발생합니다.

On Windows, it will use BCryptGenRandom().

> **더 보기**
>
> *secrets* 모듈은 고수준 함수를 제공합니다. 플랫폼에서 제공되는 난수 발생기에 대한 사용하기 쉬운 인터페이스는 *random.SystemRandom*를 참조하십시오.

버전 3.5에서 변경: 리눅스 3.17 및 이후 버전에서, 이제 getrandom() 시스템 호출을 사용할 수 있으면 사용합니다. OpenBSD 5.6 이상에서, C getentropy() 함수가 이제 사용됩니다. 이 함수들은 내부 파일 기술자의 사용을 피합니다.

버전 3.5.2에서 변경: 리눅스에서, getrandom() 시스템 호출이 블록 하면 (urandom 엔트로피 풀이 아직 초기화되지 않았으면), /dev/urandom을 읽는 것으로 대체됩니다.

버전 3.6에서 변경: 리눅스에서, getrandom() 은 이제 보안을 강화하기 위해 블로킹 모드로 사용됩니다.

버전 3.11에서 변경: On Windows, BCryptGenRandom() is used instead of CryptGenRandom() which is deprecated.

os.**GRND_NONBLOCK**

기본적으로, /dev/random에서 읽을 때, *getrandom()*는 사용할 수 있는 난수 바이트열이 없으면 블록 하고, /dev/urandom에서 읽을 때는, 엔트로피 풀이 아직 초기화되지 않았으면 블록 합니다.

GRND_NONBLOCK 플래그가 설정되면, *getrandom()*는 이럴 때 블록 하지 않고, 대신 즉시 *BlockingIOError*를 발생시킵니다.

Added in version 3.6.

os.**GRND_RANDOM**

이 비트가 설정되면, /dev/urandom 풀 대신 /dev/random 풀에서 난수 바이트열을 얻습니다.

Added in version 3.6.

16.2 io — 스트림 작업을 위한 핵심 도구

소스 코드: Lib/io.py

16.2.1 개요

io 모듈은 다양한 유형의 I/O를 처리하기 위한 파이썬의 주 장치를 제공합니다. I/O에는 세 가지 주요 유형이 있습니다: 텍스트(*text*) *I/O*, 바이너리(*binary*) *I/O* 및 원시(*raw*) *I/O*. 이들은 일반적인 범주이며 다양한 배경 저장소를 각각에 사용할 수 있습니다. 이러한 범주 중 하나에 속하는 구상 객체를 파일 객체라고 합니다. 다른 일반적인 용어는 스트림(*stream*)과 파일류 객체(*file-like object*)입니다.

범주와 상관없이, 각 구상 스트림 객체에는 다양한 기능이 있습니다: 읽기 전용, 쓰기 전용 또는 읽고-쓰기일 수 있습니다. 또한 임의의 무작위 액세스(임의의 위치로 전방이나 후방 탐색)를 허용하거나 순차적 액세스(예를 들어 소켓이나 파이프의 경우)만 허용할 수 있습니다.

모든 스트림은 그것에 제공하는 데이터형에 주의를 기울입니다. 예를 들어 str 객체를 바이너리 스트림의 write() 메서드에 제공하면 TypeError가 발생합니다. 텍스트 스트림의 write() 메서드에 bytes 객체를 제공해도 마찬가지입니다.

버전 3.3에서 변경: IOError가 이제는 OSError의 별칭이라서, IOError를 발생시켰던 연산은 이제 OSError를 발생시킵니다.

텍스트 I/O

텍스트 I/O는 str 객체를 기대하고 생성합니다. 이는 배경 저장소가 네이티브 하게 바이트열로 구성되었을 때마다(가령 파일의 경우), 플랫폼별 줄 넘김 문자의 선택적 변환뿐만 아니라 데이터의 인코딩과 디코딩이 투명하게 이루어짐을 의미합니다.

텍스트 스트림을 만드는 가장 쉬운 방법은 open()을 사용하는 것이고, 선택적으로 인코딩을 지정합니다:

```
f = open("myfile.txt", "r", encoding="utf-8")
```

인 메모리 텍스트 스트림도 StringIO 객체로 제공됩니다:

```
f = io.StringIO("some initial text data")
```

> **참고**
>
> When working with a non-blocking stream, be aware that read operations on text I/O objects might raise a BlockingIOError if the stream cannot perform the operation immediately.

텍스트 스트림 API는 TextIOBase의 설명서에 자세히 설명되어 있습니다.

바이너리 I/O

바이너리 I/O(버퍼링 된(*buffered*) *I/O*라고도 합니다)는 바이트열류 객체를 기대하고 bytes 객체를 생성합니다. 인코딩, 디코딩 또는 줄 넘김 변환이 수행되지 않습니다. 이 범주의 스트림은 모든 종류의 텍스트가 아닌 데이터에 사용할 수 있으며, 텍스트 데이터 처리를 수동으로 제어해야 할 때도 사용할 수 있습니다.

바이너리 스트림을 만드는 가장 쉬운 방법은 모드 문자열에 'b'를 제공하여 open()을 사용하는 것입니다:

```
f = open("myfile.jpg", "rb")
```

인 메모리 바이너리 스트림도 BytesIO 객체로 제공됩니다:

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

바이너리 스트림 API는 *BufferedIOBase* 설명서에 자세히 설명되어 있습니다.

다른 라이브러리 모듈은 텍스트나 바이너리 스트림을 만드는 다른 방법을 제공할 수 있습니다. 예를 들어 *socket.socket.makefile()*을 참조하십시오.

원시 I/O

원시 I/O(버퍼링 되지 않은(*unbuffered*) *I/O*라고도 합니다)는 일반적으로 바이너리와 텍스트 스트림을 위한 저수준 빌딩 블록으로 사용됩니다; 사용자 코드에서 원시 스트림을 직접 조작하는 것은 거의 유용하지 않습니다. 그런데도, 버퍼링을 비활성화해서 바이너리 모드로 파일을 열어 원시 스트림을 만들 수 있습니다:

```
f = open("myfile.jpg", "rb", buffering=0)
```

원시 스트림 API는 *RawIOBase* 설명서에 자세히 설명되어 있습니다.

16.2.2 텍스트 인코딩

The default encoding of *TextIOWrapper* and *open()* is locale-specific (*locale.getencoding()*).

However, many developers forget to specify the encoding when opening text files encoded in UTF-8 (e.g. JSON, TOML, Markdown, etc…) since most Unix platforms use UTF-8 locale by default. This causes bugs because the locale encoding is not UTF-8 for most Windows users. For example:

```
# 파일에 비 ASCII 문자가 있을 때 윈도우에서 작동하지 않을 수 있습니다.
with open("README.md") as f:
    long_description = f.read()
```

Accordingly, it is highly recommended that you specify the encoding explicitly when opening text files. If you want to use UTF-8, pass `encoding="utf-8"`. To use the current locale encoding, `encoding="locale"` is supported since Python 3.10.

> **더 보기**
>
> ***Python UTF-8 Mode***
> Python UTF-8 Mode can be used to change the default encoding to UTF-8 from locale-specific encoding.
>
> **PEP 686**
> Python 3.15 will make *Python UTF-8 Mode* default.

Opt-in EncodingWarning

Added in version 3.10: See **PEP 597** for more details.

To find where the default locale encoding is used, you can enable the `-X warn_default_encoding` command line option or set the `PYTHONWARNDEFAULTENCODING` environment variable, which will emit an *EncodingWarning* when the default encoding is used.

If you are providing an API that uses *open()* or *TextIOWrapper* and passes `encoding=None` as a parameter, you can use *text_encoding()* so that callers of the API will emit an *EncodingWarning* if they don' t pass an `encoding`. However, please consider using UTF-8 by default (i.e. `encoding="utf-8"`) for new APIs.

16.2.3 고수준 모듈 인터페이스

`io.`**`DEFAULT_BUFFER_SIZE`**

모듈의 버퍼링 된 I/O 클래스에서 사용하는 기본 버퍼 크기를 포함하는 int. *open()*은 가능하면 파일의 blksize(*os.stat()*으로 얻은)를 사용합니다.

`io.`**`open`**(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

이것은 내장 *open()* 함수의 별칭입니다.

이 함수는 인자 *path*, *mode* 및 *flags*로 감사 이벤트 `open`을 발생시킵니다. *mode*와 *flags* 인자는 원래 호출에서 수정되었거나 유추되었을 수 있습니다.

`io.open_code`(*path*)

제공된 파일을 `'rb'` 모드로 엽니다. 이 함수는 내용을 실행 코드로 취급하려고 할 때 사용해야 합니다.

*path*는 `str`이고 절대 경로여야 합니다.

이 함수의 동작은 `PyFile_SetOpenCodeHook()` 에 대한 이전 호출로 재정의될 수 있습니다. 하지만, *path*가 `str`이고 절대 경로임을 가정할 때, `open_code(path)` 는 항상 `open(path, 'rb')` 와 같게 동작해야 합니다. 동작의 재정의는 파일의 추가 유효성 검사나 사전 처리를 위한 것입니다.

Added in version 3.8.

`io.text_encoding`(*encoding*, *stacklevel=2*, /)

This is a helper function for callables that use `open()` or `TextIOWrapper` and have an `encoding=None` parameter.

This function returns *encoding* if it is not `None`. Otherwise, it returns `"locale"` or `"utf-8"` depending on *UTF-8 Mode*.

This function emits an `EncodingWarning` if `sys.flags.warn_default_encoding` is true and *encoding* is `None`. *stacklevel* specifies where the warning is emitted. For example:

```
def read_text(path, encoding=None):
    encoding = io.text_encoding(encoding)  # stacklevel=2
    with open(path, encoding) as f:
        return f.read()
```

In this example, an `EncodingWarning` is emitted for the caller of `read_text()`.

See 텍스트 인코딩 for more information.

Added in version 3.10.

버전 3.11에서 변경: `text_encoding()` returns "utf-8" when UTF-8 mode is enabled and *encoding* is `None`.

exception `io.BlockingIOError`

이것은 내장 `BlockingIOError` 예외에 대한 호환 별칭입니다.

exception `io.UnsupportedOperation`

지원되지 않는 연산이 스트림에서 호출될 때 발생하는 `OSError`와 `ValueError`를 상속하는 예외.

더 보기

sys

표준 IO 스트림을 포함합니다: `sys.stdin`, `sys.stdout` 및 `sys.stderr`.

16.2.4 클래스 위계

I/O 스트림의 구현은 클래스의 위계(hierarchy)로 구성됩니다. 먼저 다양한 범주의 스트림을 지정하는 데 사용되는 추상 베이스 클래스(ABC)가 있고, 그다음으로 표준 스트림 구현을 제공하는 구상 클래스가 있습니다.

참고

추상 베이스 클래스는 또한 구상 스트림 클래스의 구현을 돕기 위해 일부 메서드의 기본 구현을 제공합니다. 예를 들어, `BufferedIOBase`는 최적화되지 않은 `readinto()` 와 `readline()` 구현을 제공합니다.

I/O 위계의 맨 위에는 추상 베이스 클래스 *IOBase*가 있습니다. 스트림에 대한 기본 인터페이스를 정의합니다. 그러나 스트림에 대한 읽기와 쓰기가 분리되지 않음에 유의하십시오; 구현은 주어진 연산을 지원하지 않으면 *UnsupportedOperation*을 발생시킬 수 있습니다.

RawIOBase ABC는 *IOBase*를 확장합니다. 스크림에 대한 바이트열의 읽기와 쓰기를 처리합니다. *FileIO*는 *RawIOBase*를 서브 클래싱하여 기계의 파일 시스템에 있는 파일에 대한 인터페이스를 제공합니다.

BufferedIOBase ABC는 *IOBase*를 확장합니다. 원시 바이너리 스트림(*RawIOBase*)에 대한 버퍼링을 다룹니다. 이것의 서브 클래스, *BufferedWriter*, *BufferedReader* 및 *BufferedRWPair*는 각각 쓸 수 있는, 읽을 수 있는, 그리고 읽고 쓸 수 있는 원시 바이너리 스트림을 버퍼링합니다. *BufferedRandom*은 탐색할 수 있는 (seekable) 스트림에 버퍼 인터페이스를 제공합니다. 또 다른 *BufferedIOBase* 서브 클래스 *BytesIO*는 인 메모리 바이트열의 스트림입니다.

TextIOBase ABC는 *IOBase*를 확장합니다. 이것은 바이트가 텍스트를 나타내는 스트림을 다루고, 문자열과의 인코딩과 디코딩을 처리합니다. *TextIOBase*를 확장하는 *TextIOWrapper*는 버퍼링 된 원시 스트림(*BufferedIOBase*)에 대한 버퍼링 된 텍스트 인터페이스입니다. 마지막으로, *StringIO*는 텍스트에 대한 인 메모리 스트림입니다.

인자 이름은 명세의 일부가 아니며, *open()*의 인자는 키워드 인자로만 사용하려는 의도입니다.

다음 표는 *io* 모듈에서 제공하는 ABC를 요약합니다:

ABC	상속	스텁(stub) 메서드	믹스 인 메서드와 프로퍼티
IOBase		`fileno`, `seek` 및 `truncate`	`close`, `closed`, `__enter__`, `__exit__`, `flush`, `isatty`, `__iter__`, `__next__`, `readable`, `readline`, `readlines`, `seekable`, `tell`, `writable` 및 `writelines`
RawIOBase	*IOBase*	`readinto` 와 `write`	상속된 *IOBase* 메서드, `read` 및 `readall`
BufferedIOBase	*IOBase*	`detach`, `read`, `read1` 및 `write`	상속된 *IOBase* 메서드, `readinto` 및 `readinto1`
TextIOBase	*IOBase*	`detach`, `read`, `readline` 및 `write`	상속된 *IOBase* 메서드, `encoding`, `errors` 및 `newlines`

I/O 베이스 클래스

class io.IOBase

모든 I/O 클래스의 추상 베이스 클래스.

이 클래스는 파생 클래스가 선택적으로 재정의할 수 있는 많은 메서드에 대해 빈 추상 구현을 제공합니다; 기본 구현은 읽거나 쓰거나 탐색할 수 없는 파일을 나타냅니다.

*IOBase*가 서명이 다양하기 때문에 `read()` 나 `write()` 를 선언하지 않더라도, 구현과 클라이언트는 해당 메서드를 인터페이스의 일부로 고려해야 합니다. 또한, 지원하지 않는 연산이 호출될 때 구현은 *ValueError*(또는 *UnsupportedOperation*)를 발생시킬 수 있습니다.

파일에서 읽거나 파일에 쓰는 바이너리 데이터에 사용되는 기본형은 *bytes*입니다. 다른 바이트열류 객체도 메서드 인자로 허용됩니다. 텍스트 I/O 클래스는 *str* 데이터로 작동합니다.

닫힌 스트림에 대한 모든 메서드(조회조차도) 호출은 정의되어 있지 않습니다. 이 경우 구현은 *ValueError*를 발생시킬 수 있습니다.

IOBase(및 그 서브 클래스)는 이터레이터 프로토콜을 지원합니다. 즉, 스트림에서 줄을 산출하면서 *IOBase* 객체를 이터레이트 할 수 있습니다. 스트림이 바이너리 스트림(바이트열을 산출합니다)인지 텍스트 스트림(문자열을 산출합니다)인지에 따라 줄은 약간 다르게 정의됩니다. 아래 *readline()*을 참조하십시오.

*IOBase*는 컨텍스트 관리자이기도 해서, `with` 문을 지원합니다. 이 예에서, *file*은 `with` 문의 스위트가 완료된 후에 닫힙니다 — 예외가 발생하더라도:

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

*IOBase*는 다음 데이터 어트리뷰트와 메서드를 제공합니다:

close()

이 스트림을 플러시하고 닫습니다. 파일이 이미 닫혔으면 이 메서드는 효과가 없습니다. 일단 파일이 닫히면, 파일에 대한 모든 연산(예를 들어 읽기나 쓰기)이 *ValueError*를 발생시킵니다.

편의상, 이 메서드를 두 번 이상 호출할 수 있습니다; 그러나 첫 번째 호출만 효과가 있습니다.

closed

스트림이 닫혔으면 `True`.

fileno()

존재한다면 스트림의 하부 파일 기술자(정수)를 반환합니다. IO 객체가 파일 기술자를 사용하지 않으면 *OSError*가 발생합니다.

flush()

해당하면 스트림의 쓰기 버퍼를 플러시합니다. 이것은 읽기 전용과 비 블로킹 스트림에 대해서는 아무것도 하지 않습니다.

isatty()

스트림이 대화형이면 (즉, 터미널/tty 장치에 연결되었으면) `True`를 반환합니다.

readable()

스트림을 읽을 수 있으면 `True`를 반환합니다. `False`이면, `read()`는 *OSError*를 발생시킵니다.

readline(*size=-1*, /)

스트림에서 한 줄을 읽고 반환합니다. *size*가 지정되면, 최대 *size* 바이트를 읽습니다.

줄 종결자는 바이너리 파일의 경우 항상 `b'\n'`입니다; 텍스트 파일의 경우, *open()*에 대한 *newline* 인자를 사용하여 인식되는 줄 종결자를 선택할 수 있습니다.

readlines(*hint=-1*, /)

스트림에서 줄 리스트를 읽고 반환합니다. *hint*는 읽을 줄 수를 제어하도록 지정할 수 있습니다: 지금까지 모든 줄의 총 크기(바이트/문자 단위)가 *hint*를 초과하면 더는 줄을 읽지 않습니다.

hint values of `0` or less, as well as `None`, are treated as no hint.

`file.readlines()`를 호출하지 않고 `for line in file: ...`을 사용하여 파일 객체를 이미 이터레이트 할 수 있음에 유의하십시오.

seek(*offset*, *whence=os.SEEK_SET*, /)

스트림 위치를 *whence*가 가리키는 위치를 기준으로 상대적으로 해석되는 지정된 바이트 *offset*으로 변경하고, 새 절대 위치를 반환합니다. *whence*의 값은 다음과 같습니다:

- *os.SEEK_SET* 또는 `0` – 스트림의 시작(기본값); *offset*은 0이거나 양수여야 합니다
- *os.SEEK_CUR* 또는 `1` – 현재 스트림 위치; *offset*은 음수일 수 있습니다
- *os.SEEK_END* 또는 `2` – 스트림의 끝; *offset*은 일반적으로 음수입니다

Added in version 3.1: `SEEK_*` 상수.

Added in version 3.3: 일부 운영 체제는 *os.SEEK_HOLE*이나 *os.SEEK_DATA*와 같은 추가 값을 지원할 수 있습니다. 파일에 대해 유효한 값은 그것이 텍스트나 바이너리 모드 중 어느 것으로 열렸는지에 따라 달라질 수 있습니다.

seekable()

스트림이 무작위 액세스를 지원하면 `True`를 반환합니다. `False`이면, *seek()*, *tell()* 및 *truncate()*가 *OSError*를 발생시킵니다.

tell()

현재의 스트림 위치를 반환합니다.

truncate(*size=None*, /)

바이트 단위로 지정된 *size*로 스트림 크기를 조정합니다 (또는 *size*가 지정되지 않으면 현재 위치). 현재 스트림 위치는 변경되지 않습니다. 이 크기 조정은 현재 파일 크기를 늘리거나 줄일 수 있습니다. 확장의 경우, 새 파일 영역의 내용은 플랫폼에 따라 다릅니다 (대부분의 시스템에서, 추가 바이트는 0으로 채워집니다). 새 파일 크기가 반환됩니다.

버전 3.5에서 변경: 윈도우는 이제 확장 시 파일을 0으로 채웁니다.

writable()

스트림이 쓰기를 지원하면 `True`를 반환합니다. `False`이면, `write()`와 `truncate()`는 `OSError`를 발생시킵니다.

writelines(*lines*, /)

스트림에 줄 리스트를 씁니다. 줄 구분자는 추가되지 않아서, 제공된 각 줄 끝에 줄 구분자가 있는 것이 일반적입니다.

__del__()

객체 파괴를 준비합니다. `IOBase`는 인스턴스의 `close()` 메서드를 호출하는 이 메서드의 기본 구현을 제공합니다.

class io.RawIOBase

원시 바이너리 스트림의 베이스 클래스. `IOBase`를 상속합니다.

원시 바이너리 스트림은 일반적으로 하부 OS 장치나 API에 대한 저수준 액세스를 제공하며, 이것을 고수준 프리미티브로 캡슐화하려고 하지 않습니다 (이 기능은 이 페이지에서 나중에 설명할 버퍼링된 바이너리 스트림과 텍스트 스트림에서 고수준으로 수행됩니다).

`RawIOBase`는 `IOBase`에서 온 것 외에 이 메서드를 제공합니다:

read(*size=-1*, /)

객체에서 최대 *size* 바이트를 읽고 반환합니다. 편의상, *size*가 지정되지 않거나 -1이면, EOF까지의 모든 바이트가 반환됩니다. 그렇지 않으면, 하나의 시스템 호출만 수행됩니다. 운영 체제 시스템 호출이 *size* 바이트 미만을 반환하면 *size* 바이트 미만이 반환될 수 있습니다.

0바이트가 반환되고, *size*가 0이 아니면, 파일의 끝을 나타냅니다. 객체가 비 블로킹 모드이고 사용 가능한 바이트가 없으면 `None`이 반환됩니다.

기본 구현은 `readall()`과 `readinto()`로 위임합니다.

readall()

필요한 경우 스트림에 대한 다중 호출을 사용하여, EOF까지 스트림의 모든 바이트를 읽고 반환합니다.

readinto(*b*, /)

미리 할당되고, 쓰기 가능한 바이트열류 객체 *b*로 바이트를 읽고 읽은 바이트 수를 반환합니다. 예를 들어, *b*는 `bytearray`일 수 있습니다. 객체가 비 블로킹 모드이고 사용 가능한 바이트가 없으면, `None`이 반환됩니다.

write(*b*, /)

주어진 바이트열류 객체, *b*를 하부 원시 스트림에 쓰고, 쓴 바이트 수를 반환합니다. 하부 원시 스트림의 특성에 따라, 특히 비 블로킹 모드이면 바이트 단위로 *b*의 길이보다 짧을 수 있습니다. 원시 스트림이 블록 하지 않도록 설정되었고 단일 바이트를 당장 쓸 수 없으면 `None`이 반환됩니다. 호출자는 이 메서드가 반환된 후 *b*를 해제하거나 변경할 수 있어서, 구현은 메서드 호출 중에만 *b*에 액세스해야 합니다.

class io.BufferedIOBase

어떤 종류의 버퍼링을 지원하는 바이너리 스트림의 베이스 클래스. `IOBase`를 상속합니다.

The main difference with `RawIOBase` is that methods `read()`, `readinto()` and `write()` will try (respectively) to read as much input as requested or to emit all provided data.

In addition, if the underlying raw stream is in non-blocking mode, when the system returns would block `write()` will raise `BlockingIOError` with `BlockingIOError.characters_written` and `read()` will return data read so far or `None` if no data is available.

또한, `read()` 메서드에는 `readinto()`로 위임하는 기본 구현이 없습니다.

일반적인 `BufferedIOBase` 구현은 `RawIOBase` 구현에서 상속하지 말고, `BufferedWriter`와 `BufferedReader`처럼 감싸야 합니다.

`BufferedIOBase`는 `IOBase`에서 온 것 외에 다음 데이터 어트리뷰트와 메서드를 제공하거나 재정의합니다:

raw

`BufferedIOBase`가 다루는 하부 원시 스트림 (`RawIOBase` 인스턴스). 이것은 `BufferedIOBase` API의 일부가 아니며 일부 구현에는 없을 수 있습니다.

detach()

하부 원시 스트림을 버퍼에서 분리하고 반환합니다.

원시 스트림이 분리된 후에는, 버퍼가 사용할 수 없는 상태가 됩니다.

`BytesIO`와 같은 일부 버퍼에는 이 메서드가 반환할 단일 원시 스트림 개념이 없습니다. 그들은 `UnsupportedOperation`을 발생시킵니다.

Added in version 3.1.

read(*size=-1*, /)

Read and return up to *size* bytes. If the argument is omitted, `None`, or negative read as much as possible.

Fewer bytes may be returned than requested. An empty `bytes` object is returned if the stream is already at EOF. More than one read may be made and calls may be retried if specific errors are encountered, see `os.read()` and **PEP 475** for more details. Less than size bytes being returned does not imply that EOF is imminent.

When reading as much as possible the default implementation will use `raw.readall` if available (which should implement `RawIOBase.readall()`), otherwise will read in a loop until read returns `None`, an empty `bytes`, or a non-retryable error. For most streams this is to EOF, but for non-blocking streams more data may become available.

> **참고**
>
> When the underlying raw stream is non-blocking, implementations may either raise `BlockingIOError` or return `None` if no data is available. `io` implementations return `None`.

read1(*size=-1*, /)

Read and return up to *size* bytes, calling `readinto()` which may retry if `EINTR` is encountered per **PEP 475**. If *size* is `-1` or not provided, the implementation will choose an arbitrary value for *size*.

> **참고**
>
> When the underlying raw stream is non-blocking, implementations may either raise `BlockingIOError` or return `None` if no data is available. `io` implementations return `None`.

readinto(*b*, /)

미리 할당되고, 쓰기 가능한 바이트열류 객체 *b*로 바이트를 읽고 읽은 바이트 수를 반환합니다. 예를 들어, *b*는 `bytearray`일 수 있습니다.

`read()`와 마찬가지로, 하부 원시 스트림이 대화식이 아닌 한, 여러 읽기가 수행될 수 있습니다.

하부 원시 스트림이 비 블로킹 모드이고, 현재 사용 가능한 데이터가 없으면 `BlockingIOError`가 발생합니다.

readinto1(*b*, /)

하부 원시 스트림의 `read()` (또는 `readinto()`) 메서드를 최대 한 번만 호출하여, 미리 할당된 쓰기 가능한 바이트열류 객체 *b*로 바이트를 읽습니다. 읽은 바이트 수를 반환합니다.

하부 원시 스트림이 비 블로킹 모드이고, 현재 사용 가능한 데이터가 없으면 `BlockingIOError` 가 발생합니다.

Added in version 3.5.

write(*b*, /)

주어진 바이트열류 객체, *b*를 쓰고 기록된 바이트 수를 반환합니다 (쓰기에 실패하면 `OSError`가 발생하기 때문에 항상 바이트 단위로 *b* 길이와 같습니다). 실제 구현에 따라, 이러한 바이트는 하부 스트림에 쉽게 쓸 수 있거나, 성능과 지연 이유로 버퍼에 보관될 수 있습니다.

비 블로킹 모드에서, 데이터를 원시 스트림에 기록해야 하지만 원시 스트림이 블로킹 없이 모든 데이터를 받아들일 수 없으면, `BlockingIOError` 가 발생합니다.

호출자는 이 메서드가 반환된 후 *b*를 해제하거나 변경할 수 있어서, 구현은 메서드 호출 중에만 *b*에 액세스해야 합니다.

원시 파일 I/O

class io.**FileIO**(*name*, *mode='r'*, *closefd=True*, *opener=None*)

바이트열 데이터를 포함하는 OS 수준 파일을 나타내는 원시 바이너리 스트림. `RawIOBase`를 상속합니다.

*name*은 다음 두 가지 중 하나일 수 있습니다:

- 열릴 파일의 경로를 나타내는 문자열이나 `bytes` 객체. 이 경우 closefd는 `True`(기본값)이어야 합니다. 그렇지 않으면 에러가 발생합니다.
- 결과 `FileIO` 객체가 액세스할 기존 OS 수준 파일 기술자의 번호를 나타내는 정수. FileIO 객체가 닫힐 때, *closefd*가 `False`로 설정되어 있지 않은 한 이 fd도 닫힙니다.

*mode*는 읽기(기본값), 쓰기, 배타적 생성 또는 덧붙이기를 위해 `'r'`, `'w'`, `'x'` 또는 `'a'`일 수 있습니다. 쓰기나 덧붙이기로 열 때 파일이 존재하지 않으면 만들어집니다; 쓰기 위해 열면 파일이 잘립니다. 생성을 위해 열었을 때 파일이 이미 존재하면 `FileExistsError` 가 발생합니다. 생성하기 위해 파일을 여는 것은 쓰기를 의미하므로, 이 모드는 `'w'`와 유사한 방식으로 작동합니다. 읽기와 쓰기를 동시에 허락하려면 `'+'`를 모드에 추가하십시오.

이 클래스의 `read()` (양수 인자로 호출되었을 때), `readinto()` 및 `write()` 메서드는 하나의 시스템 호출만 수행합니다.

콜러블을 *opener*로 전달하여 사용자 정의 오프너를 사용할 수 있습니다. 그러면 파일 객체의 하부 파일 기술자는 (*name*, *flags*)로 *opener*를 호출하여 얻습니다. *opener*는 열린 파일 기술자를 반환해야 합니다 (`os.open`을 *opener*로 전달하면 `None`을 전달하는 것과 유사한 기능이 됩니다).

새로 만들어진 파일은 상속 불가능합니다.

opener 매개 변수 사용에 대한 예는 `open()` 내장 함수를 참조하십시오.

버전 3.3에서 변경: *opener* 매개 변수가 추가되었습니다. `'x'` 모드가 추가되었습니다.

버전 3.4에서 변경: 이제 파일이 상속 불가능합니다.

`FileIO`는 `RawIOBase`와 `IOBase`에서 온 것 외에 다음 데이터 어트리뷰트를 제공합니다:

mode

생성자에 제공된 모드.

name

파일 이름. 생성자에 이름이 지정되지 않으면 파일의 파일 기술자입니다.

버퍼링 된 스트림

버퍼링 된 I/O 스트림은 원시 I/O보다 I/O 장치에 대한 더 고수준의 인터페이스를 제공합니다.

class io.**BytesIO**(*initial_bytes=b''*)

인 메모리 바이트 버퍼를 사용하는 바이너리 스트림. `BufferedIOBase`를 상속합니다. `close()` 메서드가 호출될 때 버퍼가 폐기됩니다.

선택적 인자 *initial_bytes*는 초기 데이터를 포함하는 바이트열류 객체입니다.

`BytesIO`는 `BufferedIOBase`와 `IOBase`의 메서드 외에 다음 메서드를 제공하거나 재정의합니다:

getbuffer()

복사하지 않고 버퍼의 내용에 대한 읽을 수 있고 쓸 수 있는 뷰를 반환합니다. 또한, 뷰를 변경하면 버퍼의 내용이 투명하게 갱신됩니다:

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

> **참고**
>
> 뷰가 존재하는 한, `BytesIO` 객체의 크기를 조정하거나 닫을 수 없습니다.

Added in version 3.2.

getvalue()

버퍼의 전체 내용을 포함하는 `bytes`를 반환합니다.

read1(*size=-1, /*)

`BytesIO`에서, 이것은 `read()`와 같습니다.

버전 3.7에서 변경: *size* 인자는 이제 선택적입니다.

readinto1(*b, /*)

`BytesIO`에서, 이것은 `readinto()`와 같습니다.

Added in version 3.5.

class io.**BufferedReader**(*raw, buffer_size=DEFAULT_BUFFER_SIZE*)

읽을 수 있고 탐색할 수 없는(non seekable) `RawIOBase` 원시 바이너리 스트림에 대한 고수준의 액세스를 제공하는 버퍼링 된 바이너리 스트림. `BufferedIOBase`를 상속합니다.

이 객체에서 데이터를 읽을 때, 하부 원시 스트림에서 더 많은 양의 데이터가 요청되어, 내부 버퍼에 보관될 수 있습니다. 버퍼링 된 데이터는 후속 읽기에서 직접 반환될 수 있습니다.

생성자는 주어진 읽을 수 있는 *raw* 스트림과 *buffer_size*에 대해 `BufferedReader`를 만듭니다. *buffer_size*를 생략하면, `DEFAULT_BUFFER_SIZE`가 사용됩니다.

`BufferedReader`는 `BufferedIOBase`와 `IOBase`의 메서드 외에 다음 메서드를 제공하거나 재정의합니다:

peek(*size=0, /*)

Return bytes from the stream without advancing the position. The number of bytes returned may be less or more than requested. If the underlying raw stream is non-blocking and the operation would block, returns empty bytes.

read(*size=-1, /*)

In `BufferedReader` this is the same as `io.BufferedIOBase.read()`

read1(*size=-1, /*)

In `BufferedReader` this is the same as `io.BufferedIOBase.read1()`

버전 3.7에서 변경: *size* 인자는 이제 선택적입니다.

class io.BufferedWriter(*raw, buffer_size=DEFAULT_BUFFER_SIZE*)

쓸 수 있고 탐색할 수 없는(non seekable) `RawIOBase` 원시 바이너리 스트림에 대한 고수준 액세스를 제공하는 버퍼링 된 바이너리 스트림. `BufferedIOBase`를 상속합니다.

이 객체에 쓸 때, 데이터는 일반적으로 내부 버퍼에 배치됩니다. 버퍼는 다음과 같은 다양한 조건에서 하부 `RawIOBase` 객체에 기록됩니다:

- 계류 중인 모든 데이터에 비해 버퍼가 너무 작아질 때;
- `flush()`가 호출될 때;
- `seek()`이 요청될 때 (`BufferedRandom` 객체의 경우);
- `BufferedWriter` 객체가 닫히거나 파괴될 때.

생성자는 주어진 쓰기 가능한 *raw* 스트림에 대해 `BufferedWriter`를 만듭니다. *buffer_size*가 제공되지 않으면, 기본값은 `DEFAULT_BUFFER_SIZE`입니다.

`BufferedWriter`는 `BufferedIOBase`와 `IOBase`의 메서드 외에 다음 메서드를 제공하거나 재정의합니다:

flush()

버퍼에 있는 바이트를 원시 스트림으로 강제 출력합니다. 원시 스트림이 블록되면 `BlockingIOError`를 발생시켜야 합니다.

write(*b, /*)

Write the *bytes-like object*, *b*, and return the number of bytes written. When in non-blocking mode, a `BlockingIOError` with `BlockingIOError.characters_written` set is raised if the buffer needs to be written out but the raw stream blocks.

class io.BufferedRandom(*raw, buffer_size=DEFAULT_BUFFER_SIZE*)

탐색할 수 있는 (seekable) `RawIOBase` 원시 바이너리 스트림에 고수준의 액세스를 제공하는 버퍼링 된 바이너리 스트림. `BufferedReader`와 `BufferedWriter`를 상속합니다.

생성자는 첫 번째 인자로 주어진 탐색 가능한 원시 스트림에 대한 판독기(reader)와 기록기(writer)를 만듭니다. *buffer_size*를 생략하면 기본값은 `DEFAULT_BUFFER_SIZE`입니다.

`BufferedRandom`은 `BufferedReader`나 `BufferedWriter`가 수행 할 수 있는 모든 작업을 수행할 수 있습니다. 또한, `seek()`과 `tell()`이 구현되도록 보장됩니다.

class io.BufferedRWPair(*reader, writer, buffer_size=DEFAULT_BUFFER_SIZE, /*)

하나는 읽을 수 있고, 다른 하나는 쓸 수 있는, 두 개의 탐색할 수 없는 (non seekable) `RawIOBase` 원시 바이너리 스트림에 대한 고수준 액세스를 제공하는 버퍼링 된 바이너리 스트림. `BufferedIOBase`를 상속합니다.

*reader*와 *writer*는 각각 읽고 쓸 수 있는 `RawIOBase` 객체입니다. *buffer_size*가 생략되면 기본값은 `DEFAULT_BUFFER_SIZE`입니다.

`BufferedRWPair`는 `UnsupportedOperation`을 발생시키는 `detach()`를 제외한 모든 `BufferedIOBase`의 메서드를 구현합니다.

> **경고**
>
> `BufferedRWPair`는 하부 원시 스트림에 대한 동기화된 액세스를 시도하지 않습니다. reader와 writer로 같은 객체를 전달해서는 안 됩니다; 대신 `BufferedRandom`을 사용하십시오.

텍스트 I/O

class io.**TextIOBase**

텍스트 스트림의 베이스 클래스. 이 클래스는 스트림 I/O를 위한 문자와 줄 기반 인터페이스를 제공합니다. *IOBase*를 상속합니다.

*TextIOBase*는 *IOBase*에서 온 것 외에 다음 데이터 어트리뷰트와 메서드를 제공하거나 재정의합니다:

encoding

스트림의 바이트열을 문자열로 디코딩하고, 문자열을 바이트열로 인코딩하는 데 사용되는 인코딩의 이름.

errors

디코더나 인코더의 에러 설정.

newlines

지금까지 번역된 줄 넘김을 나타내는, 문자열, 문자열 튜플 또는 None. 구현과 초기 생성자 플래그에 따라, 사용하지 못할 수 있습니다.

buffer

The underlying binary buffer (a *BufferedIOBase* or *RawIOBase* instance) that *TextIOBase* deals with. This is not part of the *TextIOBase* API and may not exist in some implementations.

detach()

하부 바이너리 버퍼를 *TextIOBase*와 분리하여 반환합니다.

하부 버퍼가 분리된 후에는, *TextIOBase*는 사용할 수 없는 상태가 됩니다.

*StringIO*와 같은 일부 *TextIOBase* 구현에는 하부 버퍼 개념이 없을 수 있으며 이 메서드를 호출하면 *UnsupportedOperation*이 발생합니다.

Added in version 3.1.

read(*size=-1*, /)

스트림에서 최대 *size* 문자를 단일 *str*로 읽고 반환합니다. *size*가 음수이거나 None이면 EOF까지 읽습니다.

readline(*size=-1*, /)

줄 넘김이나 EOF까지 읽고 단일 *str*을 반환합니다. 스트림이 이미 EOF에 있으면, 빈 문자열이 반환됩니다.

*size*가 지정되면, 최대 *size* 문자를 읽습니다.

seek(*offset*, *whence=SEEK_SET*, /)

스트림 위치를 지정된 *offset*으로 변경합니다. 동작은 *whence* 매개 변수에 따라 다릅니다. *whence*의 기본값은 SEEK_SET입니다.

- SEEK_SET이나 0: 스트림의 시작부터 탐색합니다 (기본값); *offset*은 *TextIOBase.tell()*이 반환한 숫자이거나 0이어야 합니다. 다른 *offset* 값은 정의되지 않은 동작을 생성합니다.
- SEEK_CUR이나 1: 현재 위치로 "seek" 합니다; *offset*은 0이어야 하며, 이는 아무런 일도 하지 않습니다 (다른 모든 값은 지원되지 않습니다).
- SEEK_END나 2: 스트림의 끝으로 seek 합니다; *offset*은 0이어야 합니다 (다른 모든 값은 지원되지 않습니다).

새로운 절대 위치를 불투명한 숫자로 반환합니다.

Added in version 3.1: SEEK_* 상수.

tell()

현재 스트림 위치를 불투명한 숫자로 반환합니다. 숫자는 일반적으로 하부 바이너리 저장소의 바이트 수를 나타내지 않습니다.

write(*s*, /)

문자열 *s*를 스트림에 쓰고 쓴 문자 수를 반환합니다.

class io.**TextIOWrapper**(*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, *write_through=False*)

BufferedIOBase 버퍼링 된 바이너리 스트림에 대한 고수준의 액세스를 제공하는 버퍼링 된 텍스트 스트림. *TextIOBase*를 상속합니다.

encoding gives the name of the encoding that the stream will be decoded or encoded with. In *UTF-8 Mode*, this defaults to UTF-8. Otherwise, it defaults to *locale.getencoding()*. `encoding="locale"` can be used to specify the current locale' s encoding explicitly. See 텍스트 인코딩 for more information.

*errors*는 인코딩과 디코딩 에러 처리 방법을 지정하는 선택적 문자열입니다. 인코딩 에러가 있을 때 *ValueError* 예외를 발생시키려면 `'strict'`를 전달하고 (기본값 `None`은 같은 효과를 줍니다), 에러를 무시하려면 `'ignore'`를 전달하십시오. (인코딩 에러를 무시하면 데이터가 손실될 수 있음에 유의하십시오.) `'replace'`는 잘못된 데이터가 있는 곳에 대체 마커(가령 `'?'`)가 삽입되도록 합니다. `'backslashreplace'`는 잘못된 데이터를 역 슬래시 이스케이프 시퀀스로 대체합니다. 기록할 때, `'xmlcharrefreplace'`(적절한 XML 문자 참조로 대체합니다)나 `'namereplace'`(`\N{...}` 이스케이프 시퀀스로 대체합니다)를 사용할 수 있습니다. *codecs.register_error()*로 등록된 다른 에러 처리 이름도 유효합니다.

*newline*은 줄 끝 처리 방법을 제어합니다. `None`, `''`, `'\n'`, `'\r'` 및 `'\r\n'` 일 수 있습니다. 다음과 같이 작동합니다:

- 스트림에서 입력을 읽을 때, *newline*이 `None`이면, 유니버설 줄 넘김 모드가 활성화됩니다. 입력의 줄은 `'\n'`, `'\r'` 또는 `'\r\n'`으로 끝날 수 있으며, 호출자에게 반환되기 전에 `'\n'`으로 변환됩니다. *newline*이 `''`이면, 유니버설 줄 넘김 모드가 활성화되지만, 줄 끝은 변환되지 않은 상태로 호출자에게 반환됩니다. *newline*이 다른 유효한 값이면, 입력 줄은 주어진 문자열로만 끝나고, 줄 끝은 변환되지 않은 상태로 호출자에게 반환됩니다.
- 스트림에 출력을 기록할 때, *newline*이 `None`이면, 기록되는 모든 `'\n'` 문자는 시스템 기본 줄 구분자 *os.linesep*으로 변환됩니다. *newline*이 `''`이나 `'\n'`이면, 변환이 수행되지 않습니다. *newline*이 다른 유효한 값이면, 기록되는 모든 `'\n'` 문자는 주어진 문자열로 변환됩니다.

*line_buffering*이 `True`이면, write 호출에 줄 넘김 문자나 캐리지 리턴이 포함되어 있으면 *flush()*가 암시됩니다.

*write_through*가 `True`이면, *write()*에 대한 호출은 버퍼링 되지 않음이 보장됩니다: *TextIOWrapper* 객체에 기록된 모든 데이터는 즉시 하부 바이너리 *buffer*로 처리됩니다.

버전 3.3에서 변경: *write_through* 인자가 추가되었습니다.

버전 3.3에서 변경: 기본 *encoding*은 이제 `locale.getpreferredencoding()` 대신 `locale.getpreferredencoding(False)` 입니다. *locale.setlocale()*을 사용하여 임시 로케일 인코딩을 변경하지 않고, 사용자가 선호하는 인코딩 대신 현재 로케일 인코딩을 사용합니다.

버전 3.10에서 변경: The *encoding* argument now supports the `"locale"` dummy encoding name.

> **참고**
>
> When the underlying raw stream is non-blocking, a *BlockingIOError* may be raised if a read operation cannot be completed immediately.

*TextIOWrapper*는 *TextIOBase*와 *IOBase*에서 온 것 외에 다음 데이터 어트리뷰트와 메서드를 제공합니다:

line_buffering

줄 버퍼링이 활성화되었는지 여부.

write_through

쓰기가 하부 바이너리 버퍼로 즉시 전달되는지 여부.

Added in version 3.7.

reconfigure(*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=None*, *write_through=None*)

encoding, *errors*, *newline*, *line_buffering* 및 *write_through*에 대한 새로운 설정을 사용하여 이 텍스트 스트림을 재구성합니다.

*encoding*이 지정되었지만, *errors*가 지정되지 않았을 때 `errors='strict'`가 사용되는 것을 제외하고, 지정되지 않은 매개 변수는 현재 설정을 유지합니다.

스트림에서 일부 데이터를 이미 읽었다면 encoding이나 newline을 변경할 수 없습니다. 반면에, 기록 후의 encoding 변경은 가능합니다.

이 메서드는 새 매개 변수를 설정하기 전에 묵시적 스트림 플러시를 수행합니다.

Added in version 3.7.

버전 3.11에서 변경: The method supports `encoding="locale"` option.

seek(*cookie*, *whence=os.SEEK_SET*, /)

Set the stream position. Return the new stream position as an `int`.

Four operations are supported, given by the following argument combinations:

- `seek(0, SEEK_SET)`: Rewind to the start of the stream.
- `seek(cookie, SEEK_SET)`: Restore a previous position; *cookie* **must be** a number returned by `tell()`.
- `seek(0, SEEK_END)`: Fast-forward to the end of the stream.
- `seek(0, SEEK_CUR)`: 현재의 스트림 위치를 변경하지 않습니다.

Any other argument combinations are invalid, and may raise exceptions.

더 보기

`os.SEEK_SET`, `os.SEEK_CUR`, and `os.SEEK_END`.

tell()

Return the stream position as an opaque number. The return value of `tell()` can be given as input to `seek()`, to restore a previous stream position.

class io.**StringIO**(*initial_value=''*, *newline='\n'*)

인 메모리 텍스트 버퍼를 사용하는 텍스트 스트림. `TextIOBase`를 상속합니다.

`close()` 메서드가 호출될 때 텍스트 버퍼가 폐기됩니다.

버퍼의 초깃값은 *initial_value*를 제공하여 설정할 수 있습니다. 줄 바꿈 변환이 활성화되면, 줄 바꿈은 `write()` 한 것처럼 인코딩됩니다. 스트림은 버퍼의 시작 부분에 위치하며, 이는 기존 파일을 `w+` 모드로 여는 것을 흉내내어 시작부터 즉시 쓰기나 초기 값을 덮어쓰는 쓰기에 준비되도록 합니다. `a+` 모드로 파일을 여는 것을 흉내내려면, `f.seek(0, io.SEEK_END)`을 사용하여 스트림을 버퍼의 끝으로 재위치 시키세요.

출력을 스트림에 쓸 때, *newline*이 `None`이면, 모든 플랫폼에서 줄 넘김이 `\n`로 기록된다는 점을 제외하고는, *newline* 인자는 `TextIOWrapper`에서 처럼 작동합니다.

`StringIO`는 `TextIOBase`와 `IOBase`의 메서드 외에 이 메서드를 제공합니다:

getvalue()

버퍼의 전체 내용을 포함하는 `str`을 반환합니다. 스트림 위치는 변경되지 않지만, 줄 넘김은 `read()`와 같이 디코딩됩니다.

사용 예:

```
import io

output = io.StringIO()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
output.write('First line.\n')
print('Second line.', file=output)

# 파일 내용을 꺼냅니다 -- 'First line.\nSecond line.\n' 이 됩니다.
contents = output.getvalue()

# 객체를 닫고 메모리 버퍼를 폐기합니다 --
# .getvalue() 는 이제 예외를 발생시킵니다.
output.close()
```

class io.IncrementalNewlineDecoder

유니버설 줄 넘김 모드의 줄 넘김 디코딩을 하는 도우미 코덱. *codecs.IncrementalDecoder*를 상속합니다.

16.2.5 Static Typing

The following protocols can be used for annotating function and method arguments for simple stream reading or writing operations. They are decorated with *@typing.runtime_checkable*.

class io.Reader[*T*]()

Generic protocol for reading from a file or other input stream. `T` will usually be *str* or *bytes*, but can be any type that is read from the stream.

Added in version 3.14.

read()

read(*size*, /)

Read data from the input stream and return it. If *size* is specified, it should be an integer, and at most *size* items (bytes/characters) will be read.

For example:

```
def read_it(reader: Reader[str]):
    data = reader.read(11)
    assert isinstance(data, str)
```

class io.Writer[*T*]()

Generic protocol for writing to a file or other output stream. `T` will usually be *str* or *bytes*, but can be any type that can be written to the stream.

Added in version 3.14.

write(*data*, /)

Write *data* to the output stream and return the number of items (bytes/characters) written.

For example:

```
def write_binary(writer: Writer[bytes]):
    writer.write(b"Hello world!\n")
```

See *ABCs and Protocols for working with I/O* for other I/O related protocols and classes that can be used for static type checking.

16.2.6 성능

이 섹션에서는 제공된 구상 I/O 구현의 성능에 대해 논합니다.

바이너리 I/O

사용자가 단일 바이트를 요청할 때조차 큰 데이터 청크만 읽고 써서, 버퍼링 된 I/O는 운영 체제의 버퍼링 되지 않은 I/O 루틴을 호출하고 실행할 때의 비효율성을 숨깁니다. 이득은 OS 와 수행되는 I/O 종류에 따라 다릅니다. 예를 들어, 리눅스와 같은 일부 최신 OS에서는, 버퍼링 되지 않은 디스크 I/O는 버퍼링 된 I/O만큼 빠를 수 있습니다. 그러나 요점은 버퍼링 된 I/O가 플랫폼과 배경 장치와 관계없이 예측 가능한 성능을 제공한다는 것입니다. 따라서, 바이너리 데이터에는 버퍼링 되지 않은 I/O보다는 버퍼링 된 I/O를 사용하는 것이 거의 항상 바람직합니다.

텍스트 I/O

바이너리 저장소(가령 파일)에 대한 텍스트 I/O는 같은 저장소에 대한 바이너리 I/O보다 상당히 느린데, 문자 코덱을 사용하여 유니코드와 바이너리 데이터 간의 변환이 필요하기 때문입니다. 이것은 큰 로그 파일과 같은 많은 양의 텍스트 데이터를 처리할 때 드러날 수 있습니다. 또한, `tell()`과 `seek()`은 사용된 재구성 알고리즘으로 인해 상당히 느립니다.

그러나, `StringIO`는 네이티브 인 메모리 유니코드 컨테이너이며 `BytesIO`와 비슷한 속도를 나타냅니다.

다중 스레드

`FileIO` 객체는 감싼 운영 체제 호출(가령 유닉스의 `read(2)`)이 스레드 안전하다면 스레드 안전합니다.

바이너리 버퍼링 된 객체(`BufferedReader`, `BufferedWriter`, `BufferedRandom` 및 `BufferedRWPair` 의 인스턴스)는 록을 사용하여 내부 구조를 보호합니다; 따라서 여러 스레드에서 동시에 호출하는 것이 안전합니다.

`TextIOWrapper` 객체는 스레드 안전하지 않습니다.

재진입

바이너리 버퍼링 된 객체(`BufferedReader`, `BufferedWriter`, `BufferedRandom` 및 `BufferedRWPair` 의 인스턴스)는 재진입할 수 없습니다. 재진입 호출이 정상적인 상황에서는 발생하지 않지만, `signal` 처리기에서 I/O를 수행한다면 발생할 수 있습니다. 스레드가 이미 액세스 중인 버퍼링 된 객체에 다시 진입하려고 하면, `RuntimeError`가 발생합니다. 이것이 다른 스레드가 버퍼링 된 객체에 진입하는 것을 막는 것은 아님에 유의하십시오.

`open()` 함수는 버퍼링 된 객체를 `TextIOWrapper` 안에 감싸기 때문에, 위의 내용은 텍스트 파일에도 묵시적으로 확장됩니다. 여기에는 표준 스트림이 포함되므로 내장 `print()` 함수에도 영향을 줍니다.

16.3 `time` — 시간 액세스와 변환

이 모듈은 다양한 시간 관련 함수를 제공합니다. 관련 기능에 대해서는, `datetime`과 `calendar` 모듈도 참조하십시오.

이 모듈을 항상 사용할 수 있지만, 모든 플랫폼에서 모든 함수를 사용할 수 있는 것은 아닙니다. 이 모듈에 정의된 대부분의 함수는 같은 이름의 플랫폼 C 라이브러리 함수를 호출합니다. 이러한 함수의 의미는 플랫폼마다 달라서, 플랫폼 설명서를 참조하면 도움이 될 수 있습니다.

일부 용어와 관례에 대한 설명을 순서대로 제시합니다.

- 에포크(*epoch*)는 시간이 시작되는 시점이며, `time.gmtime(0)`의 반환값입니다. 모든 플랫폼에서 1970년 1월 1일, 00:00:00(UTC) 입니다.
- 용어 에포크 이후의 초(*seconds since the epoch*)는 에포크 이후로 총 지나간 초를 나타내는데, 보통 윤초는 제외합니다. 모든 POSIX 호환 플랫폼에서 윤초는 이 총계에서 제외됩니다.
- 이 모듈의 함수는 에포크 전이나 먼 미래의 날짜와 시간을 처리하지 못할 수 있습니다. 미래의 컷오프 지점은 C 라이브러리에 의해 결정됩니다; 32비트 시스템의 경우, 일반적으로 2038년입니다.
- 함수 `strptime()`은 `%y` 포맷 코드가 제공될 때 2자리 연도를 구문 분석할 수 있습니다. 2자리 연도를 구문 분석할 때, POSIX와 ISO C 표준에 따라 변환됩니다: 값 69–99는 1969–1999에, 값 0–68은 2000–2068에 매핑됩니다.

- UTC is Coordinated Universal Time and superseded Greenwich Mean Time or GMT as the basis of international timekeeping. The acronym UTC is not a mistake but conforms to an earlier, language-agnostic naming scheme for time standards such as UT0, UT1, and UT2.
- DST는 일광 절약 시간(Daylight Saving Time)인데, 일 년 중 일부 기간 시간대를 (일반적으로) 한 시간 조정합니다. DST 규칙은 매직(현지 법에 따라 결정됩니다)이며 해가 바뀜에 따라 변경될 수 있습니다. C 라이브러리에는 현지 규칙이 포함된 테이블이 있으며 (종종 유연성을 위해 시스템 파일에서 읽습니다) 이 점에서 진정한 지혜(True Wisdom)의 유일한 원천입니다.
- 다양한 실시간 함수의 정밀도는 값이나 인자가 표현되는 단위가 제안하는 것보다 못할 수 있습니다. 예를 들어 대부분의 유닉스 시스템에서 시계는 초당 50회나 100회만 "틱(ticks)" 합니다.
- 반면에, `time()`과 `sleep()`의 정밀도는 그들의 유닉스의 해당하는 것보다 낫습니다: 시간은 부동 소수점 숫자로 표현되고, `time()`은 사용 가능한 가장 정확한 시간을 반환하며 (사용할 수 있으면 유닉스 `gettimeofday()`를 사용합니다), `sleep()`은 0이 아닌 소수부를 갖는 시간을 받아들입니다 (사용할 수 있으면, 이것을 구현하는 데 유닉스 `select()`를 사용합니다).
- `gmtime()`, `localtime()` 및 `strptime()`에 의해 반환되고 `asctime()`, `mktime()` 및 `strftime()`이 받아들이는 시간 값은 9개의 정수의 시퀀스입니다. `gmtime()`, `localtime()` 및 `strptime()`의 반환 값은 개별 필드에 대한 어트리뷰트 이름도 제공합니다.

  이러한 객체에 대한 설명은 `struct_time`을 참조하십시오.

  버전 3.3에서 변경: 플랫폼이 해당 `struct tm` 멤버를 지원할 때 `tm_gmtoff`와 `tm_zone` 어트리뷰트를 제공하도록 `struct_time` 형이 확장되었습니다.

  버전 3.6에서 변경: `struct_time` 어트리뷰트 `tm_gmtoff`와 `tm_zone`은 이제 모든 플랫폼에서 사용 가능합니다.
- 시간 표현 간에 변환하려면 다음 함수를 사용하십시오:

변환 전	변환 후	변환 함수
에포크 이후 초	UTC의 `struct_time`	`gmtime()`
에포크 이후 초	현지 시간의 `struct_time`	`localtime()`
UTC의 `struct_time`	에포크 이후 초	`calendar.timegm()`
현지 시간의 `struct_time`	에포크 이후 초	`mktime()`

16.3.1 함수

`time.`**`asctime`**`(`[*t*]`)`

`gmtime()`이나 `localtime()` 이 반환한 시간을 나타내는 튜플이나 `struct_time`을 `'Sun Jun 20 23:21:05 1993'` 형식의 문자열로 변환합니다. 날짜(day) 필드는 두 문자 길이이며 날짜가 한자리이면 스페이스로 채워집니다, 예를 들어: `'Wed Jun 9 04:26:40 1993'`.

*t*가 제공되지 않으면, `localtime()` 에서 반환된 현재 시각이 사용됩니다. 로케일 정보는 `asctime()`에서 사용되지 않습니다.

> **참고**
>
> 같은 이름의 C 함수와 달리, `asctime()`은 끝에 줄 바꿈을 추가하지 않습니다.

`time.`**`pthread_getcpuclockid`**`(`*thread_id*`)`

지정된 *thread_id*에 대한 스레드 특정 CPU-시간 시계의 *clk_id*를 반환합니다.

*thread_id*에 적합한 값을 얻으려면 `threading.get_ident()` 나 `threading.Thread` 객체의 `ident` 어트리뷰트를 사용하십시오.

> **경고**
>
> 유효하지 않거나 만료된 *thread_id*를 전달하면 정의되지 않은 동작이 발생할 수 있습니다, 가령 세그먼트 폴트(segmentation fault).

가용성: Unix

자세한 내용은 `pthread_getcpuclockid(3)` 매뉴얼 페이지를 참조하십시오.

Added in version 3.7.

`time.`**`clock_getres`**(*clk_id*)

지정된 시계 *clk_id*의 해상도(정밀도)를 반환합니다. *clk_id*에 허용되는 값 리스트는 시계 *ID* 상수를 참조하십시오.

가용성: Unix.

Added in version 3.3.

`time.`**`clock_gettime`**(*clk_id*) → *float*

지정된 시계 *clk_id*의 시간을 반환합니다. *clk_id*에 허용되는 값 리스트는 시계 *ID* 상수를 참조하십시오.

Use `clock_gettime_ns()` to avoid the precision loss caused by the `float` type.

가용성: Unix.

Added in version 3.3.

`time.`**`clock_gettime_ns`**(*clk_id*) → *int*

`clock_gettime()`과 비슷하지만, 시간을 나노초로 반환합니다.

가용성: Unix.

Added in version 3.7.

`time.`**`clock_settime`**(*clk_id*, *time:* float)

지정된 시계 *clk_id*의 시간을 설정합니다. 현재, `CLOCK_REALTIME`이 *clk_id*에 대해 유일하게 허용되는 값입니다.

Use `clock_settime_ns()` to avoid the precision loss caused by the `float` type.

가용성: Unix, not Android, not iOS.

Added in version 3.3.

`time.`**`clock_settime_ns`**(*clk_id*, *time:* int)

`clock_settime()`과 비슷하지만, 나노초로 시간을 설정합니다.

가용성: Unix, not Android, not iOS.

Added in version 3.7.

`time.`**`ctime`**([*secs*])

에포크 이후 초로 표현된 시간을 현지 시간을 나타내는 `'Sun Jun 20 23:21:05 1993'` 형식의 문자열로 변환합니다. 날짜(day) 필드는 두 문자 길이이며 날짜가 한자리이면 스페이스로 채워집니다, 예를 들어: `'Wed Jun 9 04:26:40 1993'`.

*secs*가 제공되지 않거나 `None`이면, `time()`이 반환하는 현재 시각이 사용됩니다. `ctime(secs)`는 `asctime(localtime(secs))`와 동등합니다. 로케일 정보는 `ctime()`에서 사용되지 않습니다.

`time.`**`get_clock_info`**(*name*)

지정된 시계에 대한 정보를 이름 공간 객체로 가져옵니다. 지원되는 시계 이름과 그 값을 읽는 해당 함수는 다음과 같습니다:

- `'monotonic'`: `time.monotonic()`

- 'perf_counter': *time.perf_counter()*
- 'process_time': *time.process_time()*
- 'thread_time': *time.thread_time()*
- 'time': *time.time()*

결과는 다음과 같은 어트리뷰트를 갖습니다:

- *adjustable*: 시계가 자동으로 (예를 들어 NTP 데몬에 의해) 또는 시스템 관리자에 의해 수동으로 변경될 수 있으면 `True`, 그렇지 않으면 `False`
- *implementation*: 시계값을 얻는 데 사용되는 하부 C 함수의 이름. 가능한 값은 시계 *ID* 상수를 참조하십시오.
- *monotonic*: 시계가 뒤로 이동할 수 없으면 `True`, 그렇지 않으면 `False`
- *resolution*: 초 단위의 시계 해상도 (*float*)

Added in version 3.3.

`time.`**`gmtime`**`(`[*secs*]`)`

에포크 이후의 초 단위 시간을 dst 플래그가 항상 0인 UTC인 *struct_time*으로 변환합니다. *secs*가 제공되지 않거나 *None*이면, *time()*에서 반환된 현재 시각이 사용됩니다. 초의 소수부는 무시됩니다. *struct_time* 객체에 대한 설명은 위를 참조하십시오. 이 함수의 역에 대해서는 *calendar.timegm()*을 참조하십시오.

`time.`**`localtime`**`(`[*secs*]`)`

*gmtime()*과 같지만, 현지 시간으로 변환합니다. *secs*가 제공되지 않거나 *None*이면 *time()*에서 반환된 현재 시각이 사용됩니다. DST가 주어진 시간에 적용되면 dst 플래그는 `1`로 설정됩니다.

localtime() may raise *OverflowError*, if the timestamp is outside the range of values supported by the platform C `localtime()` or `gmtime()` functions, and *OSError* on `localtime()` or `gmtime()` failure. It' s common for this to be restricted to years between 1970 and 2038.

`time.`**`mktime`**`(`*t*`)`

이것은 *localtime()* 의 역함수입니다. 인자는 UTC가 아니라 현지 시간으로 시간을 표현하는 *struct_time*이나 전체 9-튜플 (dst 플래그가 필요하기 때문에; 알 수 없으면 dst 플래그로 `-1`을 사용하십시오)입니다. *time()*과의 호환성을 위해, 부동 소수점 숫자를 반환합니다. 입력값을 유효한 시간으로 표현할 수 없으면, *OverflowError*나 *ValueError*가 발생합니다 (유효하지 않은 값이 파이썬이나 하부 C 라이브러리 중 어디에서 잡히는지에 따라 다릅니다). 시간을 생성 할 수 있는 가장 이른 날짜는 플랫폼에 따라 다릅니다.

`time.`**`monotonic`**`()` → *float*

단조(monotonic) 시계, 즉 뒤로 갈 수 없는 시계의 값을 (소수부가 있는 초로) 반환합니다. 시계는 시스템 시계 갱신의 영향을 받지 않습니다. 반환된 값의 기준점은 정의되어 있지 않아서, 두 호출 결과 간의 차이만 유효합니다.

Clock:

- On Windows, call `QueryPerformanceCounter()` and `QueryPerformanceFrequency()`.
- On macOS, call `mach_absolute_time()` and `mach_timebase_info()`.
- On HP-UX, call `gethrtime()`.
- Call `clock_gettime(CLOCK_HIGHRES)` if available.
- Otherwise, call `clock_gettime(CLOCK_MONOTONIC)`.

Use *monotonic_ns()* to avoid the precision loss caused by the *float* type.

Added in version 3.3.

버전 3.5에서 변경: The function is now always available and the clock is now the same for all processes.

버전 3.10에서 변경: On macOS, the clock is now the same for all processes.

time.**monotonic_ns**() → *int*

*monotonic()*과 비슷하지만, 시간을 나노초로 반환합니다.

Added in version 3.7.

time.**perf_counter**() → *float*

Return the value (in fractional seconds) of a performance counter, i.e. a clock with the highest available resolution to measure a short duration. It does include time elapsed during sleep. The clock is the same for all processes. The reference point of the returned value is undefined, so that only the difference between the results of two calls is valid.

CPython 구현 상세: On CPython, use the same clock as *time.monotonic()* and is a monotonic clock, i.e. a clock that cannot go backwards.

Use *perf_counter_ns()* to avoid the precision loss caused by the *float* type.

Added in version 3.3.

버전 3.10에서 변경: On Windows, the clock is now the same for all processes.

버전 3.13에서 변경: *time.monotonic()* 과 같은 시계를 사용합니다.

time.**perf_counter_ns**() → *int*

*perf_counter()*와 비슷하지만, 시간을 나노초로 반환합니다.

Added in version 3.7.

time.**process_time**() → *float*

현재 프로세스의 시스템과 사용자 CPU 시간 합계의 값을 (소수부가 있는 초로) 반환합니다. 수면 중 경과 시간은 포함되지 않습니다. 정의상 프로세스 수준입니다. 반환된 값의 기준점은 정의되어 있지 않아서, 두 호출 결과 간의 차이만 유효합니다.

Use *process_time_ns()* to avoid the precision loss caused by the *float* type.

Added in version 3.3.

time.**process_time_ns**() → *int*

*process_time()*과 비슷하지만, 시간을 나노초로 반환합니다.

Added in version 3.7.

time.**sleep**(*secs*)

Suspend execution of the calling thread for the given number of seconds. The argument may be a floating-point number to indicate a more precise sleep time.

If the sleep is interrupted by a signal and no exception is raised by the signal handler, the sleep is restarted with a recomputed timeout.

The suspension time may be longer than requested by an arbitrary amount, because of the scheduling of other activity in the system.

윈도우 구현

On Windows, if *secs* is zero, the thread relinquishes the remainder of its time slice to any other thread that is ready to run. If there are no other threads ready to run, the function returns immediately, and the thread continues execution. On Windows 8.1 and newer the implementation uses a high-resolution timer which provides resolution of 100 nanoseconds. If *secs* is zero, `Sleep(0)` is used.

유닉스 구현

- Use `clock_nanosleep()` if available (resolution: 1 nanosecond);
- Or use `nanosleep()` if available (resolution: 1 nanosecond);
- Or use `select()` (resolution: 1 microsecond).

> **참고**
>
> To emulate a "no-op", use `pass` instead of `time.sleep(0)`.
>
> To voluntarily relinquish the CPU, specify a real-time *scheduling policy* and use `os.sched_yield()` instead.

Raises an *auditing event* `time.sleep` with argument `secs`.

버전 3.5에서 변경: 시그널 처리기가 예외를 발생시키는 경우를 제외하고, 시그널에 의해 휴면이 중단되더라도 이 함수는 이제 최소한 *secs* 동안 휴면합니다 (근거는 **PEP 475**를 참조하십시오).

버전 3.11에서 변경: On Unix, the `clock_nanosleep()` and `nanosleep()` functions are now used if available. On Windows, a waitable timer is now used.

버전 3.13에서 변경: Raises an auditing event.

`time.`**`strftime`**(*format*[, *t*])

`gmtime()`이나 `localtime()` 에 의해 반환된 시간을 나타내는 튜플이나 `struct_time`을 *format* 인자로 지정된 문자열로 변환합니다. *t*가 제공되지 않으면, `localtime()` 에서 반환된 현재 시각이 사용됩니다. *format*은 문자열이어야 합니다. *t*의 필드 중 어느 것이라도 허용 범위를 벗어나면 `ValueError`가 발생합니다.

0은 시간 튜플의 어느 위치에 대해서도 유효한 인자입니다; 그것이 일반적으로 유효하지 않으면 값이 올바른 값으로 강제 변환됩니다.

format 문자열은 다음 지시자(directives)를 포함할 수 있습니다. 선택적 필드 너비와 정밀도 명세 없이 표시되며, `strftime()` 결과에서 표시된 문자로 대체됩니다:

지시자	의미	노트
%a	로케일의 약식 요일 이름.	
%A	로케일의 전체 요일 이름.	
%b	로케일의 약식 월 이름.	
%B	로케일의 전체 월 이름.	
%c	로케일의 적절한 날짜와 시간 표현.	
%d	월중 일(day of the month)을 십진수로 [01,31].	
%f	**마이크로 초를 십진수로** [000000,999999].	(1)
%H	시 (24 시간제) 를 십진수로 [00,23].	
%I	시 (12 시간제) 를 십진수로 [01,12].	
%j	연중 일(day of the year)을 십진수로 [001,366].	
%m	월을 십진수로 [01,12].	
%M	분을 십진수로 [00,59].	
%p	AM이나 PM에 해당하는 로케일의 값.	(2)
%S	초를 십진수로 [00,61].	(3)
%U	연중 주 번호(일요일이 주의 시작)를 십진수로 [00,53]. 첫 번째 일요일에 선행하는 새해의 모든 날은 주 0으로 간주합니다.	(4)
%u	요일(월요일은 1; 일요일은 7)을 십진수로 [1, 7].	
%w	요일을 십진수로 [0(일요일),6].	
%W	연중 주 번호(월요일이 주의 시작)를 십진수로 [00,53]. 첫 번째 월요일에 선행하는 새해의 모든 말은 주 0으로 간주합니다.	(4)
%x	로케일의 적절한 날짜 표현.	
%X	로케일의 적절한 시간 표현.	
%y	세기가 없는 해(year)를 십진수로 [00,99].	
%Y	세기가 있는 해(year)를 십진수로.	
%z	UTC/GMT로부터의 양이나 음의 시차를 +HHMM 또는 -HHMM 형식으로 나타내는 시간대 오프셋. 여기서 H는 십진수 시 숫자를 나타내고 M은 십진수 분 숫자를 나타냅니다 [-23:59, +23:59].[1]	
%Z	시간대 이름 (시간대가 없으면 문자가 없습니다). 폐지되었습니다.Page 778, 1	
%G	ISO 8601 year (similar to %Y but follows the rules for the ISO 8601 calendar year). The year starts with the week that contains the first Thursday of the calendar year.	
%V	ISO 8601 week number (as a decimal number [01,53]). The first week of the year is the one that contains the first Thursday of the	

노트:

(1) The `%f` format directive only applies to `strptime()`, not to `strftime()`. However, see also `datetime.datetime.strptime()` and `datetime.datetime.strftime()` where the `%f` format directive *applies to microseconds*.

(2) `strptime()` 함수와 함께 사용할 때, `%I` 지시자를 사용하여 시(hour)를 구문 분석하면 `%p` 지시자는 출력 시(hour) 필드에만 영향을 줍니다.

(3) 범위는 실제로 0에서 61입니다; 값 60은 윤초를 나타내는 타임 스탬프에서 유효하고 값 61은 역사적 이유로 지원됩니다.

(4) `strptime()` 함수와 함께 사용할 때, `%U`와 `%W`는 주중 일(day of the week)과 해(year)가 지정된 경우에만 계산에 사용됩니다.

다음은 **RFC 2822** 인터넷 전자우편 표준에서 지정된 것과 호환되는 날짜 형식의 예입니다.[1]

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

특정 플랫폼에서는 추가 지시자가 지원될 수 있지만, 여기에 나열된 지시자에만 ANSI C에서 표준화된 의미가 있습니다. 플랫폼에서 지원되는 전체 포맷 코드 집합을 보려면, `strftime(3)` 설명서를 참조하십시오.

일부 플랫폼에서, 선택적 필드 너비와 정밀도 명세는 지시자의 초기 `'%'` 뒤에 그 순서대로 나올 수 있습니다; 이것 또한 이식성이 없습니다. 필드 너비는 일반적으로 2이며, `%j`는 3입니다.

`time.`**`strptime`**(*string*[, *format*])

포맷(format)에 따라 시간을 나타내는 문자열을 구문 분석합니다. 반환 값은 `gmtime()`이나 `localtime()` 에 의해 반환되는 것과 같은 `struct_time`입니다.

format 매개 변수는 `strftime()`에서 사용된 것과 같은 지시자를 사용합니다; 기본값은 `ctime()`이 반환한 포맷과 일치하는 `"%a %b %d %H:%M:%S %Y"`입니다. *format*에 따라 *string*을 구문 분석할 수 없거나, 구문 분석 후 여분의 데이터가 있으면, `ValueError`가 발생합니다. 더 정확한 값을 유추할 수 없을 때 누락된 데이터를 채우는 데 사용되는 기본값은 `(1900, 1, 1, 0, 0, 0, 0, 1, -1)` 입니다. *string*과 *format*은 모두 문자열이어야 합니다.

예를 들면:

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                 tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

`%Z` 지시자에 대한 지원은 `tzname`에 포함된 값과 `daylight`가 참인지를 기반으로 합니다. 이로 인해, 항상 알려진 (그리고 일광 절약 시간제가 아닌 시간대로 간주하는) UTC와 GMT를 인식하는 것을 제외하고는 플랫폼에 따라 다릅니다.

설명서에 지정된 지시자만 지원됩니다. `strftime()`은 플랫폼별로 구현되기 때문에 때로는 나열된 것보다 많은 지시자를 제공할 수 있습니다. 그러나 `strptime()`은 플랫폼과 독립적이라서 지원된다고 설명하지 않은 사용 가능한 모든 지시자를 반드시 지원하지는 않습니다.

class `time.`**`struct_time`**

`gmtime()`, `localtime()` 및 `strptime()`이 반환하는 시간 값 시퀀스의 형. 네임드 튜플 인터페이스를 갖는 객체입니다: 인덱스와 어트리뷰트 이름으로 값에 액세스 할 수 있습니다. 다음과 같은 값이 있습니다:

[1] `%Z` 사용은 이제 폐지되었지만, 선호하는 시/분 오프셋으로 확장되는 `%z` 이스케이프가 모든 ANSI C 라이브러리에서 지원되는 것은 아닙니다. 또한, 원래 1982년 **RFC 822** 표준을 엄격하게 읽으면 2자리 연도(`%Y`가 아닌 `%y`)가 필요하지만, 실제로는 2000년 훨씬 이전에 4자리 연도로 옮겨졌습니다. 그 후, **RFC 822**는 더는 사용되지 않고 4자리 연도는 **RFC 1123**에서 처음 권장한 다음 **RFC 2822**에서 의무화합니다.

인덱스	어트리뷰트	값
0	**tm_year**	(예를 들어, 1993)
1	**tm_mon**	범위 [1, 12]
2	**tm_mday**	범위 [1, 31]
3	**tm_hour**	범위 [0, 23]
4	**tm_min**	범위 [0, 59]
5	**tm_sec**	범위 [0, 61]; *strftime()* 의 노트 (2)를 참조하십시오
6	**tm_wday**	범위 [0, 6]; 월요일은 0
7	**tm_yday**	범위 [1, 366]
8	**tm_isdst**	0, 1 또는 -1; 아래를 참조하십시오
해당 없음	**tm_zone**	시간대 이름의 약어
해당 없음	**tm_gmtoff**	UTC에서 동쪽으로 초 단위 오프셋

C 구조체와 달리, 월 값의 범위는 [0, 11]이 아니라 [1, 12] 임에 유의하십시오.

mktime() 호출에서, 일광 절약 시간제가 발효 중이면 *tm_isdst*가 1로 설정되고, 그렇지 않으면 0으로 설정될 수 있습니다. 값이 -1이면 알 수 없다는 뜻이고, 일반적으로 올바른 상태가 채워집니다.

길이가 잘못된 튜플이 *struct_time*을 기대하는 함수에 전달되거나, 잘못된 형의 요소가 있으면, *TypeError*가 발생합니다.

time.**time**() → *float*

에포크 이후 초를 나타내는 시간을 부동 소수점 숫자로 반환합니다. 윤초의 처리는 플랫폼에 따라 다릅니다. 윈도우와 대부분의 유닉스 시스템에서, 윤초는 에포크 이후의 초를 나타내는 시간에 계산되지 않습니다. 이것을 흔히 유닉스 시간이라고 합니다.

시간이 항상 부동 소수점 숫자로 반환되더라도, 모든 시스템이 1초보다 정밀한 정밀도를 제공하는 것은 아님에 유의하십시오. 이 함수는 일반적으로 감소하지 않는 값을 반환하지만, 두 호출 사이에 시스템 시계가 뒤로 설정되면 이전 호출보다 작은 값을 반환할 수 있습니다.

*time()*에 의해 반환된 숫자는 더 일반적인 시간 형식(즉 년, 월, 일, 시 등…)으로 변환될 수 있는데, *gmtime()* 함수에 전달하여 UTC로, *localtime()* 함수에 전달하여 현지 시간으로 변환할 수 있습니다. 두 경우 모두 달력 날짜의 구성 요소를 어트리뷰트로 액세스 할 수 있는 *struct_time* 객체가 반환됩니다.

Clock:

- On Windows, call `GetSystemTimePreciseAsFileTime()`.

- Call `clock_gettime(CLOCK_REALTIME)` if available.
- Otherwise, call `gettimeofday()`.

Use `time_ns()` to avoid the precision loss caused by the `float` type.

버전 3.13 에서 변경: On Windows, calls `GetSystemTimePreciseAsFileTime()` instead of `GetSystemTimeAsFileTime()`.

`time.`**`time_ns`**`()` → *int*

`time()`과 비슷하지만, 시간을 에포크 이후의 나노초를 나타내는 정수로 반환합니다.

Added in version 3.7.

`time.`**`thread_time`**`()` → *float*

현재 스레드의 시스템과 사용자 CPU 시간 합계의 값을 (소수부가 있는 초로) 반환합니다. 수면 중에 지난 시간은 포함되지 않습니다. 정의상 스레드 수준입니다. 반환된 값의 기준점은 정의되어 있지 않아서, 같은 스레드에서 이루어지는 두 호출 결과 간의 차이만 유효합니다.

Use `thread_time_ns()` to avoid the precision loss caused by the `float` type.

가용성: Linux, Unix, Windows.

`CLOCK_THREAD_CPUTIME_ID`를 지원하는 유닉스 시스템.

Added in version 3.7.

`time.`**`thread_time_ns`**`()` → *int*

`thread_time()`과 비슷하지만, 시간을 나노초로 반환합니다.

Added in version 3.7.

`time.`**`tzset`**`()`

라이브러리 루틴이 사용하는 시간 변환 규칙을 재설정합니다. 환경 변수 `TZ`는 이것이 수행되는 방법을 지정합니다. 또한 변수 `tzname` (`TZ` 환경 변수에서), `timezone` (UTC에서 서쪽으로 DST가 아닌 초), `altzone` (UTC에서 서쪽으로 DST 초) 및 `daylight` (이 시간대가 일광 절약 시간 규칙이 없으면 0으로, 일광 절약 시간이 적용될 때 시간, 과거, 현재 또는 미래가 있으면 0이 아닌 값으로)를 설정합니다.

가용성: Unix.

> **참고**
>
> 많은 경우에, `TZ` 환경 변수를 변경하면 `tzset()`을 호출하지 않고도 `localtime()` 과 같은 함수의 출력에 영향을 줄 수 있지만, 이 동작에 의존해서는 안 됩니다.
>
> `TZ` 환경 변수에는 공백이 없어야 합니다.

`TZ` 환경 변수의 표준 형식은 다음과 같습니다 (명확함을 위해 공백을 추가했습니다):

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

구성 요소는 다음과 같습니다:

std와 dst
: 시간대 약어를 제공하는 세 개 이상의 영숫자. 이들은 time.tzname으로 전파됩니다

offset
: 오프셋은 다음과 같은 형식입니다: `± hh[:mm[:ss]]`. UTC에 도달하기 위해 현지 시간에 더하는 값을 나타냅니다. 앞에 '-'가 있으면, 시간대는 본초 자오선(Prime Meridian)의 동쪽입니다; 그렇지 않으면, 서쪽입니다. dst 다음에 오프셋이 없으면, 일광 절약 시간은 표준 시간보다 1시간 빠르다고 가정합니다.

start[/time], end[/time]
: 언제 DST로 변경하고, 언제 돌아오는지를 나타냅니다. 시작(start) 날짜와 종료(end) 날짜의 형식은 다음 중 하나입니다:

Jn
율리우스 일 n (1 <= n <= 365). 윤일(leap days)은 계산되지 않아서, 모든 연도에서 2월 28일은 59일이고 3월 1일은 60일입니다.

n
0부터 시작하는 율리우스 일 (0 <= n <= 365). 윤일(leap days)이 계산되며, 2월 29일을 가리킬 수 있습니다.

Mm.n.d
연중 월 m의 주 n의 d번째 요일 (0 <= d <= 6, 1 <= n <= 5, 1 <= m <= 12, 여기서 주 5는 "월 m의 마지막 d 요일"을 뜻하는데, 4번째나 5번째 주에 발생할 수 있습니다). 주 1은 d번째 요일이 등장하는 첫 주입니다. 요일 0은 일요일입니다.

`time`은 선행 부호('-'나 '+')가 허용되지 않는다는 점을 제외하고 `offset`과 형식이 같습니다. time이 제공되지 않으면, 기본값은 02:00:00입니다.

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

많은 유닉스 시스템(*BSD, 리눅스, Solaris 및 Darwin을 포함합니다)에서, 시스템의 zoneinfo (*tzfile(5)*) 데이터베이스를 사용하여 시간대 규칙을 지정하는 것이 더 편리합니다. 이렇게 하려면, `TZ` 환경 변수를 시스템 'zoneinfo' 시간대 데이터베이스의 루트(일반적으로 `/usr/share/zoneinfo`에 있습니다)에 상대적인 필요한 시간대 데이터 파일의 경로로 설정하십시오. 예를 들어, `'US/Eastern'`, `'Australia/Melbourne'`, `'Egypt'` 또는 `'Europe/Amsterdam'`.

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

16.3.2 시계 ID 상수

이 상수들은 *clock_getres()*와 *clock_gettime()*의 매개 변수로 사용됩니다.

`time.`**CLOCK_BOOTTIME**

*CLOCK_MONOTONIC*과 동일하지만, 시스템이 일시 중단된 시간도 포함합니다.

이를 통해 응용 프로그램은 `settimeofday()` 등을 사용하여 시간이 변경되면 불연속성이 발생할 수 있는 *CLOCK_REALTIME*의 복잡함을 다루지 않고도 일시 중단을 인식하는 단조 시계를 얻을 수 있습니다.

가용성: Linux >= 2.6.39.

Added in version 3.7.

`time.`**CLOCK_HIGHRES**

Solaris OS에는 최적의 하드웨어 소스를 사용하려고 하는 `CLOCK_HIGHRES` 타이머가 있으며, 나노초에 가까운 해상도를 제공합니다. `CLOCK_HIGHRES`는 조정 불가능한 고해상도 시계입니다.

가용성: Solaris.

Added in version 3.3.

time.**CLOCK_MONOTONIC**

설정할 수 없고 어떤 지정되지 않은 시작점 이후의 단조 시간을 나타내는 시계.

가용성: Unix.

Added in version 3.3.

time.**CLOCK_MONOTONIC_RAW**

*CLOCK_MONOTONIC*과 비슷하지만, NTP 조정이 적용되지 않는 원시 하드웨어 기반 시간에 대한 액세스를 제공합니다.

가용성: Linux >= 2.6.28, macOS >= 10.12.

Added in version 3.3.

time.**CLOCK_MONOTONIC_RAW_APPROX**

Similar to *CLOCK_MONOTONIC_RAW*, but reads a value cached by the system at context switch and hence has less accuracy.

가용성: macOS >= 10.12.

Added in version 3.13.

time.**CLOCK_PROCESS_CPUTIME_ID**

CPU의 고해상도 프로세스별 타이머.

가용성: Unix.

Added in version 3.3.

time.**CLOCK_PROF**

CPU의 고해상도 프로세스별 타이머.

가용성: FreeBSD, NetBSD >= 7, OpenBSD.

Added in version 3.7.

time.**CLOCK_TAI**

International Atomic Time

이것이 정확한 답을 주려면 시스템에 최신 윤초 표가 있어야 합니다. PTP나 NTP 소프트웨어는 윤초 표를 유지할 수 있습니다.

가용성: Linux.

Added in version 3.9.

time.**CLOCK_THREAD_CPUTIME_ID**

스레드별 CPU 시간 시계.

가용성: Unix.

Added in version 3.3.

time.**CLOCK_UPTIME**

절댓값이 시스템이 실행되고 정지되지 않은 시간인 시간, 절대와 간격 모두로, 정확한 가동 시간 측정을 제공합니다.

가용성: FreeBSD, OpenBSD >= 5.5.

Added in version 3.7.

time.**CLOCK_UPTIME_RAW**

주파수나 시간 조정에 영향을 받지 않고 시스템이 휴면하는 중에는 증가하지 않는 임의의 지점 이후의 시간을 추적하면서 단조 증가 하는 시계.

가용성: macOS >= 10.12.

Added in version 3.8.

time.**CLOCK_UPTIME_RAW_APPROX**

Like *CLOCK_UPTIME_RAW*, but the value is cached by the system at context switches and therefore has less accuracy.

가용성: macOS >= 10.12.

Added in version 3.13.

다음 상수는 *clock_settime()*에 보낼 수 있는 유일한 매개 변수입니다.

time.**CLOCK_REALTIME**

Real-time clock. Setting this clock requires appropriate privileges. The clock is the same for all processes.

가용성: Unix.

Added in version 3.3.

16.3.3 시간대 상수

time.**altzone**

정의된 것이 있다면, 현지 DST 시간대의 UTC 서쪽으로 초 단위 오프셋. 현지 DST 시간대가 UTC 동쪽이면 음수입니다 (영국을 포함한 서유럽의 경우). `daylight`가 0이 아닌 경우에만 사용하십시오. 아래 참고 사항을 참조하십시오.

time.**daylight**

DST 시간대가 정의되면 0이 아닙니다. 아래 참고 사항을 참조하십시오.

time.**timezone**

UTC 서쪽으로 초 단위의, 현지 (DST가 아닌) 시간대의 오프셋 (대부분 서부 유럽 지역에서는 음수, 미국에서는 양수, 영국에서는 0). 아래 참고 사항을 참조하십시오.

time.**tzname**

두 문자열의 튜플: 첫 번째는 현지 DST가 아닌 시간대의 이름이고, 두 번째는 현지 DST 시간대의 이름입니다. DST 시간대가 정의되어 있지 않으면, 두 번째 문자열을 사용하지 않아야 합니다. 아래 참고 사항을 참조하십시오.

참고

위의 시간대 상수 (*altzone*, *daylight*, *timezone* 및 *tzname*)의 경우, 값은 모듈 로드 시간이나 *tzset()*이 마지막으로 호출된 시간에 적용되는 시간대 규칙에 의해 결정되며 과거의 시간에는 올바르지 않을 수 있습니다. 시간대 정보를 얻으려면 *localtime()* 결과의 *tm_gmtoff*와 *tm_zone*을 사용하는 것이 좋습니다.

더 보기

모듈 ***datetime***
날짜와 시간에 대한 더 객체 지향적인 인터페이스.

모듈 ***locale***
국제화 서비스. 로케일 설정은 *strftime()*과 *strptime()*의 많은 포맷 지시자의 해석에 영향을 줍니다.

모듈 ***calendar***
일반적인 캘린더 관련 함수. *timegm()*은 이 모듈에 있는 *gmtime()*의 역함수입니다.

16.4 logging — 파이썬 로깅 시설

소스 코드: Lib/logging/__init__.py

Important

이 페이지는 API 레퍼런스 정보를 담고 있습니다. 자습서 정보 및 고급 주제에 대한 설명은 다음을 참조하십시오.

- 기초 자습서
- 고급 자습서
- 로깅 요리책

이 모듈은 응용 프로그램과 라이브러리를 위한 유연한 이벤트 로깅 시스템을 구현하는 함수와 클래스를 정의합니다.

표준 라이브러리 모듈로 로깅 API를 제공하는 것의 주요 이점은, 모든 파이썬 모듈이 로깅에 참여할 수 있어서, 응용 프로그램 로그에 여러분 자신의 메시지를 제삼자 모듈의 메시지와 통합할 수 있다는 것입니다.

Here' s a simple example of idiomatic usage:

```
# myapp.py
import logging
import mylib
logger = logging.getLogger(__name__)

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logger.info('Started')
    mylib.do_something()
    logger.info('Finished')

if __name__ == '__main__':
    main()
```

```
# mylib.py
import logging
logger = logging.getLogger(__name__)

def do_something():
    logger.info('Doing something')
```

If you run *myapp.py*, you should see this in *myapp.log*:

```
INFO:__main__:Started
INFO:mylib:Doing something
INFO:__main__:Finished
```

The key feature of this idiomatic usage is that the majority of code is simply creating a module level logger with `getLogger(__name__)`, and using that logger to do any needed logging. This is concise, while allowing downstream code fine-grained control if needed. Logged messages to the module-level logger get forwarded to handlers of loggers in higher-level modules, all the way up to the highest-level logger known as the root logger; this approach is known as hierarchical logging.

For logging to be useful, it needs to be configured: setting the levels and destinations for each logger, potentially changing how specific modules log, often based on command-line arguments or application configuration. In most cases, like the one above, only the root logger needs to be so configured, since all the lower level loggers at module

level eventually forward their messages to its handlers. `basicConfig()` provides a quick way to configure the root logger that handles many use cases.

이 모듈은 많은 기능과 유연성을 제공합니다. 로깅에 익숙하지 않다면, 감을 잡는 가장 좋은 방법은 자습서를 보는 것입니다 (**위와 오른쪽에 있는 링크를 참조하세요**).

모듈에 의해 정의된 기본 클래스는, 그들의 어트리뷰트와 메서드와 함께, 아래 절에서 나열됩니다.

- 로거는 응용 프로그램 코드가 직접 사용하는 인터페이스를 노출합니다.
- 처리기는 (로거가 만든) 로그 레코드를 적절한 목적지로 보냅니다.
- 필터는 출력할 로그 레코드를 결정하기 위한 더 세분된 기능을 제공합니다.
- 포매터는 최종 출력에서 로그 레코드의 배치를 지정합니다.

16.4.1 Logger 객체

로거에는 다음과 같은 어트리뷰트와 메서드가 있습니다. 로거는 결코 직접 인스턴스를 만드는 일 없이, 항상 모듈 수준의 함수 `logging.getLogger(name)` 를 거치는 것에 주의하십시오. 같은 이름(name)으로 `getLogger()` 를 여러 번 호출해도 항상 같은 로거 객체에 대한 참조를 돌려줍니다.

`name` 은 잠재적으로 `foo.bar.baz` 와 같이 마침표로 구분된 계층적 값입니다 (하지만 그냥 간단한 `foo` 도 가능합니다). 계층적 목록에서 더 아래쪽에 있는 로거는 목록에서 상위에 있는 로거의 자식입니다. 예를 들어, 이름이 `foo` 인 로거가 주어지면, `foo.bar`, `foo.bar.baz`, 그리고 `foo.bam` 의 이름을 가진 로거는 모두 `foo` 의 자손입니다. 또한, 모든 로거는 루트 로거의 자손입니다. 로거 이름 계층 구조는 파이썬 패키지 계층 구조와 비슷하며, `logging.getLogger(__name__)` 를 사용하여 모듈 단위로 로거를 구성하는 경우는 패키지 계층 구조와 같아집니다. 왜냐하면, 모듈에서, `__name__` 은 파이썬 패키지 이름 공간의 모듈 이름이기 때문입니다.

class `logging.`**`Logger`**

name

This is the logger' s name, and is the value that was passed to `getLogger()` to obtain the logger.

> **참고**
>
> This attribute should be treated as read-only.

level

The threshold of this logger, as set by the `setLevel()` method.

> **참고**
>
> Do not set this attribute directly - always use `setLevel()`, which has checks for the level passed to it.

parent

The parent logger of this logger. It may change based on later instantiation of loggers which are higher up in the namespace hierarchy.

> **참고**
>
> This value should be treated as read-only.

propagate

이 어트리뷰트가 참으로 평가되면, 이 로거에 로그 된 이벤트는 이 로거에 연결된 처리기뿐 아니라 상위 계층 (조상) 로거의 처리기로 전달됩니다. 메시지는 조상 로거의 처리기에 직접 전달됩니다 - 조상 로거의 수준이나 필터는 고려하지 않습니다.

이 값이 거짓으로 평가되면, 로깅 메시지가 조상 로거의 처리기로 전달되지 않습니다.

Spelling it out with an example: If the propagate attribute of the logger named `A.B.C` evaluates to true, any event logged to `A.B.C` via a method call such as `logging.getLogger('A.B.C').error(...)` will [subject to passing that logger' s level and filter settings] be passed in turn to any handlers attached to loggers named `A.B`, `A` and the root logger, after first being passed to any handlers attached to `A.B.C`. If any logger in the chain `A.B.C`, `A.B`, `A` has its `propagate` attribute set to false, then that is the last logger whose handlers are offered the event to handle, and propagation stops at that point.

생성자는 이 어트리뷰트를 `True` 로 설정합니다.

> **참고**
>
> 로거 와 하나 이상의 조상에 처리기를 중복해서 연결하면, 같은 레코드를 여러 번 출력할 수 있습니다. 일반적으로, 하나 이상의 로거에 처리기를 붙일 필요는 없습니다. 로거 계층에서 가장 높은 적절한 로거에 처리기를 연결하면, propagate 설정이 `True` 로 남아있는 모든 자식 로거들이 로그 하는 모든 이벤트를 보게 됩니다. 일반적인 시나리오는 루트 로거에만 처리기를 연결하고, 전파가 나머지를 처리하도록 하는 것입니다.

handlers

The list of handlers directly attached to this logger instance.

> **참고**
>
> This attribute should be treated as read-only; it is normally changed via the *addHandler()* and *removeHandler()* methods, which use locks to ensure thread-safe operation.

disabled

This attribute disables handling of any events. It is set to `False` in the initializer, and only changed by logging configuration code.

> **참고**
>
> This attribute should be treated as read-only.

setLevel(*level*)

이 로거의 수준 경계를 *level* 로 설정합니다. *level* 보다 덜 심각한 로깅 메시지는 무시됩니다; 심각도 *level* 이상의 로깅 메시지는, 처리기 수준이 *level* 보다 높은 심각도 수준으로 설정되지 않는 한, 이 로거에 연결된 처리기가 출력합니다.

로거가 만들어질 때, 수준은 *NOTSET* (로거가 루트 로거 일 때는 모든 메시지를 처리하게 하고, 로거가 루트 로거가 아니면 모든 메시지를 부모에게 위임하도록 합니다) 으로 설정됩니다. 루트 로거는 수준 *WARNING*으로 만들어짐에 유의하세요.

'부모에게 위임' 이라는 말은, 로거 수준이 NOTSET 인 경우, NOTSET 이외의 수준을 갖는 조상이 발견되거나 루트에 도달할 때까지 조상 로거 체인을 탐색함을 의미합니다.

NOTSET 이외의 수준을 갖는 조상이 발견되면, 그 조상의 수준을 조상 검색이 시작된 로거의 유효 수준으로 간주하여, 로깅 이벤트를 처리할지를 결정하는 데 사용됩니다.

루트에 도달하면, 그리고 루트가 NOTSET 수준을 갖고 있으면, 모든 메시지가 처리됩니다. 그렇지 않으면 루트 수준이 유효 수준으로 사용됩니다.

수준의 목록은 로깅 수준를 보세요.

버전 3.2에서 변경: *level* 매개 변수는 이제 *INFO*와 같은 정수 상수 대신 'INFO' 와 같은 수준의 문자열 표현을 허용합니다. 그러나 수준은 내부적으로 정수로 저장되며, *getEffectiveLevel()* 및 *isEnabledFor()* 와 같은 메서드는 정수를 반환하거나 정수가 전달되기를 기대합니다.

isEnabledFor(*level*)

심각도 *level* 의 메시지가 이 로거에서 처리될지를 알려줍니다. 이 메서드는 먼저 `logging.disable(level)` 에 의해 설정된 모듈 수준의 수준을 확인한 다음, `getEffectiveLevel()` 로 확인되는 로거의 유효 수준을 검사합니다.

getEffectiveLevel()

이 로거의 유효 수준을 알려줍니다. `setLevel()` 을 사용하여 `NOTSET` 이외의 값이 설정되면, 그 값이 반환됩니다. 그렇지 않으면, `NOTSET` 이외의 값이 발견될 때까지 루트를 향해 계층 구조를 탐색하고, 그 값이 반환됩니다. 반환되는 값은 정수이며, 일반적으로 `logging.DEBUG`, `logging.INFO` 등 중 하나입니다.

getChild(*suffix*)

접미사에 의해 결정되는, 이 로거의 자손 로거를 반환합니다. 그러므로, `logging.getLogger('abc').getChild('def.ghi')` 는 `logging.getLogger('abc.def.ghi')` 와 같은 로거를 반환합니다. 이것은 편의 메서드인데, 부모 로거가 리터럴 문자열이 아닌 이름(가령 `__name__`)을 사용하여 명명될 때 유용합니다.

Added in version 3.2.

getChildren()

Returns a set of loggers which are immediate children of this logger. So for example `logging.getLogger().getChildren()` might return a set containing loggers named `foo` and `bar`, but a logger named `foo.bar` wouldn' t be included in the set. Likewise, `logging.getLogger('foo').getChildren()` might return a set including a logger named `foo.bar`, but it wouldn' t include one named `foo.bar.baz`.

Added in version 3.12.

debug(*msg*, **args*, ***kwargs*)

이 로거에 수준 `DEBUG` 메시지를 로그 합니다. *msg* 는 메시지 포맷 문자열이고, *args* 는 문자열 포매팅 연산자를 사용하여 *msg* 에 병합되는 인자입니다. (이는 포맷 문자열에 키워드를 사용하고, 인자로 하나의 딕셔너리를 전달할 수 있음을 의미합니다.) *args* 가 제공되지 않으면 *msg* 에 % 포매팅 연산이 수행되지 않습니다.

kwargs 에서 검사되는 네 개의 키워드 인자가 있습니다: *exc_info*, *stack_info*, *stacklevel* 및 *extra*.

exc_info 가 거짓으로 평가되지 않으면, 로깅 메시지에 예외 정보가 추가됩니다. 예외 튜플 (`sys.exc_info()` 에 의해 반환되는 형식) 또는 예외 인스턴스가 제공되면 사용됩니다; 그렇지 않으면 예외 정보를 얻기 위해 `sys.exc_info()` 를 호출합니다.

두 번째 선택적 키워드 인자는 *stack_info* 이며, 기본값은 `False` 입니다. 참이면, 실제 로깅 호출을 포함하는 스택 정보가 로깅 메시지에 추가됩니다. 이것은 *exc_info* 를 지정할 때 표시되는 것과 같은 스택 정보가 아닙니다: 전자(*stack_info*)는 스택의 맨 아래에서 현재 스레드의 로깅 호출까지의 스택 프레임이며, 후자(*exc_info*)는 예외가 일어난 후에 예외 처리기를 찾으면서 되감은 스택 프레임에 대한 정보입니다.

exc_info 와는 독립적으로 *stack_info* 를 지정할 수 있습니다. 예를 들어 예외가 발생하지 않은 경우에도 코드의 특정 지점에 어떻게 도달했는지 보여줄 수 있습니다. 스택 프레임은 다음과 같은 헤더 행 다음에 인쇄됩니다:

```
Stack (most recent call last):
```

예외 프레임을 표시할 때 사용되는 `Traceback (most recent call last):` 을 흉내 내고 있습니다.

세 번째 선택적 키워드 인자는 *stacklevel* 이며, 기본값은 `1` 입니다. 1보다 크면, 로깅 이벤트용으로 만들어진 `LogRecord` 에 설정된 줄 번호와 함수 이름을 계산할 때 해당 수의 스택 프레임을 건너뜁니다. 기록된 함수 이름, 파일명 및 줄 번호가 도우미 함수/메서드에 대한 정보가 아니라 호출자의 정보가 되도록, 로깅 도우미에서 사용할 수 있습니다. 이 매개 변수의 이름은 `warnings` 모듈에 있는 비슷한 것과 같도록 맞췄습니다.

네 번째 키워드 인자는 *extra* 로, 로깅 이벤트용으로 만들어진 `LogRecord` 의 `__dict__` 를 사용자 정의 어트리뷰트로 채우는 데 사용되는 딕셔너리를 전달할 수 있습니다. 이러한 사용자 정의

어트리뷰트는 원하는 대로 사용할 수 있습니다. 예를 들어, 로그 메시지에 포함할 수 있습니다. 예를 들면:

```
FORMAT = '%(asctime)s %(clientip)-15s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)
```

는 이렇게 인쇄할 것입니다

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection␣
↪reset
```

extra 에 전달된 딕셔너리의 키가, 로깅 시스템에서 사용하는 키와 충돌해서는 안 됩니다. (어떤 키가 로깅 시스템에 의해 사용되는지에 대한 더 많은 정보는 *LogRecord* 어트리뷰트에 있는 절을 보십시오.)

로그 된 메시지에서 이러한 어트리뷰트를 사용하려면 몇 가지 주의를 기울여야 합니다. 위의 예에서, 예를 들어, `Formatter` 에 설정한 포맷 문자열은 `LogRecord`의 어트리뷰트 딕셔너리에 'clientip' 과 'user' 가 있을 것으로 기대하고 있습니다. 이것들이 없는 경우 문자열 포매팅 예외가 발생하기 때문에 메시지가 기록되지 않습니다. 따라서 이 경우, 항상 이 키를 포함하는 *extra* 딕셔너리를 전달해야 합니다.

성가신 일입니다만, 이 기능은 여러 문맥에서 같은 코드가 실행되고 관심 있는 조건들(가령 원격 클라이언트 IP 주소와 인증된 사용자 이름)이 문맥에 따라 발생하는 다중 스레드 서버와 같은 특수한 상황을 위한 것입니다. 이런 상황에서는, 특수한 `Formatter` 가 특정한 `Handler`와 함께 사용될 가능성이 큽니다.

If no handler is attached to this logger (or any of its ancestors, taking into account the relevant `Logger.propagate` attributes), the message will be sent to the handler set on `lastResort`.

버전 3.2에서 변경: *stack_info* 매개 변수가 추가되었습니다.

버전 3.5에서 변경: *exc_info* 매개 변수는 이제 예외 인스턴스를 받아들입니다.

버전 3.8에서 변경: *stacklevel* 매개 변수가 추가되었습니다.

info(*msg*, **args*, ***kwargs*)

이 로거에 수준 `INFO` 메시지를 로그 합니다. 인자는 `debug()`처럼 해석됩니다.

warning(*msg*, **args*, ***kwargs*)

이 로거에 수준 `WARNING` 메시지를 로그 합니다. 인자는 `debug()`처럼 해석됩니다.

> **참고**
>
> 기능적으로 `warning` 와 같은, 구식의 `warn` 메서드가 있습니다. `warn` 은 폐지되었으므로 사용하지 마십시오 - 대신 `warning` 을 사용하십시오.

error(*msg*, **args*, ***kwargs*)

이 로거에 수준 `ERROR` 메시지를 로그 합니다. 인자는 `debug()`처럼 해석됩니다.

critical(*msg*, **args*, ***kwargs*)

이 로거에 수준 `CRITICAL` 메시지를 로그 합니다. 인자는 `debug()`처럼 해석됩니다.

log(*level*, *msg*, **args*, ***kwargs*)

이 로거에 정수 수준 *level* 로 메시지를 로그 합니다. 다른 인자는 `debug()`처럼 해석됩니다.

exception(*msg*, **args*, ***kwargs*)

이 로거에 수준 `ERROR` 메시지를 로그 합니다. 인자는 `debug()`처럼 해석됩니다. 예외 정보가 로깅 메시지에 추가됩니다. 이 메서드는 예외 처리기에서만 호출해야 합니다.

addFilter(*filter*)

지정된 필터 *filter* 를 이 로거에 추가합니다.

removeFilter(*filter*)

이 로거에서 지정된 필터 *filter* 를 제거합니다.

filter(*record*)

이 로거의 필터를 레코드(record)에 적용하고 레코드가 처리 대상이면 `True`를 반환합니다. 필터 중 어느 하나가 거짓 값을 반환할 때까지 필터는 차례로 참조됩니다. 그중 아무것도 거짓 값을 반환하지 않으면 레코드가 처리됩니다 (처리기로 전달됩니다). 어느 하나가 거짓 값을 반환하면, 더 이상의 레코드 처리는 이루어지지 않습니다.

addHandler(*hdlr*)

지정된 처리기 *hdlr* 를 이 로거에 추가합니다.

removeHandler(*hdlr*)

이 로거에서 지정된 처리기 *hdlr* 을 제거합니다.

findCaller(*stack_info=False*, *stacklevel=1*)

호출자의 소스 파일 이름과 행 번호를 찾습니다. 파일 이름, 행 번호, 함수 이름 및 스택 정보를 4-요소 튜플로 반환합니다. 스택 정보는 *stack_info* 가 `True` 가 아니면 `None` 으로 반환됩니다.

stacklevel 매개 변수는 `debug()`과 기타 API를 호출하는 코드에서 전달됩니다. 1보다 크면, 반환되는 값을 결정하기 전에 초과분만큼 스택 프레임을 건너뜁니다. 일반적으로 도우미/래퍼 코드에서 로깅 API를 호출할 때 유용한데, 이벤트 로그의 정보가 도우미/래퍼 코드가 아니라 호출자 코드를 참조하도록 합니다.

handle(*record*)

이 로거와 그 조상(거짓 값의 *propagate* 가 발견될 때까지)과 연관된 모든 처리기에 레코드를 전달하여 레코드를 처리합니다. 이 메서드는 로컬에서 만든 레코드뿐만 아니라 소켓에서 받아서 언피클된 레코드를 처리하는 데 사용됩니다. `filter()` 를 사용하여 로거 수준 필터링을 적용합니다.

makeRecord(*name*, *level*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*)

이 메서드는 특수한 `LogRecord` 인스턴스를 만들기 위해 서브 클래스에서 재정의할 수 있는 팩토리 메서드입니다.

hasHandlers()

이 로거에 처리기가 구성되어 있는지 확인합니다. 이 로거의 처리기와 로거 계층의 부모를 찾습니다. 처리기가 발견되면 `True` 를 반환하고, 그렇지 않으면 `False` 를 반환합니다. 이 메서드는 'propagate' 어트리뷰트가 거짓으로 설정된 로거가 발견될 때 계층 구조 검색을 중지합니다 - 그 로거가 처리기가 있는지 검사하는 마지막 로거가 됩니다.

Added in version 3.2.

버전 3.7에서 변경: 이제 로거는 피클 되고 언피클 될 수 있습니다.

16.4.2 로깅 수준

로깅 수준의 숫자 값은 다음 표에 나와 있습니다. 여러분 자신의 수준을 정의하고, 미리 정의된 수준과 상대적인 특정 값을 갖도록 하려는 경우 필요합니다. 같은 숫자 값을 가진 수준을 정의하면 미리 정의된 값을 덮어씁니다; 미리 정의된 이름이 유실됩니다.

수준	숫자 값	What it means / When to use it
logging.**NOTSET**	0	When set on a logger, indicates that ancestor loggers are to be consulted to determine the effective level. If that still resolves to NOTSET, then all events are logged. When set on a handler, all events are handled.
logging.**DEBUG**	10	Detailed information, typically only of interest to a developer trying to diagnose a problem.
logging.**INFO**	20	Confirmation that things are working as expected.
logging.**WARNING**	30	An indication that something unexpected happened, or that a problem might occur in the near future (e.g. 'disk space low'). The software is still working as expected.
logging.**ERROR**	40	Due to a more serious problem, the software has not been able to perform some function.
logging.**CRITICAL**	50	A serious error, indicating that the program itself may be unable to continue running.

16.4.3 Handler 객체

처리기에는 다음과 같은 어트리뷰트와 메서드가 있습니다. *Handler* 는 절대로 직접 인스턴스로 만들어지지 않음에 주의하세요; 이 클래스는 더욱 유용한 서브 클래스의 베이스가 됩니다. 그러나, 서브 클래스의 __init__() 메서드는 *Handler.__init__()* 을 호출해야 합니다.

class logging.**Handler**

__init__(*level=NOTSET*)

수준을 설정하고, 필터 목록을 빈 리스트로 설정하고, I/O 메커니즘에 대한 액세스를 직렬화하기 위해 (*createLock()* 을 사용하여) 록을 생성함으로써 *Handler* 인스턴스를 초기화합니다.

createLock()

스레드 안전하지 않은 하부 I/O 기능에 대한 액세스를 직렬화하는 데 사용할 수 있는 스레드 록을 초기화합니다.

acquire()

createLock() 로 생성된 스레드 록을 확보합니다.

release()

acquire() 로 확보한 스레드 록을 반납합니다.

setLevel(*level*)

이 처리기의 수준 경계를 *level* 로 설정합니다. *level* 보다 덜 심각한 로깅 메시지는 무시됩니다. 처리기가 만들어질 때, 수준은 *NOTSET* (모든 메시지가 처리되게 합니다) 으로 설정됩니다.

수준의 목록은 로깅 수준를 보세요.

버전 3.2에서 변경: *level* 매개 변수는 이제 *INFO*와 같은 정수 상수 대신 'INFO' 와 같은 수준 문자열 표현을 허용합니다.

setFormatter(*fmt*)

Sets the formatter for this handler to *fmt*. The *fmt* argument must be a *Formatter* instance or None.

addFilter(*filter*)

지정된 필터 *filter* 를 이 처리기에 추가합니다.

removeFilter(*filter*)

이 처리기에서 지정된 필터 *filter* 를 제거합니다.

filter(*record*)

이 처리기의 필터를 레코드에 적용하고 레코드가 처리 대상이면 `True`를 반환합니다. 필터 중 어느 하나가 거짓 값을 반환할 때까지 필터는 차례로 확인됩니다. 그중 아무것도 거짓 값을 반환하지 않으면 레코드가 출력됩니다. 어느 하나가 거짓 값을 반환하면 처리기는 레코드를 출력하지 않습니다.

flush()

모든 로그 출력이 플러시 되었음을 확실히 합니다. 이 버전은 아무것도 하지 않으며, 서브 클래스에 의해 구현됩니다.

close()

처리기가 사용하는 자원을 정리합니다. 이 버전은 출력하지 않지만, 이름으로 처리기를 검색하는 데 사용되는 내부 맵에서 처리기를 제거합니다.

서브 클래스는 이것이 재정의된 `close()` 메서드에서 이 메서드를 호출해야 합니다.

handle(*record*)

처리기에 추가된 필터에 따라 조건부로, 지정된 로깅 레코드를 출력합니다. 레코드의 실제 출력을 I/O 스레드 록의 확보/해제로 둘러쌉니다.

handleError(*record*)

이 메서드는 `emit()` 호출 중에 예외가 발생할 때 처리기에서 호출됩니다. 모듈 수준 어트리뷰트 `raiseExceptions` 가 `False` 인 경우 예외는 조용히 무시됩니다. 이 동작은 대부분 로깅 시스템에서 원하는 방식입니다 - 대부분 사용자는 로깅 시스템 자체의 에러에 관심이 없고, 응용 프로그램 에러에 더 관심이 있습니다. 그러나 원하는 경우, 사용자 정의 처리기로 바꿀 수 있습니다. 지정된 레코드는 예외가 발생할 때 처리되고 있던 레코드입니다. (`raiseExceptions` 의 기본값은 `True` 입니다. 개발 중에 더 유용합니다).

format(*record*)

레코드를 포맷합니다 - 포매터가 설정된 경우 사용합니다. 그렇지 않으면 모듈의 기본 포매터를 사용합니다.

emit(*record*)

지정된 로깅 레코드를 실제로 로그 하는 데 필요한 작업을 수행합니다. 이 버전은 서브 클래스에 의해 구현될 것으로 보고 `NotImplementedError`를 발생시킵니다.

> **경고**
>
> This method is called after a handler-level lock is acquired, which is released after this method returns. When you override this method, note that you should be careful when calling anything that invokes other parts of the logging API which might do locking, because that might result in a deadlock. Specifically:
>
> - Logging configuration APIs acquire the module-level lock, and then individual handler-level locks as those handlers are configured.
> - Many logging APIs lock the module-level lock. If such an API is called from this method, it could cause a deadlock if a configuration call is made on another thread, because that thread will try to acquire the module-level lock *before* the handler-level lock, whereas this thread tries to acquire the module-level lock *after* the handler-level lock (because in this method, the handler-level lock has already been acquired).

표준으로 포함된 처리기 목록은 `logging.handlers` 를 참조하십시오.

16.4.4 Formatter 객체

class logging.**Formatter**(*fmt=None, datefmt=None, style='%', validate=True, *, defaults=None*)

Responsible for converting a `LogRecord` to an output string to be interpreted by a human or external system.

매개변수

- **fmt** (str) – A format string in the given *style* for the logged output as a whole. The possible mapping keys are drawn from the `LogRecord` object' s *LogRecord* 어트리뷰트. If not specified, `'%(message)s'` is used, which is just the logged message.
- **datefmt** (str) – A format string for the date/time portion of the logged output. If not specified, the default described in `formatTime()` is used.
- **style** (str) – Can be one of `'%'`, `'{'` or `'$'` and determines how the format string will be merged with its data: using one of *printf* 스타일 문자열 포매팅 (`%`), `str.format()` (`{`) or `string.Template` (`$`). This only applies to *fmt* (e.g. `'%(message)s'` versus `'{message}'`), not to the actual log messages passed to the logging methods. However, there are other ways to use `{`- and `$`-formatting for log messages.
- **validate** (bool) – `True`면 (기본값), *fmt*와 *style*이 잘못되거나 일치하지 않을 때 `ValueError`를 발생시킵니다; 예를 들어, `logging.Formatter('%(asctime)s - %(message)s', style='{')`.
- **defaults** (dict[str, Any]) – A dictionary with default values to use in custom fields. For example, `logging.Formatter('%(ip)s %(message)s', defaults={"ip": None})`

버전 3.2에서 변경: *style* 매개 변수가 추가되었습니다.

버전 3.8에서 변경: *validate* 매개 변수가 추가되었습니다.

버전 3.10에서 변경: *defaults* 매개 변수가 추가되었습니다.

format(*record*)

레코드의 어트리뷰트 딕셔너리가 문자열 포매팅 연산의 피연산자로 사용됩니다. 결과 문자열을 반환합니다. 딕셔너리를 포맷하기 전에 몇 가지 준비 단계가 수행됩니다. 레코드의 *message* 어트리뷰트를 *msg % args* 를 사용하여 계산합니다. 포매팅 문자열에 `'(asctime)'` 이 들어 있으면, `formatTime()` 이 호출되어 이벤트 시간을 포매팅합니다. 예외 정보가 있는 경우, `formatException()` 을 사용하여 포매팅 되고 메시지에 덧붙입니다. 포맷된 예외 정보는 *exc_text* 어트리뷰트에 캐시 됩니다. 예외 정보를 피클 해서 네트워크를 통해 전송할 수 있으므로 유용합니다만, 예외 정보의 포매팅을 사용자 정의하는 `Formatter` 서브 클래스가 두 개 이상 있는 경우 주의해야 합니다. 이 경우, 한 포매터가 포매팅을 완료한 후 (*exc_text* 어트리뷰트를 `None`으로 설정하여) 캐시 된 값을 지워서 그 이벤트를 처리하는 다음 포매터가 캐시 된 값을 사용하지 않고, 새로 계산할 수 있도록 해야 합니다.

스택 정보가 있는 경우, 예외 정보 뒤에 덧붙입니다. 필요할 경우 `formatStack()` 을 사용하여 변환합니다.

formatTime(*record, datefmt=None*)

이 메서드는 포맷된 시간을 사용하려는 포매터에 의해 `format()`에서 호출되어야 합니다. 이 메서드는 특정 요구 사항을 제공하기 위해 포매터에서 재정의될 수 있지만, 기본 동작은 다음과 같습니다: *datefmt*(문자열)이 지정된 경우, `time.strftime()` 를 사용하여 레코드 생성 시간을 포매팅합니다. 그렇지 않으면 '%Y-%m-%d %H:%M:%S,uuu' 포맷이 사용됩니다. 여기서 uuu 부분은 밀리 초 값이고, 다른 문자들은 `time.strftime()` 설명서를 따릅니다. 이 포맷의 표현된 시간의 예는 `2003-01-23 00:29:50,411` 입니다. 결과 문자열이 반환됩니다.

이 함수는 사용자가 구성할 수 있는 함수를 사용하여 생성 시간을 튜플로 변환합니다. 기본적으로 `time.localtime()` 이 사용됩니다; 특정 포매터 인스턴스에서 이를 변경하려면, `converter` 어트리뷰트를 `time.localtime()` 또는 `time.gmtime()` 과 같은 서명을 가진 함수로 설정하십시오. 모든 포매터를 변경하려면, 예를 들어 모든 로깅 시간을 GMT로 표시하려면, `Formatter` 클래스의 `converter` 어트리뷰트를 설정하십시오.

버전 3.3에서 변경: 예전에는, 기본 포맷이 다음과 같이 하드 코딩되었습니다: `2010-09-06 22:38:15,292`. 쉼표 앞에 있는 부분은 strptime 포맷 문자열(`'%Y-%m-%d %H:%M:%S'`)이며, 쉼

표 뒤의 부분은 밀리 초 값입니다. strptime에 밀리 초 포맷 표시자가 없으므로, 밀리 초 값은 다른 포맷 문자열 `'%s,%03d'` 을 사용하여 추가됩니다 — 이 두 포맷 문자열 모두 이 메서드에 하드 코드 되었습니다. 이 변경으로, 이 문자열들은 클래스 수준 어트리뷰트로 정의되었고, 원하는 경우 인스턴스 수준에서 재정의할 수 있습니다. 어트리뷰트 이름은 `default_time_format`(strptime 포맷 문자열)과 `default_msec_format`(밀리 초 값 추가용)입니다.

버전 3.9에서 변경: `default_msec_format` 은 `None` 일 수 있습니다.

formatException(*exc_info*)

지정된 예외 정보(`sys.exc_info()` 에 의해 반환되는 표준 예외 튜플)를 문자열로 포맷합니다. 이 기본 구현은 `traceback.print_exception()` 을 사용합니다. 결과 문자열이 반환됩니다.

formatStack(*stack_info*)

지정된 스택 정보(`traceback.print_stack()` 에 의해 반환된 문자열이지만 마지막 줄 바꿈이 제거됩니다)을 문자열로 포맷합니다. 이 기본 구현은 입력 값을 그대로 반환합니다.

class logging.**BufferingFormatter**(*linefmt=None*)

A base formatter class suitable for subclassing when you want to format a number of records. You can pass a `Formatter` instance which you want to use to format each line (that corresponds to a single record). If not specified, the default formatter (which just outputs the event message) is used as the line formatter.

formatHeader(*records*)

Return a header for a list of *records*. The base implementation just returns the empty string. You will need to override this method if you want specific behaviour, e.g. to show the count of records, a title or a separator line.

formatFooter(*records*)

Return a footer for a list of *records*. The base implementation just returns the empty string. You will need to override this method if you want specific behaviour, e.g. to show the count of records or a separator line.

format(*records*)

Return formatted text for a list of *records*. The base implementation just returns the empty string if there are no records; otherwise, it returns the concatenation of the header, each record formatted with the line formatter, and the footer.

16.4.5 Filter 객체

`Filter` 는 수준을 통해 제공되는 것보다 더 정교한 필터링을 위해 `Handler` 와 `Logger` 에 의해 사용될 수 있습니다. 베이스 필터 클래스는 로거 계층 구조의 특정 지점 아래에 있는 이벤트만 허용합니다. 예를 들어 'A.B' 로 초기화된 필터는, 로거 'A.B', 'A.B.C', 'A.B.C.D', 'A.B.D' 등이 로그 한 이벤트를 허용하지만, 'A.BB', 'B.A.B' 등은 허용하지 않습니다. 빈 문자열을 사용하면 모든 이벤트를 통과시킵니다.

class logging.**Filter**(*name=''*)

`Filter` 클래스의 인스턴스를 반환합니다. *name* 을 제공하면, 필터를 통과하도록 허용할 로거(그 자식들도 포함합니다)의 이름을 지정합니다. *name* 이 빈 문자열이면, 모든 이벤트를 허용합니다.

filter(*record*)

Is the specified record to be logged? Returns false for no, true for yes. Filters can either modify log records in-place or return a completely different record instance which will replace the original log record in any future processing of the event.

처리기에 연결된 필터는 이벤트를 처리기가 출력하기 전에 호출되는 반면, 로거에 연결된 필터는 이벤트가 로깅될 때마다 (`debug()`, `info()` 등) 처리기로 이벤트를 보내기 전에 호출됩니다. 이는 자손 로거가 만든 이벤트들은, 같은 필터가 자손들에게도 적용되지 않는 한, 로거의 필터 설정으로 필터링 되지 않는다는 것을 뜻합니다.

실제로 `Filter` 의 서브 클래스를 만들 필요는 없습니다: 같은 의미가 있는 `filter` 메서드를 가진 인스턴스는 무엇이건 전달할 수 있습니다.

버전 3.2에서 변경: 특수한 `Filter` 클래스를 만들거나 `filter` 메서드를 가진 다른 클래스를 사용할 필요가 없습니다: 함수(또는 다른 콜러블)를 필터로 사용할 수 있습니다. 필터링 로직은 필터 객체가 `filter`

어트리뷰트를 가졌는지 확인합니다: 만약 있다면 `Filter` 라고 가정하고 `filter()` 메서드를 호출합니다. 그렇지 않으면 콜러블이라고 가정하고 레코드를 단일 매개 변수로 호출합니다. 반환된 값은 `filter()` 가 반환하는 값과 같은 의미를 지녀야 합니다.

버전 3.12에서 변경: You can now return a `LogRecord` instance from filters to replace the log record rather than modifying it in place. This allows filters attached to a `Handler` to modify the log record before it is emitted, without having side effects on other handlers.

필터는 수준보다 정교한 기준에 따라 레코드를 필터링하는 데 주로 사용되지만, 필터가 연결되는 처리기나 로거에서 처리되는 모든 레코드를 볼 수 있습니다: 이 특성은, 특정 로거나 처리기가 얼마나 많은 레코드를 처리하는지 센다거나, 처리 중인 `LogRecord`에 어트리뷰트를 추가, 변경, 삭제하려고 할 때 유용합니다. 당연히, LogRecord를 변경하는 것은 주의를 필요로 하는 일이지만, 로그에 문맥 정보를 주입하는 것을 허용합니다 (filters-contextual를 보세요).

16.4.6 LogRecord 객체

`LogRecord` 인스턴스는 뭔가 로깅 될 때마다 `Logger` 에 의해 자동으로 생성되며, `makeLogRecord()` 를 통해 수동으로 생성될 수 있습니다 (예를 들어, 네트워크에서 수신된 피클 된 이벤트의 경우).

class `logging.`**`LogRecord`**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*)

로그 되는 이벤트와 관련된 모든 정보를 담고 있습니다.

주요 정보는 *msg*와 *args* 로 전달되며, `msg % args` 를 사용하여 병합되어 레코드의 `message` 어트리뷰트를 만듭니다.

매개변수

- **name** (`str`) – 이 `LogRecord`가 나타내는 이벤트를 로그 하는데 사용된 로거의 이름. `LogRecord`의 로거 이름은 다른 (조상) 로거에 연결된 처리기가 출력하더라도 항상 이 값을 갖게됨에 유의하십시오.
- **level** (`int`) – 로깅 이벤트의 숫자 수준 (가령 `10`은 `DEBUG`, `20`은 `INFO` 등). 이 값은 LogRecord의 두 어트리뷰트로 변환됩니다: 숫자 값을 위한 `levelno` 와 해당 수준 이름을 위한 `levelname`.
- **pathname** (`str`) – 로깅 호출이 발생한 소스 파일의 전체 문자열 경로.
- **lineno** (`int`) – 로깅 호출이 발생한 소스 파일의 행 번호.
- **msg** (`Any`) – 이벤트 설명 메시지. 변수 데이터를 위한 자리 표시자가 있는 %-포맷 문자열이거나, 임의의 객체일 수 있습니다 (arbitrary-object-messages를 참조하세요).
- **args** (`tuple | dict[str, Any]`) – 이벤트 설명을 얻기 위해 *msg* 인자에 병합할 변수 데이터.
- **exc_info** (`tuple[type[BaseException], BaseException, types.TracebackType] | None`) – 현재의 예외 정보를 가지는 예외 튜플로, `sys.exc_info()`가 반환하는 값. 예외 정보가 없는 경우는 `None`.
- **func** (`str | None`) – 로깅 호출을 호출한 함수 또는 메서드의 이름.
- **sinfo** (`str | None`) – 현재 스레드에서 스택의 바닥부터 로깅 호출까지의 스택 정보를 나타내는 텍스트 문자열.

getMessage()

사용자가 제공 한 인자를 메시지와 병합한 후, 이 `LogRecord` 인스턴스에 대한 메시지를 반환합니다. 로깅 호출에 제공된 사용자 제공 메시지 인자가 문자열이 아닌 경우, `str()` 이 호출되어 문자열로 변환됩니다. 이렇게 해서 사용자 정의 클래스를 메시지로 사용할 수 있도록 하는데, 그 클래스의 `__str__` 메서드는 사용할 실제 포맷 문자열을 반환 할 수 있습니다.

버전 3.2에서 변경: 레코드를 생성하는 데 사용되는 팩토리를 제공함으로써, `LogRecord`의 생성을 더 구성할 수 있게 만들었습니다. 팩토리는 `getLogRecordFactory()`와 `setLogRecordFactory()` (팩토리의 서명은 이곳을 참조하십시오) 를 사용하여 설정할 수 있습니다.

이 기능은 *LogRecord* 생성 시에 여러분 자신의 값을 주입하는데 사용할 수 있습니다. 다음과 같은 패턴을 사용할 수 있습니다:

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

이 패턴을 사용하면 여러 팩토리를 체인으로 묶을 수 있으며, 서로의 어트리뷰트를 덮어쓰거나 위에 나열된 표준 어트리뷰트를 실수로 덮어쓰지 않는 한 놀랄만한 일이 일어나지는 않아야 합니다.

16.4.7 LogRecord 어트리뷰트

LogRecord에는 많은 어트리뷰트가 있으며, 대부분 어트리뷰트는 생성자의 매개 변수에서 옵니다. (LogRecord 생성자 매개 변수와 LogRecord 어트리뷰트의 이름이 항상 정확하게 일치하는 것은 아닙니다.) 이러한 어트리뷰트를 사용하여 레코드의 데이터를 포맷 문자열로 병합 할 수 있습니다. 다음 표는 어트리뷰트 이름, 의미와 해당 자리 표시자를 %-스타일 포맷 문자열로 (알파벳 순서로) 나열합니다.

{}-포매팅(*str.format()*)을 사용한다면, `{attrname}` 을 포맷 문자열의 자리 표시자로 사용할 수 있습니다. $-포매팅(*string.Template*)을 사용하고 있다면, `${attrname}` 형식을 사용하십시오. 두 경우 모두, 물론, `attrname` 을 사용하려는 실제 어트리뷰트 이름으로 대체하십시오.

{}-포매팅의 경우, 어트리뷰트 이름 다음에 콜론(:)으로 구분하여 포매팅 플래그를 지정할 수 있습니다. 예를 들어, `{msecs:03.0f}` 자리 표시자는 밀리 초 값 `4` 를 `004` 로 포맷합니다. 사용할 수 있는 옵션에 대한 자세한 내용은 *str.format()* 설명서를 참조하십시오.

어트리뷰트 이름	포맷	설명
args	직접 포맷할 필요는 없습니다.	message 를 생성하기 위해 msg 에 병합되는 인자의 튜플. 또는 (인자가 하나뿐이고 딕셔너리일 때) 병합을 위해 값이 사용되는 딕셔너리.
asctime	%(asctime)s	사람이 읽을 수 있는, LogRecord 가 생성된 시간. 기본적으로 '2003-07-08 16:49:45,896' 형식입니다 (쉼표 뒤의 숫자는 밀리 초 부분입니다).
created	%(created)f	LogRecord 가 생성된 시간 (time.time_ns() / 1e9 이 반환하는 값).
exc_info	직접 포맷할 필요는 없습니다.	예외 튜플 (sys.exc_info 에서 제공) 또는, 예외가 발생하지 않았다면, None.
filename	%(filename)s	pathname 의 파일명 부분.
funcName	%(funcName)s	로깅 호출을 포함하는 함수의 이름.
levelname	%(levelname)s	메시지의 텍스트 로깅 수준 ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL').
levelno	%(levelno)s	메시지의 숫자 로깅 수준 (DEBUG, INFO, WARNING, ERROR, CRITICAL).
lineno	%(lineno)d	로깅 호출이 일어난 소스 행 번호 (사용 가능한 경우).
message	%(message)s	로그 된 메시지. msg % args 로 계산됩니다. Formatter.format() 이 호출 될 때 설정됩니다.
module	%(module)s	모듈 (filename 의 이름 부분).
msecs	%(msecs)d	LogRecord 가 생성된 시간의 밀리 초 부분.
msg	직접 포맷할 필요는 없습니다.	원래 로깅 호출에서 전달된 포맷 문자열. args 와 병합하여 message 를 만듭니다. 또는 임의의 객체 (arbitrary-object-messages 를 보세요).
name	%(name)s	로깅 호출에 사용된 로거의 이름.
pathname	%(pathname)s	로깅 호출이 일어난 소스 파일의 전체 경로명 (사용 가능한 경우).
process	%(process)d	프로세스 ID (사용 가능한 경우).
processName	%(processName)s	프로세스 이름 (사용 가능한 경우).
relativeCreated	%(relativeCreat	logging 모듈이 로드된 시간을 기준으로 LogRecord 가 생성된 시간 (밀리 초).
stack_info	직접 포맷할 필요는 없습니다.	현재 스레드의 스택 바닥에서 이 레코드를 생성한 로깅 호출의 스택 프레임까지의 스택 프레임 정보 (사용 가능한 경우).
thread	%(thread)d	스레드 ID (사용 가능한 경우).
threadName	%(threadName)s	스레드 이름 (사용 가능한 경우).
taskName	%(taskName)s	asyncio.Task 이름 (사용 가능한 경우).

버전 3.1 에서 변경: *processName* 이 추가되었습니다.

버전 3.12 에서 변경: *taskName* 이 추가되었습니다.

16.4.8 LoggerAdapter 객체

LoggerAdapter 인스턴스는 문맥 정보를 로깅 호출에 편리하게 전달하는 데 사용됩니다. 사용 예는, 로그 출력에 문맥 정보 추가 섹션을 참조하십시오.

class logging.**LoggerAdapter**(*logger*, *extra*, *merge_extra=False*)

Returns an instance of LoggerAdapter initialized with an underlying Logger instance, a dict-like object (*extra*), and a boolean (*merge_extra*) indicating whether or not the *extra* argument of individual log calls should be merged with the LoggerAdapter extra. The default behavior is to ignore the *extra* argument of individual log calls and only use the one of the LoggerAdapter instance

process(*msg*, *kwargs*)

문맥 정보를 삽입하기 위해 로깅 호출에 전달된 메시지 와 키워드 인자를 수정합니다. 이 구현은 생성자에 *extra* 로 전달된 객체를 가져와서 'extra' 키를 사용하여 *kwargs* 에 추가합니다. 반환 값은 전달된 인자의 (수정된) 버전을 담은 (*msg*, *kwargs*) 튜플입니다.

manager

Delegates to the underlying `manager` on *logger*.

_log

Delegates to the underlying `_log()` method on *logger*.

위의 것에 더해, `LoggerAdapter` 는 다음과 같은 `Logger` 의 메서드를 지원합니다: `debug()`, `info()`, `warning()`, `error()`, `exception()`, `critical()`, `log()`, `isEnabledFor()`, `getEffectiveLevel()`, `setLevel()`, `hasHandlers()`. 이 메서드들은 `Logger` 에 있는 것과 똑같은 서명을 가지므로, 두 형의 인스턴스를 바꿔쓸 수 있습니다.

버전 3.2에서 변경: `isEnabledFor()`, `getEffectiveLevel()`, `setLevel()` 그리고 `hasHandlers()` 메서드가 `LoggerAdapter` 에 추가되었습니다. 이 메서드는 하부 로거로 위임합니다.

버전 3.6에서 변경: Attribute `manager` and method `_log()` were added, which delegate to the underlying logger and allow adapters to be nested.

버전 3.13에서 변경: *merge_extra* 인자가 추가되었습니다.

16.4.9 스레드 안전성

The logging module is intended to be thread-safe without any special work needing to be done by its clients. It achieves this through using threading locks; there is one lock to serialize access to the module' s shared data, and each handler also creates a lock to serialize access to its underlying I/O.

`signal` 모듈을 사용하여 비동기 시그널 처리기를 구현한다면, 그 처리기 내에서는 logging을 사용할 수 없을 수도 있습니다. 이는 `threading` 모듈의 록 구현이 언제나 재진입할 수 있지는 않아서 그러한 시그널 처리기에서 호출할 수 없기 때문입니다.

16.4.10 모듈 수준 함수

위에서 설명한 클래스 외에도 많은 모듈 수준 함수가 있습니다.

`logging.`**`getLogger`**(*name=None*)

지정된 이름(name)의 로거를 돌려주거나, name이 `None` 인 경우, 계층의 루트 로거를 돌려줍니다. 지정된 경우, name은 일반적으로 '*a*', '*a.b*' 또는 '*a.b.c.d*' 와 같이 점으로 구분된 계층적 이름입니다. 이 이름의 선택은 전적으로 logging을 사용하는 개발자에게 달려 있지만, *Logger* 객체에서 언급한 대로, 특별한 이유가 없는 한 `__name__`을 사용하는 것을 권장합니다.

같은 이름으로 이 함수를 여러 번 호출하면 모두 같은 로거 인스턴스를 반환합니다. 이것은 응용 프로그램의 다른 부분 간에 로거 인스턴스를 전달할 필요가 없다는 것을 뜻합니다.

`logging.`**`getLoggerClass`**()

표준 `Logger` 클래스를 반환하거나, `setLoggerClass()` 에 전달된 마지막 클래스를 반환합니다. 이 함수는 새 클래스 정의 내에서 호출하여, 사용자 정의 `Logger` 클래스를 설치할 때 다른 코드가 이미 적용한 사용자 정의를 취소하지 않도록 할 수 있습니다. 예를 들면:

```
class MyLogger(logging.getLoggerClass()):
    # ... 여기에서 동작을 재정의 합니다
```

`logging.`**`getLogRecordFactory`**()

`LogRecord` 를 생성하는 데 사용되는 콜러블을 반환합니다.

Added in version 3.2: 이 함수는 `setLogRecordFactory()` 와 함께 제공되어, 개발자가 로깅 이벤트를 나타내는 `LogRecord` 가 만들어지는 방법을 더욱 잘 제어 할 수 있도록 합니다.

팩토리가 어떻게 호출되는지에 대한 더 자세한 정보는 `setLogRecordFactory()` 를 보세요.

`logging.`**`debug`**(*msg, *args, **kwargs*)

This is a convenience function that calls `Logger.debug()`, on the root logger. The handling of the arguments is in every way identical to what is described in that method.

The only difference is that if the root logger has no handlers, then `basicConfig()` is called, prior to calling `debug` on the root logger.

For very short scripts or quick demonstrations of `logging` facilities, `debug` and the other module-level functions may be convenient. However, most programs will want to carefully and explicitly control the logging configuration, and should therefore prefer creating a module-level logger and calling `Logger.debug()` (or other level-specific methods) on it, as described at the beginning of this documentation.

`logging.info`(*msg, *args, **kwargs*)

루트 로거에 수준 `INFO` 메시지를 로그 합니다. 그 밖의 인자와 동작은 `debug()`와 같습니다.

`logging.warning`(*msg, *args, **kwargs*)

루트 로거에 수준 `WARNING` 메시지를 로그 합니다. 그 밖의 인자와 동작은 `debug()`와 같습니다.

> **참고**
>
> 기능적으로 `warning`와 같은, 구식의 `warn` 함수가 있습니다. `warn`은 폐지되었으므로 사용하지 마십시오 - 대신 `warning`을 사용하십시오.

`logging.error`(*msg, *args, **kwargs*)

루트 로거에 수준 `ERROR` 메시지를 로그 합니다. 그 밖의 인자와 동작은 `debug()`와 같습니다.

`logging.critical`(*msg, *args, **kwargs*)

루트 로거에 수준 `CRITICAL` 메시지를 로그 합니다. 그 밖의 인자와 동작은 `debug()`와 같습니다.

`logging.exception`(*msg, *args, **kwargs*)

루트 로거에 수준 `ERROR` 메시지를 로그 합니다. 그 밖의 인자와 동작은 `debug()`와 같습니다. 예외 정보가 로깅 메시지에 추가됩니다. 이 메서드는 예외 처리기에서만 호출해야 합니다.

`logging.log`(*level, msg, *args, **kwargs*)

루트 로거에 수준 *level*의 메시지를 로그 합니다. 그 밖의 인자와 동작은 `debug()`와 같습니다.

`logging.disable`(*level=CRITICAL*)

모든 로거의 수준을 *level*로 오버라이드합니다. 로거 자체 수준보다 우선합니다. 전체 응용 프로그램에서 로깅 출력을 일시적으로 억제해야 할 필요가 생길 때 이 함수가 유용합니다. 그 효과는 심각도 *level* 및 그 밑의 모든 로깅 호출을 무효화시킵니다. 따라서 INFO 값으로 호출하면 모든 INFO 및 DEBUG 이벤트는 삭제되지만, WARNING 이상의 심각도는 로거의 유효 수준에 따라 처리됩니다. `logging.disable(logging.NOTSET)`이 호출되면, 이 오버라이딩 수준을 실질적으로 제거하므로, 로깅 출력은 다시 개별 로거의 유효 수준에 따르게 됩니다.

`CRITICAL`보다 더 높은 사용자 정의 로깅 수준을 정의했다면 (권장하지 않습니다), *level* 매개 변수의 기본값에 의존할 수 없고 적절한 값을 명시적으로 제공해야 합니다.

버전 3.7에서 변경: *level* 매개 변수의 기본값은 수준 `CRITICAL`입니다. 이 변경 사항에 대한 자세한 내용은 bpo-28524를 참조하십시오.

`logging.addLevelName`(*level, levelName*)

내부 딕셔너리에 수준 *level*을 텍스트 *levelName*과 연결합니다. 이 딕셔너리는 숫자 수준을 텍스트 표현으로 매핑하는데 (예를 들어, `Formatter`가 메시지를 포매팅할 때) 사용됩니다. 이 기능을 사용해서 여러분 자신의 수준을 정의할 수도 있습니다. 제약 조건은, 사용되는 모든 수준이 이 함수를 사용하여 등록되어야 하고, 수준은 양의 정수이어야 하며, 심각도가 높아질수록 값이 커져야 한다는 것입니다.

> **참고**
>
> 자신만의 수준을 정의할 생각이라면 custom-levels 섹션을 보십시오.

`logging.getLevelNamesMapping`()

Returns a mapping from level names to their corresponding logging levels. For example, the string "CRITICAL" maps to `CRITICAL`. The returned mapping is copied from an internal mapping on each call to this function.

Added in version 3.11.

logging.**getLevelName**(*level*)

로깅 수준 *level* 의 텍스트 표현을 반환합니다.

level 이 미리 정의된 수준 `CRITICAL`, `ERROR`, `WARNING`, `INFO` 또는 `DEBUG` 중 하나면 해당 문자열을 얻게 됩니다. `addLevelName()` 을 사용하여 수준과 이름을 연관 지었다면, *level* 과 연결된 이름이 반환됩니다. 정의된 수준 중 하나에 해당하는 숫자 값이 전달되면, 해당 문자열 표현이 반환됩니다.

level 매개 변수는 'INFO' 와 같은 수준의 문자열 표현도 받아들입니다. 이럴 때, 이 함수는 해당 수준의 숫자 값을 반환합니다.

일치하는 숫자나 문자열 값이 전달되지 않으면, 문자열 'Level %s' % level 이 반환됩니다.

> **참고**
>
> 수준은 (로깅 로직에서 비교해야 하므로) 내부적으로 정수입니다. 이 함수는 정수 수준과 `%(levelname)s` 포맷 지정자(*LogRecord* 어트리뷰트를 보세요)로 포맷된 로그 출력에 표시된 이름 간의 변환에 사용됩니다.

버전 3.4에서 변경: 3.4 이전의 파이썬 버전에서, 이 함수로 텍스트 수준을 전달할 수 있고, 해당 수준의 숫자 값을 반환합니다. 이 문서로 만들어지지 않은 동작은 실수로 간주하여, 파이썬 3.4에서 제거되었습니다. 하지만 이전 버전과의 호환성을 유지하기 위해 3.4.2에서 복원되었습니다.

logging.**getHandlerByName**(*name*)

Returns a handler with the specified *name*, or `None` if there is no handler with that name.

Added in version 3.12.

logging.**getHandlerNames**()

Returns an immutable set of all known handler names.

Added in version 3.12.

logging.**makeLogRecord**(*attrdict*)

어트리뷰트가 *attrdict* 로 정의된 새로운 `LogRecord` 인스턴스를 만들어서 반환합니다. 이 함수는 피클 된 `LogRecord` 어트리뷰트 딕셔너리를 소켓으로 보내고, 수신 단에서 `LogRecord` 인스턴스로 재구성할 때 유용합니다.

logging.**basicConfig**(***kwargs*)

기본 `Formatter`로 `StreamHandler` 를 생성하고 루트 로거에 추가하여 로깅 시스템의 기본 구성을 수행합니다. 함수 `debug()`, `info()`, `warning()`, `error()` 그리고 `critical()` 은 루트 로거에 처리기가 정의되어 있지 않으면 자동으로 `basicConfig()` 를 호출합니다.

이 함수는 루트 로거에 이미 처리기가 구성되어있는 경우, 키워드 인자 *force* 가 `True`로 설정되지 않는 한, 아무 작업도 수행하지 않습니다.

> **참고**
>
> 이 함수는 다른 스레드가 시작되기 전에 메인 스레드에서 호출되어야 합니다. 2.7.1과 3.2 이전의 파이썬 버전에서, 이 함수를 여러 스레드에서 호출하면, (드문 경우지만) 처리기가 두 번 이상 루트 로거에 추가되어, 로그에 메시지가 중복되는 것과 같은 예기치 않은 결과가 발생할 수 있습니다.

다음 키워드 인자가 지원됩니다.

포맷	설명
filename	`StreamHandler` 대신 지정된 파일명을 사용해 `FileHandler`를 만들도록 지정합니다.
filemode	*filename* 이 지정되었으면, 이 모드 로 파일을 엽니다. 기본값은 `'a'` 입니다.
format	처리기에 지정된 포맷 문자열을 사용합니다. 기본값은 콜론으로 구분된 어트리뷰트 `levelname`, `name` 및 `message` 입니다.
datefmt	`time.strftime()` 에서 허용하는 방식대로 지정된 날짜/시간 포맷을 사용합니다.
style	*format* 을 지정하면, 포맷 문자열에 이 스타일을 사용합니다. `'%'`, `'{'`, `'$'` 중 하나인데 각각 *printf* 스타일, `str.format()`, `string.Template` 에 대응됩니다. 기본값은 `'%'` 입니다.
level	루트 로거의 수준을 지정된 수준 으로 설정합니다.
stream	`StreamHandler`의 초기화에 지정된 스트림을 사용합니다. 이 인자는 *filename* 과 호환되지 않습니다 - 둘 다 있으면 `ValueError` 가 발생합니다.
handlers	지정된 경우, 루트 로거에 추가할 이미 만들어진 처리기의 이터러블이어야 합니다. 아직 포매터 세트가 없는 처리기에는 이 함수에서 만들어진 기본 포매터가 지정됩니다. 이 인자는 *filename* 또는 *stream* 과 호환되지 않습니다 - 둘 다 있으면 `ValueError` 가 발생합니다.
force	이 키워드 인자가 참으로 지정되면, 루트 로거에 연결된 기존 처리기는 다른 인자에 지정된 대로 구성을 수행하기 전에 모두 제거되고 닫힙니다.
encoding	이 키워드 인자가 *filename*과 함께 지정되면, 그 값이 `FileHandler`가 만들어질 때 사용되어, 결국 출력 파일을 열 때 사용됩니다.
errors	이 키워드 인자가 *filename*과 함께 지정되면, 그 값이 `FileHandler`가 만들어질 때 사용되어, 결국 출력 파일을 열 때 사용됩니다. 지정하지 않으면, 'backslashreplace' 값이 사용됩니다. `None`이 지정되면, `open()`에 그대로 전달되는데, 이는 'errors' 를 전달한 것처럼 처리된다는 것에 유의하십시오.

버전 3.2에서 변경: *style* 인자가 추가되었습니다.

버전 3.3에서 변경: *handlers* 인자가 추가되었습니다. 호환되지 않는 인자(예를 들어, *handlers* 를 *stream* 이나 *filename* 과 함께 쓰거나, *stream* 을 *filename* 과 함께 쓰는 경우)가 있는 상황을 파악하기 위한 검사가 추가되었습니다.

버전 3.8에서 변경: *force* 인자가 추가되었습니다.

버전 3.9에서 변경: *encoding*과 *errors* 인자가 추가되었습니다.

`logging.`**`shutdown`**`()`

로깅 시스템에 모든 처리기를 플러시하고 닫아서 순차적인 종료를 수행하도록 알립니다. 응용 프로그램 종료 시 호출되어야 하고, 이 호출 후에는 로깅 시스템을 더는 사용하지 않아야 합니다.

logging 모듈이 임포트 될 때, 이 함수를 종료 처리기(`atexit`를 참조하십시오)로 등록하므로, 일반적으로 수동으로 수행할 필요가 없습니다.

`logging.`**`setLoggerClass`**`(`*klass*`)`

로거의 인스턴스를 만들 때 *klass* 클래스를 사용하도록 로깅 시스템에 지시합니다. 클래스는 `__init__()` 을 정의해야 하는데, name만 필수 인자로 요구하고, `__init__()` 는 `Logger.__init__()` 을 호출해야 합니다. 이 함수는 일반적으로 사용자 정의된 로거 동작이 필요한 응용 프로그램에서 로거의 인스턴스가 만들어지기 전에 호출됩니다. 이 호출 후, 다른 때와 마찬가지로, 서브 클래스를 사용하여 직접 로거의 인스턴스를 만들지 마십시오: 계속 `logging.getLogger()` API를 사용하여 로거를 얻으십시오.

`logging.`**`setLogRecordFactory`**`(`*factory*`)`

`LogRecord` 를 만드는데 사용되는 콜러블을 설정합니다.

매개변수

`factory` – 로그 레코드의 인스턴스를 만드는데 사용되는 팩토리 콜러블.

Added in version 3.2: 이 함수는 `getLogRecordFactory()`와 함께 제공되어, 개발자가 로깅 이벤트를 나타내는 `LogRecord` 가 만들어지는 방법을 더욱 잘 제어 할 수 있도록 합니다.

팩토리의 서명은 다음과 같습니다:

factory(name, level, fn, lno, msg, args, exc_info, func=None, sinfo=None, **kwargs)

name
로거 이름.

level
로깅 수준 (숫자).

fn
로깅 호출이 이루어진 파일의 전체 경로명.

lno
로깅 호출이 이루어진 파일의 행 번호.

msg
로깅 메시지

args
로깅 메시지에 대한 인자.

exc_info
예외 튜플 또는 None.

func
로깅 호출을 호출한 함수 또는 메서드의 이름

sinfo
traceback.print_stack() 가 제공하는 것과 같은 스택 트레이스백. 호출 계층 구조를 보여줍니다.

kwargs
추가 키워드 인자.

16.4.11 모듈 수준 어트리뷰트

logging.**lastResort**

"최후 수단 처리기" 는 이 어트리뷰트를 통해 제공됩니다. 이것은 WARNING 수준으로 sys.stderr 에 쓰는 *StreamHandler* 이고, 로깅 구성이 없을 때 로깅 이벤트를 처리하는 데 사용됩니다. 최종 결과는 sys.stderr 에 메시지를 출력하기만 하는 것입니다. 이것이 예전의 "no handlers could be found for logger XYZ" 라는 에러 메시지를 대체합니다. 어떤 이유로 이전 동작이 필요하면 lastResort 를 None 으로 설정할 수 있습니다.

Added in version 3.2.

logging.**raiseExceptions**

Used to see if exceptions during handling should be propagated.

기본값: True.

If *raiseExceptions* is False, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors.

16.4.12 warnings 모듈과의 통합

captureWarnings() 함수는 *logging*을 *warnings* 모듈과 통합하는데 사용될 수 있습니다.

logging.**captureWarnings**(*capture*)

이 함수는 logging 이 경고를 캡처하는 것을 켜고 끄는 데 사용됩니다.

capture 가 True 면, *warnings* 모듈에 의해 발행된 경고는 로깅 시스템으로 리디렉션됩니다. 특히, 경고는 *warnings.formatwarning()* 을 사용하여 포맷되고, 결과 문자열을 'py.warnings' 라는 이름의 로거에 심각도 *WARNING*으로 로그 합니다.

capture 가 `False` 면, 로깅 시스템으로의 경고 리디렉션은 멈추고, 경고는 원래 목적지(즉, `captureWarnings(True)` 가 호출되기 전에 적용되던 곳)로 리디렉션됩니다.

더 보기

모듈 *logging.config*
logging 모듈용 구성 API.

모듈 *logging.handlers*
logging 모듈에 포함된 유용한 처리기.

PEP 282 - 로깅 시스템
파이썬 표준 라이브러리에 포함하기 위해 이 기능을 설명한 제안.

원본 파이썬 로깅 패키지
logging 패키지의 원래 소스입니다. 이 사이트에서 제공되는 패키지 버전은 표준 라이브러리에 *logging* 패키지를 포함하지 않는 파이썬 1.5.2, 2.1.x 및 2.2.x에서 사용하기에 적합합니다.

16.5 logging.config — 로깅 구성

소스 코드: Lib/logging/config.py

Important

이 페이지에는 레퍼런스 정보만 있습니다. 자습서는 다음을 참조하십시오

- 기초 자습서
- 고급 자습서
- 로깅 요리책

이 절에서는 logging 모듈을 구성하기 위한 API에 관해 설명합니다.

16.5.1 구성 함수

다음 함수는 logging 모듈을 구성합니다. *logging.config* 모듈에 있습니다. 사용은 선택 사항입니다 — 이 함수들을 사용하거나 (*logging* 자체에서 정의된) 주 API를 호출하고 *logging*이나 *logging.handlers*에서 선언된 처리기를 정의해서 logging 모듈을 구성할 수 있습니다.

`logging.config.`**`dictConfig`**(*config*)

딕셔너리로 로깅 구성을 받습니다. 이 딕셔너리의 내용은 아래의 구성 딕셔너리 스키마에 설명되어 있습니다.

구성 중에 에러를 만나면, 이 함수는 적절하게 설명하는 메시지와 함께 *ValueError*, *TypeError*, *AttributeError* 또는 *ImportError*를 발생시킵니다. 다음은 에러를 발생시킬 수 있는 (불완전한) 조건 목록입니다:

- 문자열이 아니거나 실제 로깅 수준과 일치하지 않는 문자열인 `level`.
- 불리언이 아닌 `propagate` 값.
- 해당 대상이 없는 id.
- 증분(incremental) 호출 중에 발견된 존재하지 않는 처리기 id.
- 잘못된 로거 이름.
- 결정할 수 없는 내부나 외부 객체.

구문 분석은 DictConfigurator 클래스에 의해 수행되며, 생성자로는 구성에 사용되는 딕셔너리가 전달되고, 객체는 configure() 메서드를 가집니다. *logging.config* 모듈에는 초기에 DictConfigurator로 설정된 콜러블 어트리뷰트 dictConfigClass가 있습니다. 여러분 자신의 적절한 구현으로 dictConfigClass의 값을 바꿀 수 있습니다.

*dictConfig()*는 dictConfigClass를 호출해서 지정된 딕셔너리를 전달한 다음, 반환된 객체의 configure() 메서드를 호출하여 구성을 적용합니다:

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

예를 들어, DictConfigurator의 서브 클래스는 자체 __init__() 에서 DictConfigurator.__init__() 를 호출한 다음, 후속 configure() 호출에서 사용할 수 있는 사용자 정의 접두사를 설정할 수 있습니다. dictConfigClass는 이 새 서브 클래스에 연결되고, *dictConfig()*는 기본, 사용자 정의되지 않은 상태에서와 똑같이 호출될 수 있습니다.

Added in version 3.2.

logging.config.**fileConfig**(*fname, defaults=None, disable_existing_loggers=True, encoding=None*)

configparser-형식 파일에서 로깅 구성을 읽습니다. 파일 형식은 구성 파일 형식에 설명된 것과 같아야 합니다. 이 함수는 응용 프로그램에서 여러 번 호출 할 수 있어서, 최종 사용자가 여러 가지 미리 준비된 구성 중에서 선택할 수 있도록 합니다 (개발자가 선택 사항을 표시하고 선택한 구성을 로드하는 메커니즘을 제공한다면).

It will raise *FileNotFoundError* if the file doesn' t exist and *RuntimeError* if the file is invalid or empty.

매개변수

- **fname** – 파일명, 또는 파일류 객체, 또는 *RawConfigParser*에서 파생된 인스턴스. RawConfigParser-파생 인스턴스가 전달되면, 그대로 사용됩니다. 그렇지 않으면, *ConfigParser*의 인스턴스가 만들어지고, 이것으로 fname으로 전달된 객체로부터 구성을 읽습니다. *readline()* 메서드가 있으면, 파일류 객체라고 가정하고, *read_file()*을 사용하여 읽습니다; 그렇지 않으면, 파일명으로 간주하고 *read()*로 전달됩니다.
- **defaults** – ConfigParser로 전달되는 기본값을 이 인자로 지정할 수 있습니다.
- **disable_existing_loggers** – False로 지정되면, 이 호출이 이루어졌을 때 존재하는 로거는 활성화된 상태로 남습니다. 기본값은 True이므로, 과거 호환성을 유지하도록 이전 동작을 활성화합니다. 이 동작은 이미 존재하는 비 루트 로거를 그들이나 그들의 조상이 로깅 구성에서 명시적으로 명명되지 않으면 비활성화하는 것입니다.
- **encoding** – The encoding used to open file when *fname* is filename.

버전 3.4에서 변경: *RawConfigParser*의 서브 클래스의 인스턴스가 이제 fname에 대한 값으로 허용됩니다. 이것은 다음을 쉽게 합니다:

- 로깅 구성이 전체 응용 프로그램 구성의 일부인 구성 파일의 사용.
- 파일에서 읽어 들인 다음 fileConfig로 전달되기 전에 사용하는 응용 프로그램이 (예를 들어, 명령 줄 매개 변수나 실행 시간 환경의 다른 측면에 기반하여) 수정하는 구성의 사용.

버전 3.10에서 변경: Added the *encoding* parameter.

버전 3.12에서 변경: An exception will be thrown if the provided file doesn' t exist or is invalid or empty.

logging.config.**listen**(*port=DEFAULT_LOGGING_CONFIG_PORT, verify=None*)

지정된 포트에서 소켓 서버를 시작하고, 새 구성을 수신 대기합니다. 포트를 지정하지 않으면, 모듈의 기본 DEFAULT_LOGGING_CONFIG_PORT가 사용됩니다. 로깅 구성은 *dictConfig()*나 *fileConfig()*로 처리하기에 적합한 파일로 전송됩니다. 서버를 시작하기 위해 *start()*를 호출할 수 있는 *Thread* 인스턴스를 반환하고, 적절할 때 *join()*할 수 있습니다. 서버를 중지하려면, *stopListening()*을 호출하십시오.

verify 인자가 지정되면, 소켓을 통해 수신된 바이트열이 유효하고 처리되어야 하는지를 확인하는 콜러블이어야 합니다. 소켓을 통해 전송되는 것을 암호화 및/또는 서명하고, verify 콜러블이 서명

확인 및/또는 암호 해독을 수행할 수 있습니다. `verify` 콜러블은 단일 인자(소켓을 통해 수신된 바이트열)로 호출되며, 처리할 바이트열이나 바이트열을 버려야 함을 나타내기 위해 `None`을 반환합니다. 반환된 바이트열은 전달된 바이트열과 같을 수 있고 (예를 들어, 확인만 수행될 때), 또는 완전히 다를 수 있습니다 (아마도 암호 해독이 수행될 때).

소켓으로 구성을 보내려면, 구성 파일을 읽어서 소켓에 `struct.pack('>L', n)`를 사용하여 바이너리로 만든 4바이트의 길이를 앞에 붙인 바이트 시퀀스를 보냅니다.

> **참고**
>
> 구성 일부가 *eval()*로 전달되므로, 이 함수를 사용하면 사용자를 보안 위험에 노출할 수 있습니다. 이 함수는 소켓을 `localhost`에만 바인드하고, 원격 기계의 연결은 허용하지 않지만, 신뢰할 수 없는 코드가 *listen()*을 호출하는 프로세스의 계정으로 실행될 수 있는 시나리오가 있습니다. 특히, *listen()*을 호출하는 프로세스가 사용자가 서로를 신뢰할 수 없는 다중 사용자 시스템에서 실행되는 경우, 악의적인 사용자는 피해자의 *listen()* 소켓에 연결하여 공격자가 피해자의 프로세스에서 실행하고자 하는 코드를 실행하는 구성을 보내는 것만으로도, 피해 사용자의 프로세스에서 사실상 임의의 코드를 실행할 수 있습니다. 이것은 기본 포트를 사용하는 경우 특히 쉽게 수행할 수 있지만, 다른 포트가 사용되는 경우에도 어렵지는 않습니다. 이러한 일이 발생할 위험을 피하려면, *listen()*에 `verify` 인자를 사용하여 인식되지 않은 구성이 적용되지 않도록 하십시오.

버전 3.4에서 변경: `verify` 인자가 추가되었습니다.

> **참고**
>
> 리스너에 기존 로거를 비활성화하지 않는 구성을 보내려면, *dictConfig()*를 사용하도록 구성에 JSON 형식을 사용해야 합니다. 이 방법은 보내는 구성에서 `disable_existing_loggers`를 `False`로 지정할 수 있도록 합니다.

`logging.config.`**`stopListening`**`()`

*listen()*에 대한 호출로 만들어진 리스닝 서버를 중지합니다. 이것은 일반적으로 *listen()*의 반환값에 대해 `join()`을 호출하기 전에 호출됩니다.

16.5.2 보안 고려 사항

The logging configuration functionality tries to offer convenience, and in part this is done by offering the ability to convert text in configuration files into Python objects used in logging configuration - for example, as described in 사용자 정의 객체. However, these same mechanisms (importing callables from user-defined modules and calling them with parameters from the configuration) could be used to invoke any code you like, and for this reason you should treat configuration files from untrusted sources with *extreme caution* and satisfy yourself that nothing bad can happen if you load them, before actually loading them.

16.5.3 구성 딕셔너리 스키마

로깅 구성을 기술하려면 만들려는 다양한 객체와 그들 간의 연결을 나열해야 합니다; 예를 들어, 'console' 이라는 처리기를 만든 다음 'startup' 이라는 로거가 'console' 처리기에 메시지를 보낼 것이라고 말할 수 있습니다. 사용자 자신의 포매터나 처리기 클래스를 작성할 수 있으므로, 이러한 객체가 *logging* 모듈에서 제공하는 객체로만 제한되지는 않습니다. 이러한 클래스의 매개 변수는 `sys.stderr`과 같은 외부 객체를 포함할 수도 있습니다. 이러한 객체와 연결을 기술하는 문법은 아래의 객체 연결에 정의되어 있습니다.

딕셔너리 스키마 세부사항

*dictConfig()*에 전달되는 딕셔너리에는 반드시 다음 키가 있어야 합니다:

- *version* - 스키마 버전을 나타내는 정숫값으로 설정됩니다. 현재 유효한 유일한 값은 1이지만, 이 키를 사용하면 과거 호환성을 유지하면서 스키마를 발전시킬 수 있습니다.

다른 모든 키는 선택 사항이지만, 있으면 아래에 설명된 대로 해석됩니다. 아래에서 '구성 딕셔너리 (configuring dict)' 가 언급되는 모든 경우에, 특수한 `'()'` 키를 검사해서 사용자 정의 인스턴스화가 필요한지를 확인합니다. 있다면, 아래의 사용자 정의 객체에 설명된 메커니즘을 사용하여 인스턴스를 만듭니다; 그렇지 않다면, 어떤 인스턴스를 만들지를 결정하는데 문맥이 사용됩니다.

- *formatters* - 해당 값은 딕셔너리인데, 각 키는 포매터 id이고, 각 값은 해당 `Formatter` 인스턴스를 구성하는 방법을 설명하는 딕셔너리입니다.

  `Formatter` 객체를 만들 때 전달된 인자에 대응되는, 다음과 같은 선택적 키를 구성 딕셔너리에서 검색합니다:

  - `format`
  - `datefmt`
  - `style`
  - `validate` (버전 >=3.8 부터)
  - `defaults` (버전 >=3.12 부터)

  선택적 `class` 키는 포매터 클래스의 이름을 나타냅니다 (점으로 구분된 모듈과 클래스 이름). 인스턴스화 인자는 `Formatter`와 같아서, 이 키는 사용자 정의 `Formatter` 서브 클래스를 인스턴스화하는 데 가장 유용합니다. 예를 들어, 대체 클래스는 확장되거나 축약한 형식으로 예외 트레이스백을 표시할 수 있습니다. 포매터가 다른 또는 추가 구성 키를 요구하면, 사용자 정의 객체를 사용해야 합니다.

- *filters* - 해당 값은 딕셔너리인데, 각 키가 필터 id이고 각 값은 해당 Filter 인스턴스를 구성하는 방법을 설명하는 딕셔너리입니다.

  구성 딕셔너리는 키 `name`(기본값은 빈 문자열)으로 검색되며, 이는 `logging.Filter` 인스턴스를 만드는 데 사용됩니다.

- *handlers* - 해당 값은 딕셔너리인데, 각 키가 처리기 id이고 각 값은 해당 Handler 인스턴스를 구성하는 방법을 설명하는 딕셔너리입니다.

  구성 딕셔너리는 다음 키에서 검색합니다:

  - `class` (필수). 이것은 처리기 클래스의 완전히 정규화된 이름입니다.
  - `level` (선택). 처리기의 수준.
  - `formatter` (선택). 이 처리기의 포매터의 id.
  - `filters` (선택). 이 처리기의 필터의 id의 리스트.

    버전 3.11에서 변경: `filters` can take filter instances in addition to ids.

  모든 다른 키는, 처리기의 생성자에 키워드 인자로 전달됩니다. 예를 들어, 다음과 같이 주어진 조각에서:

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

  id가 `console` 인 처리기는 `sys.stdout`를 하부 스트림으로 사용하는 `logging.StreamHandler`로 인스턴스가 만들어집니다. id가 `file` 인 처리기는 키워드 인자 `filename='logconfig.log'`,

maxBytes=1024, backupCount=3를 사용하여 *logging.handlers.RotatingFileHandler*로 인스턴스가 만들어집니다.

- *loggers* - 해당 값은 딕셔너리인데, 각 키가 로거 이름이고 각 값은 해당 Logger 인스턴스를 구성하는 방법을 설명하는 딕셔너리입니다.

  구성 딕셔너리는 다음 키에서 검색합니다:

  - `level` (선택). 로거의 수준.
  - `propagate` (선택). 로거의 전파(propagation) 설정.
  - `filters` (선택). 이 로거의 필터의 id의 리스트

    버전 3.11에서 변경: `filters` can take filter instances in addition to ids.
  - `handlers` (선택). 이 로거의 처리기의 id의 리스트.

  지정된 로거는 지정된 수준, 전파, 필터와 처리기에 따라 구성됩니다.

- *root* - 루트 로거에 대한 구성입니다. `propagate` 설정을 적용할 수 없다는 점을 제외하고 구성 처리는 모든 로거와 같습니다.
- *incremental* - 구성을 기존 구성의 증분으로 해석할지 여부. 이 값의 기본값은 `False`이며, 이는 지정된 구성이 기존 구성을 기존 *fileConfig()* API에서 사용된 것과 같은 의미로 대체 함을 뜻합니다.

  지정된 값이 `True`이면, 증분 구성 절에서 설명하는 대로 구성이 처리됩니다.

- *disable_existing_loggers* - 기존의 루트가 아닌 로거를 비활성화할지 여부. 이 설정은 *fileConfig()*의 같은 이름의 매개 변수를 반영합니다. 없으면, 이 매개 변수의 기본값은 `True`입니다. *incremental*이 `True`이면 이 값은 무시됩니다.

증분 구성

증분 구성에 완벽한 유연성을 제공하기는 어렵습니다. 예를 들어, 필터와 포매터와 같은 객체는 익명이므로, 일단 구성이 설정되면, 이러한 익명 객체를 참조하여 구성을 보강할 수 없습니다.

또한, 일단 구성이 설정되면, 실행 시간에 로거, 처리기, 필터, 포매터의 객체 그래프를 임의로 변경해야 할 강력한 사례는 없습니다; 로거와 처리기의 상세도는 단지 수준(과, loggers에서는 전파 플래그)을 설정하여 제어할 수 있습니다. 객체 그래프를 임의로 안전하게 변경하는 것은 다중 스레드 환경에서 문제가 됩니다; 불가능하지는 않지만, 구현에 추가되는 복잡성을 상쇄할만한 가치가 없습니다.

따라서, 구성 딕셔너리의 `incremental` 키가 있고 `True`이면, 시스템은 `formatters`와 `filters` 항목을 완전히 무시하고 `handlers` 항목의 `level` 설정과 `loggers`와 `root` 항목의 `level`과 `propagate` 설정만 처리합니다.

구성 딕셔너리의 값을 사용하면 구성을 피클 된 딕셔너리의 형태로 네트워크를 통해 소켓 리스너로 전송할 수 있습니다. 따라서, 장기 실행 응용 프로그램의 로깅 상세도는 응용 프로그램을 중지하고 다시 시작할 필요 없이 도중에 변경될 수 있습니다.

객체 연결

스키마는 객체 그래프에서 서로 연결된 로깅 객체 집합(로거, 처리기, 포매터, 필터)을 기술합니다. 따라서, 스키마는 객체 간의 연결을 표현할 필요가 있습니다. 예를 들어, 일단 구성되면, 특정 로거가 특정 처리기에 연결된다고 합시다. 이 토론의 목적을 위해, 둘 간의 연결에서 로거는 소스를, 처리기는 대상(destination)을 나타낸다고 할 수 있습니다. 물론 구성된 객체에서 이것은 처리기에 대한 참조를 갖는 로거로 표현됩니다. 구성 딕셔너리에서, 각 대상 객체에 명확하게 식별하는 id를 부여한 다음, 소스 객체의 구성에서 그 id를 사용하여, 소스와 그 id를 갖는 대상 객체 사이에 연결이 있음을 나타냅니다.

그래서, 예를 들어, 다음 YAML 조각을 고려해보십시오:

```
formatters:
  brief:
    # id 'brief'인 포매터 구성이 여기에 옵니다
  precise:
    # id 'precise'인 포매터 구성이 여기에 옵니다
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
handlers:
  h1: #이것은 id입니다
    # id 'h1'인 처리기 구성이 여기에 옵니다
    formatter: brief
  h2: #이것은 다른 id입니다.
    # id 'h2'인 처리기 구성이 여기에 옵니다
    formatter: precise
loggers:
  foo.bar.baz:
    # 로거 'foo.bar.baz'의 다른 구성
    handlers: [h1, h2]
```

(참고: 딕셔너리에 해당하는 파이썬 소스 형식보다 약간 더 읽기 쉬우므로 여기에서 YAML을 사용했습니다.)

로거의 id는 로거로의 참조를 얻기 위해서 프로그램적으로 사용되는 로거 이름입니다, 예를 들어 `foo.bar.baz`. 포매터와 필터의 id는 임의의 문자열 값(가령 위의 `brief`, `precise`)이 될 수 있으며, 일시적이므로 구성 딕셔너리 처리에만 의미가 있고 객체 간의 연결을 결정하는 데 사용되며, 구성 호출이 완료된 후에는 어디에도 남아있지 않습니다.

위의 조각은 `foo.bar.baz`라는 로거에 두 개의 처리기가 연결되어 있어야 하며, 이 처리기들은 처리기 id `h1`과 `h2`에 의해 기술됩니다. `h1`의 포매터는 id `brief`로 기술되는 것이고, `h2`의 포매터는 id `precise`로 기술되는 것입니다.

사용자 정의 객체

스키마는 처리기, 필터 및 포매터에 대한 사용자 정의 객체를 지원합니다. (로거에는 인스턴스마다 다른 형이 필요하지 않으므로, 이 구성 스키마에는 사용자 정의 로거 클래스에 대한 지원이 없습니다.)

구성할 객체는 구성을 자세히 설명하는 딕셔너리로 시술됩니다. 어떤 곳에서는, 로깅 시스템이 객체를 어떻게 인스턴스화할지 문맥으로부터 추측할 수 있지만, 사용자 정의 객체를 인스턴스화 해야 할 때, 시스템은 이를 수행하는 방법을 알 수 없습니다. 사용자 정의 객체 인스턴스화를 위한 완벽한 유연성을 제공하기 위해, 사용자는 '팩토리' 를 제공해야 하는데, 구성 딕셔너리로 호출되고 인스턴스화 된 객체를 반환하는 콜러블입니다. 이것은 특수키 `'()'`로 제공되는 팩토리로의 절대적 임포트 경로로 표시됩니다. 다음은 구체적인 예입니다:

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
      bar: baz
      spam: 99.9
      answer: 42
```

위의 YAML 조각은 세 가지 포매터를 정의합니다. 첫 번째(id `brief`)는 지정된 포맷 문자열을 갖는 표준 *`logging.Formatter`* 인스턴스입니다. 두 번째(id `default`)는 더 긴 포맷을 가지며 명시적으로 시간 포맷을 정의하기도 하고, 이 두 포맷 문자열로 초기화된 *`logging.Formatter`*가 됩니다. 파이썬 소스 형식으로 표시하면, `brief`와 `default` 포매터는 각각 다음과 같은 구성 서브 딕셔너리를 갖습니다:

```
{
  'format' : '%(message)s'
}
```

그리고:

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

그리고, 이 딕셔너리에는 특수키 `'()'`가 포함되어 있지 않으므로, 문맥에서 인스턴스가 추론됩니다: 결과적으로, 표준 *logging.Formatter* 인스턴스가 만들어집니다. 세 번째 포매터(id `custom`)에 대한 구성 서브 딕셔너리는 다음과 같습니다:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42
}
```

여기에는 특수키 `'()'`가 포함되어 있는데, 사용자 정의 인스턴스가 필요하다는 뜻입니다. 이때, 지정된 팩토리 콜러블이 사용됩니다. 그것이 실제 콜러블이면 직접 사용됩니다 - 그렇지 않고, (예에서와같이) 문자열을 지정하면 일반적인 임포트 메커니즘을 사용하여 실제 콜러블을 얻습니다. 콜러블은 구성 서브 딕셔너리의 **나머지** 항목을 키워드 인자로 호출됩니다. 위의 예제에서, id가 `custom`인 포매터는 다음과 같은 호출이 반환한다고 가정합니다:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

> **경고**
>
> The values for keys such as `bar`, `spam` and `answer` in the above example should not be configuration dictionaries or references such as `cfg://foo` or `ext://bar`, because they will not be processed by the configuration machinery, but passed to the callable as-is.

`'()'` 키가 유효한 키워드 매개 변수 이름이 아니라서 특수키로 사용되었습니다. 그러므로 호출에 사용되는 키워드 인자의 이름과 충돌하지 않습니다. `'()'`는 해당 값이 콜러블이라는 표시로도 기능합니다.

버전 3.11에서 변경: The `filters` member of `handlers` and `loggers` can take filter instances in addition to ids.

You can also specify a special key `'.'` whose value is a mapping of attribute names to values. If found, the specified attributes will be set on the user-defined object before it is returned. Thus, with the following configuration:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42,
  '.' {
    'foo': 'bar',
    'baz': 'bozz'
  }
}
```

the returned formatter will have attribute `foo` set to `'bar'` and attribute `baz` set to `'bozz'`.

> **경고**
>
> The values for attributes such as `foo` and `baz` in the above example should not be configuration dictionaries or references such as `cfg://foo` or `ext://bar`, because they will not be processed by the configuration machinery, but set as attribute values as-is.

처리기 구성 순서

Handlers are configured in alphabetical order of their keys, and a configured handler replaces the configuration dictionary in (a working copy of) the `handlers` dictionary in the schema. If you use a construct such as `cfg://handlers.foo`, then initially `handlers['foo']` points to the configuration dictionary for the handler named `foo`, and later (once that handler has been configured) it points to the configured handler instance. Thus, `cfg://handlers.foo` could resolve to either a dictionary or a handler instance. In general, it is wise to name handlers in a way such that dependent handlers are configured *after* any handlers they depend on; that allows something like `cfg://handlers.foo` to be used in configuring a handler that depends on handler `foo`. If that dependent handler were named `bar`, problems would result, because the configuration of `bar` would be attempted before that of `foo`, and `foo` would not yet have been configured. However, if the dependent handler were named `foobar`, it would be configured after `foo`, with the result that `cfg://handlers.foo` would resolve to configured handler `foo`, and not its configuration dictionary.

외부 객체에 대한 액세스

구성에서 구성 외부의 객체를 참조해야 하는 경우가 있습니다, 예를 들어 `sys.stderr`. 구성 딕셔너리가 파이썬 코드를 사용하여 만들어질 때는 간단하지만, 구성이 텍스트 파일(예를 들어, JSON, YAML)을 통해 제공될 때 문제가 발생합니다. 텍스트 파일에서는, `sys.stderr`를 리터럴 문자열 `'sys.stderr'`과 구별하는 표준 방법이 없습니다. 이 구별을 쉽게 하기 위해, 구성 시스템은 문자열 값에서 특정 접두사를 찾아 특수하게 처리합니다. 예를 들어, 리터럴 문자열 `'ext://sys.stderr'`이 구성에서 값으로 제공되면, `ext://`는 제거되고 값의 나머지 부분을 일반 임포트 메커니즘을 사용하여 처리합니다.

이러한 접두사의 처리는 프로토콜 처리와 유사한 방식으로 수행됩니다: 정규식 `^(?P<prefix>[a-z]+)://(?P<suffix>.*)$`와 일치하는 접두사를 찾는 일반 메커니즘이 있습니다. `prefix`가 인식되면 `suffix`는 접두사 종속적 방식으로 처리되고 처리 결과가 문자열 값을 대체합니다. 접두사가 인식되지 않으면, 문자열 값은 그대로 남습니다.

내부 객체에 대한 액세스

외부 객체뿐만 아니라, 때로 구성에 있는 객체를 참조할 필요도 있습니다. 이것은 구성 시스템이 알고 있는 것들에 대해 묵시적으로 수행됩니다. 예를 들어, 로거나 처리기의 `level`에 대한 문자열 값 `'DEBUG'`은 자동으로 값 `logging.DEBUG`으로 변환되고, `handlers`, `filters` 및 `formatter` 항목은 객체 id를 받아서 적절한 대상 객체로 결정합니다.

하지만, *logging* 모듈에 알려지지 않은 사용자 정의 객체에는 더욱 일반적인 메커니즘이 필요합니다. 예를 들어, 위임할 다른 처리기인 `target` 인자를 취하는 *logging.handlers.MemoryHandler*를 고려해봅시다. 시스템이 이미 이 클래스에 대해 알고 있으므로, 구성에서, 주어진 `target`은 단지 관련 target 처리기의 객체 id이기만 하면 되며, 시스템은 id로부터 처리기를 결정합니다. 그러나 사용자가 `alternate` 처리기를 갖는 `my.package.MyHandler`를 정의하면, 구성 시스템은 `alternate`가 처리기를 참조한다는 것을 알 수 없습니다. 이 문제를 해결하기 위해, 일반 결정 시스템은 사용자가 다음과 같이 지정할 수 있게 합니다:

```
handlers:
  file:
    # 파일 처리기의 구성은 여기에 옵니다

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

리터럴 문자열 `'cfg://handlers.file'`은 `ext://` 접두사가 있는 문자열과 비슷하게 결정되지만, 임포트 이름 공간이 아닌 구성 자체를 조회합니다. 이 메커니즘은 `str.format`에서 제공하는 것과 유사한 방식으로 점이나 인덱스로 액세스하는 것을 허락합니다. 따라서, 구성에서 다음과 같은 조각이 주어질 때:

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

문자열 'cfg://handlers'는 키 handlers의 딕셔너리로 결정되고, 문자열 'cfg://handlers.email은 handlers 딕셔너리에 있는 키 email의 딕셔너리로 결정됩니다, 등등. 문자열 'cfg://handlers.email.toaddrs[1]은 'dev_team@domain.tld'로 결정되고 문자열 'cfg://handlers.email.toaddrs[0]'은 값 'support_team@domain.tld'로 결정됩니다. subject 값은 'cfg://handlers.email.subject'나 동등하게 'cfg://handlers.email[subject]'를 사용하여 액세스할 수 있습니다. 후자의 형식은 키에 공백이나 영숫자가 아닌 문자가 포함되어있을 때만 필요합니다. 키에는 문자 [와] 가 허용되지 않음에 유의하십시오. 인덱스값이 십진수로만 구성되면, 해당 정숫값을 사용하여 액세스가 시도되고, 필요하면 문자열 값으로 다시 시도합니다.

문자열 cfg://handlers.myhandler.mykey.123 이 주어지면, config_dict['handlers']['myhandler']['mykey']['123']으로 변환됩니다. 문자열이 cfg://handlers.myhandler.mykey[123]로 지정되면, 시스템은 config_dict['handlers']['myhandler']['mykey'][123]에서 값을 가져오려고 시도하고, 실패하면 config_dict['handlers']['myhandler']['mykey']['123']으로 폴백합니다.

임포트 결정과 사용자 정의 임포터

임포트 결정은, 기본적으로, 임포트 하는데 내장 *__import__()* 함수를 사용합니다. 이것을 자신의 임포트 메커니즘으로 바꾸고 싶을 수 있습니다: 그렇다면, DictConfigurator나 그것의 슈퍼 클래스 (BaseConfigurator 클래스)의 importer 어트리뷰트를 바꿀 수 있습니다. 그러나, 함수가 클래스에서 디스크립터를 통해 액세스 되는 방식 때문에 주의해야 합니다. 파이썬 콜러블을 사용하여 임포트를 수행하려고 하고, 인스턴스 수준이 아닌 클래스 수준에서 정의하려고 한다면, *staticmethod()*로 감쌀 필요가 있습니다. 예를 들면:

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

구성자 *instance*에서 임포트 콜러블을 설정한다면, *staticmethod()*로 감쌀 필요가 없습니다.

Configuring QueueHandler and QueueListener

If you want to configure a *QueueHandler*, noting that this is normally used in conjunction with a *QueueListener*, you can configure both together. After the configuration, the QueueListener instance will be available as the *listener* attribute of the created handler, and that in turn will be available to you using *getHandlerByName()* and passing the name you have used for the QueueHandler in your configuration. The dictionary schema for configuring the pair is shown in the example YAML snippet below.

```
handlers:
  qhand:
    class: logging.handlers.QueueHandler
    queue: my.module.queue_factory
    listener: my.package.CustomListener
    handlers:
      - hand_name_1
      - hand_name_2
      ...
```

The queue and listener keys are optional.

If the queue key is present, the corresponding value can be one of the following:

- An object implementing the *Queue.put_nowait* and *Queue.get* public API. For instance, this may be

an actual instance of *queue.Queue* or a subclass thereof, or a proxy obtained by *multiprocessing.managers.SyncManager.Queue()*.

This is of course only possible if you are constructing or modifying the configuration dictionary in code.

- A string that resolves to a callable which, when called with no arguments, returns the queue instance to use. That callable could be a *queue.Queue* subclass or a function which returns a suitable queue instance, such as `my.module.queue_factory()`.
- A dict with a `'()'` key which is constructed in the usual way as discussed in 사용자 정의 객체. The result of this construction should be a *queue.Queue* instance.

If the `queue` key is absent, a standard unbounded *queue.Queue* instance is created and used.

If the `listener` key is present, the corresponding value can be one of the following:

- A subclass of *logging.handlers.QueueListener*. This is of course only possible if you are constructing or modifying the configuration dictionary in code.
- A string which resolves to a class which is a subclass of `QueueListener`, such as `'my.package.CustomListener'`.
- A dict with a `'()'` key which is constructed in the usual way as discussed in 사용자 정의 객체. The result of this construction should be a callable with the same signature as the `QueueListener` initializer.

If the `listener` key is absent, *logging.handlers.QueueListener* is used.

The values under the `handlers` key are the names of other handlers in the configuration (not shown in the above snippet) which will be passed to the queue listener.

Any custom queue handler and listener classes will need to be defined with the same initialization signatures as *QueueHandler* and *QueueListener*.

Added in version 3.12.

16.5.4 구성 파일 형식

*fileConfig()*이 이해하는 구성 파일 형식은 *configparser* 기능을 기반으로 합니다. 파일에는 `[loggers]`, `[handlers]` 및 `[formatters]`라는 섹션이 있어야 하며, 이 섹션에서는 파일에 정의된 각 유형의 엔티티를 이름으로 식별합니다. 이러한 엔티티마다 해당 엔티티 구성 방법을 식별하는 별도의 섹션이 있습니다. 따라서, `[loggers]` 섹션에서 `log01`이라고 이름 붙은 로거에 대해, 관련 구성 세부 사항은 `[logger_log01]` 섹션에 담깁니다. 마찬가지로, `[handlers]` 섹션에서 `hand01`이라고 부르는 처리기는 `[handler_hand01]`이라는 섹션에 구성이 담기고, `[formatters]` 섹션에서 `form01`이라고 부르는 포매터는 `[formatter_form01]`이라는 섹션에서 구성이 지정됩니다. 루트 로거 구성은 `[logger_root]`라는 섹션에서 지정해야 합니다.

> **참고**
>
> *fileConfig()* API는 *dictConfig()* API보다 오래되었으며 로깅의 특정 측면을 다루는 기능을 제공하지 않습니다. 예를 들어, *fileConfig()*를 사용해서는 간단한 정수 수준을 넘어서는 메시지 필터링을 제공하는 *Filter* 객체를 구성할 수 없습니다. 로깅 구성에 *Filter* 인스턴스가 필요하면, *dictConfig()*를 사용해야 합니다. 향후 구성 기능의 개선은 *dictConfig()*에 추가될 것임에 유의하십시오. 따라서, 편리할 때 이 새로운 API로 전환하는 것을 고려해 볼 가치가 있습니다.

파일에 있는 이 절의 예는 아래에 나와 있습니다.

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

루트 로거는 수준과 처리기 목록을 지정해야 합니다. 루트 로거 섹션의 예가 아래에 나와 있습니다.

```
[logger_root]
level=NOTSET
handlers=hand01
```

`level` 항목은 `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL` 또는 `NOTSET` 중 하나일 수 있습니다. 루트 로거에서만, `NOTSET`는 모든 메시지가 로그 됨을 의미합니다. 수준 값은 `logging` 패키지의 이름 공간 컨텍스트에서 평가됩니다.

`handlers` 항목은 `[handlers]` 섹션에 나타나야 하는 처리기 이름의 쉼표로 구분된 목록입니다. 이 이름들은 `[handlers]` 섹션에 나타나야 하며, 구성 파일에 해당 섹션이 있어야 합니다.

루트 로거가 아닌 로거의 경우, 몇 가지 추가 정보가 필요합니다. 이것은 다음 예제가 보여줍니다.

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

`level`과 `handlers` 항목은 루트 로거에서처럼 해석됩니다. 단, 루트가 아닌 로거의 수준이 `NOTSET`로 지정되면, 시스템은 로거의 유효 수준을 판별하기 위해 상위 계층 로거를 참조합니다. `propagate` 항목은 메시지가 이 로거로부터 더 높은 로거 계층의 처리기로 전파되어야 함을 나타내려면 1로 설정되고, 메시지가 계층 위의 처리기로 전달되지 **않음**을 나타내려면 0으로 설정됩니다. `qualname` 항목은 로거의 계층적 채널 이름, 즉 응용 프로그램에서 로거를 가져오는 데 사용되는 이름입니다.

처리기 구성을 지정하는 섹션은 다음과 같이 예시됩니다.

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

`class` 항목은 (`logging` 패키지의 이름 공간에서 *`eval()`*로 결정되는) 처리기의 클래스를 나타냅니다. `level`은 로거에서처럼 해석되며, `NOTSET`은 '모든 것을 로깅' 을 의미합니다.

`formatter` 항목은 이 처리기의 포매터의 키 이름을 나타냅니다. 비어 있으면, 기본 포매터(`logging._defaultFormatter`)가 사용됩니다. 이름이 지정되면, `[formatters]` 섹션에 나타나야 하며 구성 파일에 해당 섹션이 있어야 합니다.

`args` 항목은, `logging` 패키지의 이름 공간 컨텍스트에서 평가될 때, 처리기 클래스의 생성자에 대한 인자 목록입니다. 일반적인 항목 작성 방법을 보려면, 관련 처리기의 생성자나 아래 예제를 참조하십시오. 제공되지 않으면, 기본값은 `()` 입니다.

선택적 `kwargs` 항목은, `logging` 패키지의 이름 공간 컨텍스트에서 평가될 때, 처리기 클래스의 생성자에 대한 키워드 인자 딕셔너리입니다. 제공되지 않으면, 기본값은 `{}` 입니다.

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')
kwargs={'timeout': 10.0}

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
kwargs={'secure': True}
```

포매터 구성을 지정하는 섹션은 다음과 같이 예시됩니다.

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s %(customfield)s
datefmt=
style=%
validate=True
defaults={'customfield': 'defaultvalue'}
class=logging.Formatter
```

The arguments for the formatter configuration are the same as the keys in the dictionary schema *formatters section*.

`defaults` 항목은, `logging` 패키지의 이름 공간 컨텍스트에서 평가될 때, 사용자 정의 포매팅 필드에 대한 기본 값의 딕셔너리입니다. 제공되지 않으면, 기본값은 `None` 입니다.

참고

위에서 설명한 대로 `eval()`를 사용하기 때문에, `listen()`을 사용하여 소켓을 통해 구성을 보내고 받을 때 발생할 수 있는 잠재적인 보안 위험이 있습니다. 위험은 상호 신뢰가 없는 여러 사용자가 같은 기계에서 코드를 실행할 때로 제한됩니다; 자세한 내용은 `listen()` 설명서를 참조하십시오.

더 보기

모듈 ***logging***
logging 모듈에 관한 API 레퍼런스.

모듈 ***logging.handlers***
logging 모듈에 포함된 유용한 처리기.

16.6 logging.handlers — 로깅 처리기

소스 코드: Lib/logging/handlers.py

Important

이 페이지에는 레퍼런스 정보만 있습니다. 자습서는 다음을 참조하십시오

- 기초 자습서
- 고급 자습서
- 로깅 요리책

다음과 같은 유용한 처리기가 패키지에서 제공됩니다. 3개의 처리기(`StreamHandler`, `FileHandler`, `NullHandler`)는 실제로는 `logging` 모듈 자체에 정의되어 있지만, 다른 처리기들과 함께 여기에서 설명합니다.

16.6.1 StreamHandler

핵심 `logging` 패키지에 있는 `StreamHandler` 클래스는 *sys.stdout*, *sys.stderr* 또는 임의의 파일류 객체(또는 더 정확하게, `write()`와 `flush()` 메서드를 지원하는 모든 객체)와 같은 스트림으로 로깅 출력을 보냅니다.

class logging.**StreamHandler**(*stream=None*)

`StreamHandler` 클래스의 새로운 인스턴스를 반환합니다. *stream* 이 지정되면, 인스턴스는 그것을 로그 출력용으로 사용합니다; 그렇지 않으면, *sys.stderr* 이 사용됩니다.

emit(*record*)

포매터가 지정되면, 레코드를 포맷하는 데 사용됩니다. 그런 다음 레코드는 `terminator`를 붙여 스트림에 기록됩니다. 예외 정보가 있으면, `traceback.print_exception()`을 사용하여 포맷한 후 스트림에 덧붙입니다.

flush()

스트림의 `flush()` 메서드를 호출해서 플러시 합니다. `close()` 메서드는 `Handler`에서 상속되고, 출력이 없으므로, 명시적 `flush()` 호출이 필요할 수도 있습니다.

setStream(*stream*)

지정한 값이 현재 값과 다르면, 인스턴스의 스트림을 지정된 값으로 설정합니다. 새 스트림이 설정되기 전에 이전 스트림이 플러시 됩니다.

매개변수
stream – 처리기가 사용할 스트림.

반환
스트림이 변경되면 이전 스트림, 그렇지 않으면 `None`.

Added in version 3.7.

terminator

포맷된 레코드를 스트림에 쓸 때 종결자로 사용되는 문자열. 기본값은 `'\n'` 입니다.

줄 바꿈 종료를 원하지 않으면, 처리기 인스턴스의 `terminator` 어트리뷰트를 빈 문자열로 설정할 수 있습니다.

이전 버전에서는, 종결자가 `'\n'` 으로 하드 코딩되었습니다.

Added in version 3.2.

16.6.2 FileHandler

핵심 `logging` 패키지에 있는 `FileHandler` 클래스는 로깅 출력을 디스크 파일로 보냅니다. `StreamHandler` 에서 출력 기능을 상속받습니다.

class logging.**FileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

`FileHandler` 클래스의 새로운 인스턴스를 반환합니다. 지정된 파일이 열리고 로깅을 위한 스트림으로 사용됩니다. *mode* 가 지정되지 않으면, `'a'` 가 사용됩니다. *encoding* 이 `None` 이 아니면, *encoding* 을 사용하여 파일을 엽니다. *delay* 가 참이면, 파일 열기는 `emit()` 의 첫 번째 호출이 있을 때까지 연기됩니다. 기본적으로, 파일은 제한 없이 커집니다. *errors* 가 지정되면, 인코딩 에러 처리 방법을 결정합니다.

버전 3.6에서 변경: 문자열 값뿐만 아니라, `Path` 객체도 *filename* 인자로 허용됩니다.

버전 3.9에서 변경: *errors* 매개 변수가 추가되었습니다.

close()

파일을 닫습니다.

emit(*record*)

레코드를 파일에 출력합니다.

Note that if the file was closed due to logging shutdown at exit and the file mode is 'w' , the record will not be emitted (see bpo-42378).

16.6.3 NullHandler

Added in version 3.1.

핵심 `logging` 패키지에 있는 `NullHandler` 클래스는 포맷이나 출력을 일절 하지 않습니다. 기본적으로 라이브러리 개발자가 사용하는 'no-op' 처리기입니다.

class logging.**NullHandler**

`NullHandler` 클래스의 새로운 인스턴스를 반환합니다.

emit(*record*)

이 메서드는 아무것도 하지 않습니다.

handle(*record*)

이 메서드는 아무것도 하지 않습니다.

createLock()

액세스를 직렬화해야 하는 하부 I/O가 없으므로, 이 메서드는 록으로 `None` 을 반환합니다.

`NullHandler` 사용법에 대한 더 많은 정보는 library-config를 참조하세요.

16.6.4 WatchedFileHandler

logging.handlers 모듈에 있는 *WatchedFileHandler* 클래스는 로깅 중인 파일을 감시하는 FileHandler 입니다. 파일이 변경되면, 닫은 후에 같은 이름의 파일을 다시 엽니다.

로그 파일 회전을 수행하는 *newsyslog* 나 *logrotate* 와 같은 프로그램의 사용으로 인해 파일이 변경될 수 있습니다. 유닉스/리눅스에서 사용하기 위한 이 처리기는 마지막 출력 이후에 파일이 변경되었는지 감시합니다. (파일의 장치나 inode가 변경되면 파일이 변경된 것으로 간주합니다.) 파일이 변경되면, 이전 파일 스트림이 닫히고, 새 스트림을 얻기 위해 파일을 엽니다.

이 처리기는 윈도우에서 사용하기에 적합하지 않습니다. 윈도우에서는 열린 로그 파일을 이동하거나 이름을 변경할 수 없어서 - logging은 파일을 배타적 록으로 엽니다 - 이런 처리기가 필요하지 않기 때문입니다. 또한 *ST_INO* 는 윈도우에서 지원되지 않습니다; *stat()* 는 항상 이 값에 대해 0을 반환합니다.

class logging.handlers.**WatchedFileHandler**(*filename, mode='a', encoding=None, delay=False, errors=None*)

FileHandler 클래스의 새로운 인스턴스를 반환합니다. 지정된 파일이 열리고 로깅을 위한 스트림으로 사용됩니다. *mode* 가 지정되지 않으면, 'a' 가 사용됩니다. *encoding* 이 None 이 아니면, *encoding* 을 사용하여 파일을 엽니다. *delay* 가 참이면, 파일 열기는 *emit()* 의 첫 번째 호출이 있을 때까지 연기됩니다. 기본적으로, 파일은 제한 없이 커집니다. *errors* 가 제공되면, 인코딩 에러 처리 방법을 결정합니다.

버전 3.6에서 변경: 문자열 값뿐만 아니라, *Path* 객체도 *filename* 인자로 허용됩니다.

버전 3.9에서 변경: *errors* 매개 변수가 추가되었습니다.

reopenIfNeeded()

파일이 변경되었는지 확인합니다. 그렇다면, 기존 스트림을 플러시 한 후 닫고, 파일을 다시 엽니다. 일반적으로 레코드를 파일로 출력하기 전에 수행합니다.

Added in version 3.6.

emit(*record*)

레코드를 파일에 출력하지만, 파일이 변경되었을 때 다시 열기 위해 *reopenIfNeeded()* 를 먼저 호출합니다.

16.6.5 BaseRotatingHandler

logging.handlers 모듈에 있는 *BaseRotatingHandler* 클래스는 회전하는 파일 처리기들 (*RotatingFileHandler*와 *TimedRotatingFileHandler*)의 베이스 클래스입니다. 이 클래스의 인스턴스를 만들 필요는 없지만, 재정의가 필요할 수 있는 어트리뷰트와 메서드가 있습니다.

class logging.handlers.**BaseRotatingHandler**(*filename, mode, encoding=None, delay=False, errors=None*)

매개 변수는 FileHandler 와 같습니다. 어트리뷰트는 다음과 같습니다:

namer

이 어트리뷰트가 콜러블로 설정되면, *rotation_filename()* 메서드는 이 콜러블에 위임합니다. 콜러블로 전달되는 매개 변수는 *rotation_filename()* 로 전달되는 것입니다.

> **참고**
>
> namer 함수는 롤오버 중에 꽤 자주 호출되므로, 가능한 한 간단하고 빨라야 합니다. 또한, 주어진 입력에 대해 매번 같은 출력을 반환해야 합니다, 그렇지 않으면 롤오버 동작이 예상대로 작동하지 않을 수 있습니다.
>
> It' s also worth noting that care should be taken when using a namer to preserve certain attributes in the filename which are used during rotation. For example, *RotatingFileHandler* expects to have a set of log files whose names contain successive integers, so that rotation works as expected, and *TimedRotatingFileHandler* deletes old log files (based on the backupCount parameter passed to the handler' s initializer) by determining the oldest files to delete. For this to happen, the filenames should be sortable using the date/time portion of the filename, and a namer needs to respect

this. (If a namer is wanted that doesn' t respect this scheme, it will need to be used in a subclass of `TimedRotatingFileHandler` which overrides the `getFilesToDelete()` method to fit in with the custom naming scheme.)

Added in version 3.3.

rotator

이 어트리뷰트가 콜러블로 설정되면, `rotate()` 메서드는 이 콜러블에 위임합니다. 콜러블로 전달되는 매개 변수는 `rotate()`로 전달되는 것입니다.

Added in version 3.3.

rotation_filename(*default_name*)

회전할 때 로그 파일의 파일명을 수정합니다.

사용자 정의 파일명을 제공할 수 있게 하려고 제공됩니다.

기본 구현은 처리기의 'namer' 어트리뷰트를(콜러블이라면) 호출하는데, 기본 이름을 전달합니다. 어트리뷰트가 콜러블이 아니면 (기본값은 `None` 입니다), 이름은 변경되지 않고 반환됩니다.

매개변수

default_name – 로그 파일의 기본 이름.

Added in version 3.3.

rotate(*source*, *dest*)

회전할 때, 현재 로그를 회전합니다.

기본 구현은 처리기의 'rotator' 어트리뷰트를(콜러블이라면) 호출하는데, source와 dest 인자를 전달합니다. 어트리뷰트가 콜러블이 아니면 (기본값은 `None` 입니다), source를 dest 로 단순히 이름을 바꿉니다.

매개변수

- **source** – 소스 파일명. 이것은 일반적으로 기본 파일명입니다, 예를 들어 'test.log' .
- **dest** – 대상 파일명. 이것은 일반적으로 소스가 회전되는 곳입니다, 예를 들어 'test.log.1' .

Added in version 3.3.

어트리뷰트가 존재하는 이유는 서브 클래싱해야 할 필요를 줄이는 것입니다 - `RotatingFileHandler`와 `TimedRotatingFileHandler` 의 인스턴스에 같은 콜러블을 사용할 수 있습니다. namer 나 rotator 콜러블이 예외를 발생시키면, `emit()` 동안 발생하는 다른 예외와 같은 방식으로 처리됩니다, 즉 처리기의 `handleError()` 메서드를 통해.

회전 처리를 더 크게 변경해야 하면, 메서드를 재정의할 수 있습니다.

예는 cookbook-rotator-namer를 보십시오.

16.6.6 RotatingFileHandler

`logging.handlers` 모듈에 있는 `RotatingFileHandler` 클래스는 디스크 로그 파일 회전을 지원합니다.

class logging.handlers.**RotatingFileHandler**(*filename*, *mode='a'*, *maxBytes=0*, *backupCount=0*, *encoding=None*, *delay=False*, *errors=None*)

`RotatingFileHandler` 클래스의 새로운 인스턴스를 반환합니다. 지정된 파일이 열리고 로깅을 위한 스트림으로 사용됩니다. *mode* 가 지정되지 않으면, `'a'` 가 사용됩니다. *encoding* 이 `None` 이 아니면, *encoding*을 사용하여 파일을 엽니다. *delay* 가 참이면, 파일 열기는 `emit()`의 첫 번째 호출이 있을 때까지 연기됩니다. 기본적으로, 파일은 제한 없이 커집니다. *errors*가 제공되면, 인코딩 에러 처리 방법을 결정합니다.

미리 결정된 크기에서 파일을 롤오버 (*rollover*) 하기 위해 *maxBytes* 와 *backupCount* 값을 사용할 수 있습니다. 크기가 초과하려고 할 때, 파일이 닫히고 출력을 위해 새 파일이 조용히 열립니다. 롤오버는

현재 로그 파일이 거의 *maxBytes* 길이일 때마다 발생합니다; 그러나 *maxBytes* 나 *backupCount* 가 0이면 롤오버가 발생하지 않으므로, 일반적으로 *backupCount* 를 1 이상으로 설정하고, 0이 아닌 *maxBytes* 를 사용하기를 원할 겁니다. *backupCount* 가 0이 아니면, 시스템은 파일명에 확장자 '.1', '.2' 등을 추가하여 지난 로그 파일을 저장합니다. 예를 들어, *backupCount* 가 5이고 기본 파일명이 `app.log` 면, `app.log`, `app.log.1`, `app.log.2`부터 `app.log.5` 까지의 파일을 얻게 됩니다. 기록되는 파일은 항상 `app.log` 입니다. 이 파일이 채워지면, 닫히고 `app.log.1` 로 이름이 변경됩니다, 그리고 파일 `app.log.1`, `app.log.2` 등이 존재하면, 이것들도 각기 `app.log.2`, `app.log.3` 등으로 이름이 변경됩니다.

버전 3.6에서 변경: 문자열 값뿐만 아니라, `Path` 객체도 *filename* 인자로 허용됩니다.

버전 3.9에서 변경: *errors* 매개 변수가 추가되었습니다.

doRollover()

위에서 설명한 대로 롤오버를 수행합니다.

emit(*record*)

앞에서 설명한 대로 롤오버를 처리하면서, 파일에 레코드를 출력합니다.

shouldRollover(*record*)

See if the supplied record would cause the file to exceed the configured size limit.

16.6.7 TimedRotatingFileHandler

`logging.handlers` 모듈에 있는 `TimedRotatingFileHandler` 클래스는 특정 시간 간격의 디스크 로그 파일 회전을 지원합니다.

class logging.handlers.**TimedRotatingFileHandler**(*filename, when='h', interval=1, backupCount=0, encoding=None, delay=False, utc=False, atTime=None, errors=None*)

`TimedRotatingFileHandler` 클래스의 새로운 인스턴스를 반환합니다. 지정된 파일이 열리고 로깅을 위한 스트림으로 사용됩니다. 회전 시 파일명 접미사도 설정합니다. *when* 과 *interval* 에 따라 회전이 일어납니다.

when 을 사용하여 *interval* 의 유형을 지정할 수 있습니다. 가능한 값의 목록은 아래와 같습니다. 대소문자를 구분하지 않는다는 것에 유의하세요.

값	interval의 유형	*atTime* 이 사용되는지와 사용되는 방식
'S'	초	무시됩니다
'M'	분	무시됩니다
'H'	시간	무시됩니다
'D'	일	무시됩니다
'W0'-'W6	요일 (0=월요일)	최초 롤오버 시간을 계산하는 데 사용됩니다
'midnigh	*atTime* 을 지정하지 않으면 자정에, 그렇지 않으면 *atTime* 에 롤오버 합니다	최초 롤오버 시간을 계산하는 데 사용됩니다

요일 기반 회전을 사용할 때, 월요일은 'W0', 화요일은 'W1', 등등 일요일은 'W6' 까지 지정하십시오. 이 경우, *interval* 에 전달된 값은 사용되지 않습니다.

시스템은 파일명에 확장자를 추가하여 지난 로그 파일을 저장합니다. 확장자는 날짜와 시간 기반이며, 롤오버 간격에 따라 strftime 형식 `%Y-%m-%d_%H-%M-%S` 이나 그 앞부분을 사용합니다.

다음 롤오버 시간을 처음 계산할 때 (처리기가 만들어질 때), 기존 로그 파일의 마지막 수정 시간 또는 (없으면) 현재 시각이 다음 회전이 발생할 때를 계산하는 데 사용됩니다.

utc 인자가 참이면, UTC 시간이 사용됩니다; 그렇지 않으면 현지 시간이 사용됩니다.

backupCount 가 0이 아니면, 최대 *backupCount* 개의 파일이 보관되고, 롤오버가 발생할 때 더 많은 파일이 생성되면 가장 오래된 파일이 삭제됩니다. 삭제 논리는 interval을 사용하여 삭제할 파일을 결정하므로, interval을 변경하면 오래된 파일이 남아있을 수 있습니다.

delay 가 참이면, 파일 열기는 `emit()` 에 대한 첫 번째 호출까지 지연됩니다.

atTime 이 `None` 이 아니면, 반드시 `datetime.time` 인스턴스여야 하는데, 롤오버가 "자정에" 또는 "특정 요일에" 발생하도록 설정된 경우에 롤오버가 발생하는 시간을 지정합니다. 이 경우, *atTime* 값은 최초 롤오버를 계산하는 데 사용되며, 이후 롤오버는 일반적인 간격 계산을 통해 계산됩니다.

errors 가 지정되면, 인코딩 에러 처리 방법을 결정하는 데 사용됩니다.

> **참고**
>
> 최초 롤오버 시간의 계산은 처리기가 초기화될 때 수행됩니다. 후속 롤오버 시간 계산은 롤오버가 발생하는 경우에만 수행되며, 롤오버는 출력을 내보낼 때만 발생합니다. 이것을 명심하지 않으면, 혼란이 생길 수 있습니다. 예를 들어, "매분" 간격을 설정하면, 이것이 항상 1분 간격의 (파일명을 갖는) 로그 파일들을 보게 된다는 것을 뜻하지는 않습니다; 응용 프로그램을 실행하는 동안, 로그 출력이 1분당 한 번보다 더 자주 발생하면, 1분 간격의 로그 파일을 볼 것으로 예상할 수 있습니다. 반면, (가령) 로깅 메시지가 5분마다 한 번만 출력되면, 출력이 없는 (따라서 롤오버가 없는) 분에 해당하는 파일 시간의 틈이 생깁니다.

버전 3.4에서 변경: *atTime* 매개 변수가 추가되었습니다.

버전 3.6에서 변경: 문자열 값뿐만 아니라, `Path` 객체도 *filename* 인자로 허용됩니다.

버전 3.9에서 변경: *errors* 매개 변수가 추가되었습니다.

doRollover()

위에서 설명한 대로 롤오버를 수행합니다.

emit(*record*)

위에서 설명한 대로 롤오버를 처리하면서, 파일에 레코드를 출력합니다.

getFilesToDelete()

Returns a list of filenames which should be deleted as part of rollover. These

shouldRollover(*record*)

See if enough time has passed for a rollover to occur and if it has, compute the next rollover time.

16.6.8 SocketHandler

`logging.handlers` 모듈에 있는 `SocketHandler` 클래스는 로깅 출력을 네트워크 소켓에 보냅니다. 베이스 클래스는 TCP 소켓을 사용합니다.

class logging.handlers.**SocketHandler**(*host*, *port*)

host 와 *port* 로 주어진 주소의 원격 기계와 통신하기 위한, `SocketHandler` 클래스의 새로운 인스턴스를 반환합니다.

버전 3.4에서 변경: `port` 가 `None` 으로 지정되면, `host` 의 값을 사용하여 유닉스 도메인 소켓이 만들어집니다 - 그렇지 않으면 TCP 소켓이 만들어집니다.

close()

소켓을 닫습니다.

emit()

레코드의 어트리뷰트 딕셔너리를 피클하고 바이너리 형식으로 소켓에 씁니다. 소켓에 에러가 있으면 조용히 패킷을 버립니다. 이전에 연결이 끊어졌으면, 연결을 다시 맺습니다. 수신 단에서 레코드를 `LogRecord` 로 역 피클 하려면, `makeLogRecord()` 함수를 사용하십시오.

handleError()

`emit()` 중에 발생한 에러를 처리합니다. 가장 큰 원인은 연결이 끊어지는 것입니다. 다음 이벤트에서 다시 시도할 수 있도록 소켓을 닫습니다.

makeSocket()

이것은 서브 클래스가 원하는 소켓의 정확한 유형을 정의 할 수 있게 하는 팩토리 메서드입니다. 기본 구현은 TCP 소켓(`socket.SOCK_STREAM`)을 만듭니다.

makePickle(*record*)

레코드의 어트리뷰트 딕셔너리를 바이너리 형식으로 피클하고 길이를 앞에 붙여서, 소켓을 통해 전송할 준비가 된 상태로 반환합니다. 이 연산의 세부 사항은 다음과 동등합니다:

```
data = pickle.dumps(record_attr_dict, 1)
datalen = struct.pack('>L', len(data))
return datalen + data
```

피클은 완전히 안전하지 않습니다. 보안이 염려되면, 이 메서드를 재정의하여 더욱 안전한 메커니즘을 구현할 수 있습니다. 예를 들어, HMAC를 사용하여 피클에 서명한 다음 수신 단에서 확인하거나, 수신 단에서 전역 객체의 역 피클링을 비활성화할 수 있습니다.

send(*packet*)

피클 된 바이트열 *packet* 을 소켓으로 보냅니다. 보내진 바이트열의 형식은 `makePickle()`의 설명서에 있습니다.

이 함수는 네트워크가 붐빌 때 발생할 수 있는 부분 전송을 허용합니다.

createSocket()

소켓을 만들려고 합니다; 실패 시, 지수 백 오프 알고리즘을 사용합니다. 최초 실패 시 처리기는 보내려는 메시지를 버립니다. 후속 메시지가 같은 인스턴스에 의해 처리될 때, 일정한 시간이 지날 때까지 연결을 시도하지 않습니다. 기본 파라미터를 쓸 때, 최초 지연은 1초이고, 지연 후에도 연결을 만들 수 없으면, 처리기가 최대 30초가 될 때까지 매번 지연 시간을 두 배로 늘립니다.

이 동작은 다음 처리기 어트리뷰트에 의해 제어됩니다:

- `retryStart` (최초 지연, 기본값은 1.0 초).
- `retryFactor` (배율, 기본값은 2.0).
- `retryMax` (최대 지연, 기본값은 30.0 초).

이것은, 처리기가 사용된 후에 원격 수신기가 시작되면, 메시지가 손실될 수 있음을 뜻합니다 (처리기가 지연이 경과 할 때까지 연결을 시도하지조차 않고, 지연 기간에 메시지를 조용히 버리기 때문입니다).

16.6.9 DatagramHandler

`logging.handlers` 모듈에 있는 `DatagramHandler` 클래스는 UDP 소켓을 통해 로깅 메시지를 보낼 수 있도록 `SocketHandler`를 상속합니다.

class logging.handlers.**DatagramHandler**(*host*, *port*)

host 와 *port*로 주어진 주소의 원격 기계와 통신하기 위한, `DatagramHandler` 클래스의 새로운 인스턴스를 반환합니다.

> **참고**
>
> As UDP is not a streaming protocol, there is no persistent connection between an instance of this handler and *host*. For this reason, when using a network socket, a DNS lookup might have to be made each time an event is logged, which can introduce some latency into the system. If this affects you, you can do a lookup yourself and initialize this handler using the looked-up IP address rather than the hostname.

버전 3.4에서 변경: `port`가 `None`으로 지정되면, `host`의 값을 사용하여 유닉스 도메인 소켓이 만들어집니다 - 그렇지 않으면 UDP 소켓이 만들어집니다.

emit()

레코드의 어트리뷰트 딕셔너리를 피클하고 바이너리 형식으로 소켓에 씁니다. 소켓에 에러가 있으면 조용히 패킷을 버립니다. 수신 단에서 레코드를 `LogRecord`로 역 피클 하려면, `makeLogRecord()` 함수를 사용하십시오.

makeSocket()

UDP 소켓(`socket.SOCK_DGRAM`)을 만들기 위해 `SocketHandler`의 팩토리 메서드가 여기에서 재정의되었습니다.

send(*s*)

피클 된 바이트열을 소켓으로 보냅니다. 보낸 바이트열의 형식은 `SocketHandler.makePickle()` 설명서에 있습니다.

16.6.10 SysLogHandler

`logging.handlers` 모듈에 있는 `SysLogHandler` 클래스는 원격 또는 로컬 유닉스 syslog로 로깅 메시지를 보내는 것을 지원합니다.

class logging.handlers.**SysLogHandler**(*address=('localhost', SYSLOG_UDP_PORT), facility=LOG_USER, socktype=socket.SOCK_DGRAM, timeout=None*)

Returns a new instance of the `SysLogHandler` class intended to communicate with a remote Unix machine whose address is given by *address* in the form of a `(host, port)` tuple. If *address* is not specified, `('localhost', 514)` is used. The address is used to open a socket. An alternative to providing a `(host, port)` tuple is providing an address as a string, for example '/dev/log' . In this case, a Unix domain socket is used to send the message to the syslog. If *facility* is not specified, `LOG_USER` is used. The type of socket opened depends on the *socktype* argument, which defaults to `socket.SOCK_DGRAM` and thus opens a UDP socket. To open a TCP socket (for use with the newer syslog daemons such as rsyslog), specify a value of `socket.SOCK_STREAM`. If *timeout* is specified, it sets a timeout (in seconds) for the socket operations. This can help prevent the program from hanging indefinitely if the syslog server is unreachable. By default, *timeout* is `None`, meaning no timeout is applied.

서버가 UDP 포트 514에서 수신을 기다리지 않으면, `SysLogHandler`가 작동하지 않는 것처럼 보일 수 있습니다. 이 경우, 도메인 소켓에 대해 사용해야 하는 주소를 확인하십시오 - 이는 시스템에 따라 다릅니다. 예를 들어 리눅스에서는 보통 '/dev/log' 이지만 OS/X에서는 '/var/run/syslog' 입니다. 플랫폼을 확인하고 적절한 주소를 사용해야 합니다 (응용 프로그램을 여러 플랫폼에서 실행해야 하는 경우 실행 시간에 검사를 수행해야 할 수도 있습니다). 윈도우에서는, UDP 옵션을 사용해야 합니다.

> **참고**
>
> On macOS 12.x (Monterey), Apple has changed the behaviour of their syslog daemon - it no longer listens on a domain socket. Therefore, you cannot expect `SysLogHandler` to work on this system.
>
> See gh-91070 for more information.

버전 3.2에서 변경: *socktype* 이 추가되었습니다.

버전 3.14에서 변경: *timeout* was added.

close()

원격 호스트로의 소켓을 닫습니다.

createSocket()

Tries to create a socket and, if it' s not a datagram socket, connect it to the other end. This method is called during handler initialization, but it' s not regarded as an error if the other end isn' t listening at this point - the method will be called again when emitting an event, if there is no socket at that point.

Added in version 3.11.

emit(*record*)

레코드가 포맷된 다음, syslog 서버로 전송됩니다. 예외 정보가 있으면, 서버로 보내 지지 않습니다.

버전 3.2.1에서 변경: (bpo-12168를 보세요.) 이전 버전에서, syslog 데몬으로 보낸 메시지는 NUL 바이트로 항상 종료되었는데, 이전 버전의 데몬에서 관련 사양(**RFC 5424**)에 없는데도 불구하고 NUL 종료 메시지를 요구했기 때문입니다. 최신 버전의 데몬은 NUL 바이트를 기대하지는 않지만, 있는 경우 이를 제거하고, 더 최근의 (RFC 5424와 더 가깝게 일치하는) 데몬은 NUL 바이트를 메시지 일부로 전달합니다.

이러한 모든 다른 데몬 동작에 직면하여 syslog 메시지를 더욱 쉽게 처리할 수 있도록, NUL 바이트를 추가하는 작업은 클래스 수준 어트리뷰트 `append_nul`을 사용하여 구성할 수 있게 만들었습니다. 기본값은 `True`(기존 동작 유지)이지만, 특정 인스턴스가 NUL 종결자를 추가하지 않도록 `SysLogHandler` 인스턴스에서 `False`로 설정할 수 있습니다.

버전 3.3에서 변경: (bpo-12419를 보세요.) 이전 버전에서는, 메시지 소스를 식별하는 "ident" 나 "tag" 접두사를 위한 기능이 없었습니다. 이제는 클래스 수준의 어트리뷰트를 사용하여 지정할 수 있습니다, `""`을 기본값으로 사용하여 기존 동작을 유지하지만, `SysLogHandler` 인스턴스에서 재정의하여 해당 인스턴스가 처리하는 모든 메시지에 ident를 추가하도록 할 수 있습니다. 제공된 ident는 바이트열이 아닌 텍스트여야 하며 그대로 메시지 앞에 추가됩니다.

encodePriority(*facility*, *priority*)

시설(facility)과 우선순위를 정수로 인코딩합니다. 문자열이나 정수를 전달할 수 있습니다 - 문자열이 전달되면, 내부 매핑 딕셔너리를 사용하여 정수로 변환합니다.

`LOG_` 기호 값은 *`SysLogHandler`*에 정의되고 `sys/syslog.h` 헤더 파일에 정의된 값을 그대로 옮깁니다.

우선순위

이름 (문자열)	기호 값
`alert`	LOG_ALERT
`crit` 또는 `critical`	LOG_CRIT
`debug`	LOG_DEBUG
`emerg` 또는 `panic`	LOG_EMERG
`err` 또는 `error`	LOG_ERR
`info`	LOG_INFO
`notice`	LOG_NOTICE
`warn` 또는 `warning`	LOG_WARNING

시설

이름 (문자열)	기호 값
auth	LOG_AUTH
authpriv	LOG_AUTHPRIV
cron	LOG_CRON
daemon	LOG_DAEMON
ftp	LOG_FTP
kern	LOG_KERN
lpr	LOG_LPR
mail	LOG_MAIL
news	LOG_NEWS
syslog	LOG_SYSLOG
user	LOG_USER
uucp	LOG_UUCP
local0	LOG_LOCAL0
local1	LOG_LOCAL1
local2	LOG_LOCAL2
local3	LOG_LOCAL3
local4	LOG_LOCAL4
local5	LOG_LOCAL5
local6	LOG_LOCAL6
local7	LOG_LOCAL7

mapPriority(*levelname*)

로깅 수준 이름을 syslog 우선순위 이름으로 매핑합니다. 사용자 정의 수준을 사용하거나 기본 알고리즘이 여러분의 요구에 적합하지 않으면, 이 값을 재정의해야 할 수 있습니다. 기본 알고리즘은 `DEBUG`, `INFO`, `WARNING`, `ERROR` 및 `CRITICAL`을 동등한 syslog 이름으로 매핑하고, 다른 모든 수준 이름은 'warning' 으로 매핑합니다.

16.6.11 NTEventLogHandler

`logging.handlers` 모듈에 있는 *`NTEventLogHandler`* 클래스는 로깅 메시지를 로컬 윈도우 NT, 윈도우 2000 또는 윈도우 XP 이벤트 로그로 보내는 것을 지원합니다. 사용할 수 있으려면 먼저 Mark Hammond의 파이썬 용 Win32 확장이 설치되어 있어야 합니다.

class `logging.handlers`.**NTEventLogHandler**(*appname*, *dllname=None*, *logtype='Application'*)

`NTEventLogHandler` 클래스의 새 인스턴스를 반환합니다. *appname* 은 이벤트 로그에 나타나는 응용 프로그램 이름을 정의하는 데 사용됩니다. 이 이름을 사용하여 적절한 레지스트리 항목이 만들어집니다. *dllname* 은 로그에 보관할 메시지 정의를 포함하는 .dll 또는 .exe의 완전히 정규화된 경로명을 제공해야 합니다 (지정되지 않으면, `'win32service.pyd'`이 사용됩니다 - 이것은 Win32 확장과 함께 설치되며 몇 가지 기본 자리 표시자 메시지 정의를 포함합니다. 이 자리 표시자를 사용하면 전체 메시지 소스가 로그에 보관되므로 이벤트 로그가 커진다는 것에 유의하십시오. 간략한 로그를 원하면, 이벤트 로그에서 사용할 원하는 메시지 정의가 포함된 .dll 또는 .exe의 이름을 전달해야 합니다). *logtype* 은 `'Application'`, `'System'` 또는 `'Security'` 중 하나이며, 기본값은 `'Application'`입니다.

close()

이 시점에서, 이벤트 로그 항목의 소스로서의 응용 프로그램 이름을 레지스트리에서 제거할 수 있습니다. 그러나, 이렇게 하면, 이벤트 로그 뷰어에서 의도한 대로 이벤트를 볼 수 없게 됩니다 - 이벤트 로그 뷰어는 .dll 이름을 가져오기 위해 레지스트리에 액세스할 수 있어야 합니다. 현재 버전은 그렇게 하지 않습니다.

emit(*record*)

메시지 ID, 이벤트 범주 및 이벤트 유형을 결정한 다음, 메시지를 NT 이벤트 로그에 기록합니다.

getEventCategory(*record*)

레코드의 이벤트 범주를 반환합니다. 여러분 자신의 범주를 지정하려면, 이것을 재정의하십시오. 이 버전은 0을 반환합니다.

getEventType(*record*)

레코드의 이벤트 유형을 반환합니다. 여러분 자신의 유형을 지정하려면, 이것을 재정의하십시오. 이 버전은 처리기의 typemap 어트리뷰트를 사용하여 매핑하는데, `__init__()` 에서 `DEBUG`, `INFO`, `WARNING`, `ERROR` 및 `CRITICAL`에 대한 매핑이 포함된 딕셔너리로 설정됩니다. 여러분 자신의 수준을 사용한다면, 이 메서드를 재정의하거나 처리기의 *typemap* 어트리뷰트에 적절한 딕셔너리를 배치해야 합니다.

getMessageID(*record*)

레코드의 메시지 ID를 반환합니다. 여러분 자신의 메시지를 사용한다면, 로거에 전달된 *msg*를 포맷 문자열이 아닌 ID로 사용할 수 있습니다. 그런 다음 여기에서 딕셔너리 조회를 사용하여 메시지 ID를 가져올 수 있습니다. 이 버전은 `win32service.pyd`의 기본 메시지 ID인 1을 반환합니다.

16.6.12 SMTPHandler

`logging.handlers` 모듈에 있는 `SMTPHandler` 클래스는 SMTP를 통해 전자 메일 주소로 로깅 메시지를 보내는 것을 지원합니다.

class logging.handlers.**SMTPHandler**(*mailhost, fromaddr, toaddrs, subject, credentials=None, secure=None, timeout=1.0*)

`SMTPHandler` 클래스의 새 인스턴스를 반환합니다. 인스턴스는 전자 메일의 보내는 주소, 받는 주소와 제목 줄을 사용하여 초기화됩니다. *toaddrs* 는 문자열 리스트여야 합니다. 비표준 SMTP 포트를 지정하려면, *mailhost* 인자에 (host, port) 튜플 형식을 사용하십시오. 문자열을 사용하면 표준 SMTP 포트가 사용됩니다. SMTP 서버가 인증을 요구하면, *credentials* 인자에 (username, password) 튜플을 지정할 수 있습니다.

보안 프로토콜(TLS)의 사용을 지정하려면, *secure* 인자에 튜플을 전달하십시오. 이것은 인증 자격 증명(credentials)이 제공될 때만 사용됩니다. 튜플은 빈 튜플이거나, 키 파일 이름을 가진 단일 값 튜플이거나, 키 파일과 인증서 파일의 이름을 가진 2-튜플이어야 합니다. (이 튜플은 `smtplib.SMTP.starttls()` 메서드에 전달됩니다.)

timeout 인자를 사용하여 SMTP 서버와의 통신에 시간제한을 지정할 수 있습니다.

버전 3.3에서 변경: *timeout* 매개 변수가 추가되었습니다.

emit(*record*)

레코드를 포맷하고 지정된 주소로 보냅니다.

getSubject(*record*)

레코드에 종속적인 제목 줄을 지정하려면, 이 메서드를 재정의하십시오.

16.6.13 MemoryHandler

`logging.handlers` 모듈에 있는 `MemoryHandler` 클래스는 메모리에 로깅 레코드를 버퍼링하고, 주기적으로 대상 (*target*) 처리기로 플러시 하는 것을 지원합니다. 플러시는 버퍼가 꽉 찼거나 특정 심각도 이상의 이벤트가 발생할 때마다 발생합니다.

`MemoryHandler`는 추상 클래스이면서, 더 일반적인 `BufferingHandler`의 서브 클래스입니다. 이것은 레코드 로깅을 메모리에 버퍼링합니다. 각 레코드가 버퍼에 추가될 때마다, `shouldFlush()` 를 호출하여 버퍼를 플러시 할지 확인합니다. 필요하면, `flush()` 가 플러시를 수행할 것으로 기대합니다.

class logging.handlers.**BufferingHandler**(*capacity*)

지정된 용량(capacity)의 버퍼로 처리기를 초기화합니다. 여기서 *capacity*는 버퍼링 된 로깅 레코드 수를 의미합니다.

emit(*record*)

레코드를 버퍼에 추가합니다. `shouldFlush()`가 참을 반환하면 `flush()`를 호출하여 버퍼를 처리합니다.

flush()

For a `BufferingHandler` instance, flushing means that it sets the buffer to an empty list. This method can be overwritten to implement more useful flushing behavior.

shouldFlush(*record*)

버퍼의 용량이 찼으면 `True`를 반환합니다. 이 메서드는 사용자 정의 플러시 전략을 구현하기 위해 재정의될 수 있습니다.

class logging.handlers.**MemoryHandler**(*capacity, flushLevel=ERROR, target=None, flushOnClose=True*)

`MemoryHandler` 클래스의 새 인스턴스를 반환합니다. 인스턴스는 *capacity*(버퍼 된 레코드 수)의 버퍼 크기로 초기화됩니다. *flushLevel*을 지정하지 않으면, `ERROR`가 사용됩니다. *target* 이 지정되지 않으면, 이 처리기가 유용한 것을 하기 전에, `setTarget()`를 사용해 대상을 설정할 필요가 있습니다. *flushOnClose* 가 `False`로 지정되면, 처리기가 닫힐 때 버퍼가 플러시 되지 않습니다. 지정되지 않거나 `True`로 지정되면, 처리기가 닫힐 때 버퍼를 플러시 하는 이전 동작이 발생합니다.

버전 3.6에서 변경: *flushOnClose* 매개 변수가 추가되었습니다.

close()

`flush()`를 호출하고, 대상(target)을 `None`으로 설정하고, 버퍼를 비웁니다.

flush()

`MemoryHandler` 인스턴스의 경우, 플러시는 버퍼링 된 레코드가 있다면 대상으로 보내는 것을 뜻합니다. 버퍼링된 레코드가 대상에 전송될 때 버퍼도 지워집니다. 다른 행동을 원하면 재정의하십시오.

setTarget(*target*)

이 처리기의 대상 처리기를 설정합니다.

shouldFlush(*record*)

버퍼 가득 참이나 레코드가 *flushLevel* 이상을 만드는지 확인합니다.

16.6.14 HTTPHandler

`logging.handlers` 모듈에 있는 `HTTPHandler` 클래스는 `GET` 또는 `POST` 를 사용해서 로깅 메시지를 웹 서버로 보내는 것을 지원합니다.

class logging.handlers.**HTTPHandler**(*host, url, method='GET', secure=False, credentials=None, context=None*)

`HTTPHandler` 클래스의 새 인스턴스를 반환합니다. *host* 는 특정 포트 번호를 사용해야 하면 `host:port` 형식일 수 있습니다. *method* 를 지정하지 않으면 `GET`이 사용됩니다. *secure* 가 참이면, HTTPS 연결이 사용됩니다. *context* 매개 변수는 `ssl.SSLContext` 인스턴스로 설정되어, HTTPS 연결에 사용되는 SSL 설정을 구성할 수 있습니다. *credentials* 가 지정되면, 기본 인증을 사용하여 HTTP 'Authorization' 헤더에 배치되는 사용자 ID와 암호로 구성된 2-튜플이어야 합니다. credentials를 지정하면, 사용자 ID와 암호가 단순 텍스트로 전달되지 않도록 secure=True를 지정해야 합니다.

버전 3.5에서 변경: *context* 매개 변수가 추가되었습니다.

mapLogRecord(*record*)

URL 인코딩되어 웹 서버로 전송되는, `record`에 기반한 딕셔너리를 제공합니다. 기본 구현은 `record.__dict__`를 반환합니다. 이 메서드는 재정의할 수 있는데, 예를 들어 `LogRecord`의 일부만 웹 서버로 보내지거나, 서버로 보내는 내용에 대한 보다 구체적인 사용자 정의가 필요한 경우입니다.

emit(*record*)

URL 인코딩된 딕셔너리로 웹 서버에 레코드를 보냅니다. `mapLogRecord()` 메서드가 레코드를 전송할 딕셔너리로 변환하는 데 사용됩니다.

> **참고**
>
> 웹 서버로 보내기 위해 레코드를 준비하는 것은, 일반 포매팅 연산과 같지 않으므로, `setFormatter()`를 사용해서 `HTTPHandler`의 `Formatter`를 지정하는 것은 효과가 없습니다. `format()`을 호출하는 대신, 이 처리기는 `mapLogRecord()`를 호출한 다음, `urllib.parse.urlencode()`를 호출하여 웹 서버로 보내기에 적합한 형식으로 딕셔너리를 인코딩합니다.

16.6.15 QueueHandler

Added in version 3.2.

logging.handlers 모듈에 있는 *QueueHandler* 클래스는, *queue* 나 *multiprocessing* 모듈에 구현된 것과 같은 큐에 로깅 메시지를 보내는 것을 지원합니다.

QueueListener 클래스와 함께, *QueueHandler*를 사용하여 처리기가 로깅을 수행하는 스레드와 다른 스레드에서 작업을 수행하도록 할 수 있습니다. 이는 클라이언트를 처리하는 스레드가 가능한 한 신속하게 응답하고, 느린 작업(가령 *SMTPHandler*를 통해 전자 메일 보내기)은 별도의 스레드에서 수행되어야 하는 웹 응용 프로그램과 다른 서비스 응용 프로그램에서 중요합니다.

class logging.handlers.**QueueHandler**(*queue*)

QueueHandler 클래스의 새 인스턴스를 반환합니다. 인스턴스는 메시지를 보낼 큐로 초기화됩니다. *queue*는 임의의 큐류(queue-like) 객체일 수 있습니다; 메시지를 보내는 방법을 알아야 하는 *enqueue()* 메서드가 있는 그대로 사용합니다. 큐는 작업 추적 API를 갖도록 요구되지 않아서, *queue*에 *SimpleQueue* 인스턴스를 사용할 수 있습니다.

> **참고**
>
> If you are using *multiprocessing*, you should avoid using *SimpleQueue* and instead use *multiprocessing.Queue*.

> **경고**
>
> The *multiprocessing* module uses an internal logger created and accessed via *get_logger()*. *multiprocessing.Queue* will log DEBUG level messages upon items being queued. If those log messages are processed by a *QueueHandler* using the same *multiprocessing.Queue* instance, it will cause a deadlock or infinite recursion.

emit(*record*)

LogRecord를 준비한 결과를 큐에 넣습니다. 예외가 발생하면 (예를 들어, 유한(bounded) 큐가 다 차서), *handleError()* 메서드가 호출되어 에러를 처리합니다. 이로 인해 레코드가 조용히 버려지거나 (*logging.raiseExceptions*가 False 인 경우), 메시지가 sys.stderr에 인쇄됩니다 (*logging.raiseExceptions*가 True 인 경우).

prepare(*record*)

큐에 넣기 위해 레코드를 준비합니다. 이 메서드에 의해 반환된 객체는 큐에 들어갑니다.

The base implementation formats the record to merge the message, arguments, exception and stack information, if present. It also removes unpickleable items from the record in-place. Specifically, it overwrites the record' s msg and message attributes with the merged message (obtained by calling the handler' s *format()* method), and sets the args, exc_info and exc_text attributes to None.

레코드를 dict 나 JSON 문자열로 변환하거나, 원본을 그대로 두고 레코드의 수정 된 복사본을 보내길 원한다면 이 메서드를 재정의할 수 있습니다.

> **참고**
>
> The base implementation formats the message with arguments, sets the message and msg attributes to the formatted message and sets the args and exc_text attributes to None to allow pickling and to prevent further attempts at formatting. This means that a handler on the *QueueListener* side won' t have the information to do custom formatting, e.g. of exceptions. You may wish to subclass QueueHandler and override this method to e.g. avoid setting exc_text to None. Note that the message / msg / args changes are related to ensuring the record is pickleable, and you might or might not be able to avoid doing that depending on whether your args are pickleable. (Note that you may have to consider not only your own code but also code in any libraries that you use.)

enqueue(*record*)

put_nowait() 를 사용하여 큐에 레코드를 넣습니다; 블로킹 동작이나 시간제한이나, 사용자 정의 큐 구현을 사용하려면 이 메서드를 재정의할 수 있습니다.

listener

When created via configuration using *dictConfig()*, this attribute will contain a *QueueListener* instance for use with this handler. Otherwise, it will be None.

Added in version 3.12.

16.6.16 QueueListener

Added in version 3.2.

logging.handlers 모듈에 있는 *QueueListener* 클래스는 *queue* 나 *multiprocessing* 모듈에 구현된 것과 같은 큐에서 로깅 메시지를 수신하는 것을 지원합니다. 메시지는 내부 스레드의 큐에서 수신되고 처리를 위해 같은 스레드에서 하나 이상의 처리기로 전달됩니다. *QueueListener* 자체는 처리기가 아니지만, *QueueHandler* 와 함께 사용되기 때문에 여기에 설명되어 있습니다.

QueueHandler 클래스와 함께, *QueueListener*를 사용하여 처리기가 로깅을 수행하는 스레드와 다른 스레드에서 작업을 수행하도록 할 수 있습니다. 이는 클라이언트를 처리하는 스레드가 가능한 한 신속하게 응답하고, 느린 작업(가령 *SMTPHandler*를 통해 전자 메일 보내기)은 별도의 스레드에서 수행되어야 하는 웹 응용 프로그램과 다른 서비스 응용 프로그램에서 중요합니다.

class logging.handlers.**QueueListener**(*queue, *handlers, respect_handler_level=False*)

QueueListener 클래스의 새 인스턴스를 반환합니다. 인스턴스는 메시지를 보내는 큐와 큐에 있는 항목을 처리할 처리기의 리스트로 초기화됩니다. 큐는 임의의 큐류(queue-like) 객체일 수 있습니다; 메시지를 꺼내는 방법을 알아야 하는 *dequeue()* 메서드가 있는 그대로 사용합니다. 큐는 작업 추적 API를 갖도록 요구되지 않아서 (가능하면 사용됩니다), *queue*에 *SimpleQueue* 인스턴스를 사용할 수 있습니다.

> **참고**
>
> If you are using *multiprocessing*, you should avoid using *SimpleQueue* and instead use *multiprocessing.Queue*.

respect_handler_level 이 True 면, 처리기에 메시지를 전달할지를 결정할 때, 처리기의 수준이 존중됩니다 (메시지의 수준과 비교); 그렇지 않으면, 이전 파이썬 버전과 같게 동작합니다 - 항상 각 메시지를 모든 처리기에 전달합니다.

버전 3.5에서 변경: respect_handler_level 인자가 추가되었습니다.

버전 3.14에서 변경: *QueueListener* can now be used as a context manager via with. When entering the context, the listener is started. When exiting the context, the listener is stopped. *__enter__()* returns the *QueueListener* object.

dequeue(*block*)

레코드를 큐에서 꺼내 반환합니다. 선택적으로 블록 됩니다.

기본 구현은 get() 을 사용합니다. 시간제한을 사용하거나 사용자 정의 큐 구현을 사용하려면 이 메서드를 재정의할 수 있습니다.

prepare(*record*)

처리를 위해 레코드를 준비합니다.

이 구현은 단지 전달된 레코드를 반환합니다. 사용자 정의 직렬화를 수행하거나 처리기에 전달하기 전에 레코드를 조작해야 하면, 이 메서드를 재정의할 수 있습니다.

handle(*record*)

레코드를 처리합니다.

이것은 단지 모든 처리기로 레코드를 제공합니다. 처리기에 전달되는 실제 객체는 *prepare()* 에서 반환된 객체입니다.

start()

수신기를 시작합니다.

이것은 처리하기 위해 큐에서 LogRecord를 관찰하는 배경 스레드를 시작합니다.

버전 3.14에서 변경: Raises `RuntimeError` if called and the listener is already running.

stop()

수신기를 정지합니다.

스레드가 종료하도록 요청한 다음, 스레드가 종료할 때까지 대기합니다. 응용 프로그램이 종료되기 전에 이 함수를 호출하지 않으면, 레코드가 큐에 남아있을 수 있고, 이것들은 처리되지 않습니다.

enqueue_sentinel()

수신자에게 종료하도록 알리기 위해 큐에 종료 신호(sentinel)를 씁니다. 이 구현은 `put_nowait()`를 사용합니다. 시간제한을 사용하거나 사용자 정의 큐 구현을 사용하려면 이 메서드를 재정의할 수 있습니다.

Added in version 3.3.

더 보기

모듈 *logging*
logging 모듈에 관한 API 레퍼런스.

모듈 *logging.config*
logging 모듈용 구성 API.

16.7 platform — 하부 플랫폼의 식별 데이터에 대한 액세스

소스 코드: Lib/platform.py

참고

각 플랫폼은 알파벳순으로 나열되고, 리눅스는 유닉스 절에 포함됩니다.

16.7.1 Cross platform

platform.**architecture**(*executable=sys.executable, bits='', linkage=''*)

다양한 아키텍처 정보에 대해 주어진 실행 파일(기본값은 파이썬 인터프리터 바이너리)을 조회합니다.

실행 파일에 사용된 비트 아키텍처와 링크 형식에 대한 정보가 들어있는 튜플 `(bits, linkage)`를 반환합니다. 두 값은 모두 문자열로 반환됩니다.

결정할 수 없는 값은 매개 변수 사전 설정에 따라 반환됩니다. bits가 `''`로 주어지면, `sizeof(pointer)`(또는 파이썬 버전 < 1.5.2 에서는 `sizeof(long)`)가 지원되는 포인터 크기를 나타내는 데 사용됩니다.

함수는 시스템의 `file` 명령을 사용하여 실제 작업을 수행합니다. 이것은 대부분(전부가 아니라면)의 유닉스 플랫폼과 일부 유닉스가 아닌 플랫폼에서 가능하며 실행 파일이 파이썬 인터프리터를 가리키는 경우에만 가능합니다. 위의 요구가 충족되지 않으면 합리적인 기본값이 사용됩니다.

> **참고**
>
> 맥 OS(그리고 아마도 다른 플랫폼에서도)에서, 실행 파일은 다중 아키텍처를 포함하는 유니버설 파일일 수 있습니다.
>
> 현재 인터프리터가 "64-비트" 인지를 판단하려면, *sys.maxsize* 어트리뷰트를 조회하는 것이 더 신뢰성 있습니다.:
>
> `is_64bits = sys.maxsize > 2**32`

platform.**machine**()

기계 유형을 반환합니다, 예를 들어 `'AMD64'`. 값을 판별할 수 없으면 빈 문자열이 반환됩니다.

platform.**node**()

컴퓨터의 네트워크 이름을 반환합니다 (완전히 정규화되지 않았을 수 있습니다!). 값을 판별할 수 없으면 빈 문자열이 반환됩니다.

platform.**platform**(*aliased=False*, *terse=False*)

하부 플랫폼을 식별하는 가능한 한 많은 유용한 정보를 포함하는 단일 문자열을 반환합니다.

출력은 기계가 구문 분석하기보다는 사람이 읽을 수 있도록 합니다. 다른 플랫폼에서는 다르게 보일 수 있는데, 이는 의도 된 것입니다.

*aliased*가 참이면, 함수는 일반 이름과 다른 시스템 이름을 보고하는 다양한 플랫폼에 대해 별칭을 사용합니다, 예를 들어 SunOS는 Solaris로 보고됩니다. 이를 구현하는 데 *system_alias()* 함수가 사용됩니다.

*terse*를 참으로 설정하면 함수가 플랫폼을 식별하는 데 필요한 절대적으로 최소한의 정보만 반환합니다.

버전 3.8에서 변경: macOS에서, 이 함수는 이제 darwin 버전 대신 macOS 버전을 얻기 위해 *mac_ver()*를 사용합니다 (비어 있지 않은 릴리스 문자열을 반환한다면).

platform.**processor**()

(실제) 프로세서 이름을 반환합니다, 예를 들어 `'amdk6'`.

값을 판별할 수 없으면 빈 문자열이 반환됩니다. 많은 플랫폼이 이 정보를 제공하지 않거나 단순히 *machine()*과 같은 값을 반환함에 유의하십시오. NetBSD가 그렇습니다.

platform.**python_build**()

파이썬 빌드 번호와 날짜를 문자열로 나타내는 튜플 `(buildno, builddate)`를 반환합니다.

platform.**python_compiler**()

파이썬 컴파일에 사용된 컴파일러를 식별하는 문자열을 반환합니다.

platform.**python_branch**()

파이썬 구현 SCM 브랜치를 식별하는 문자열을 반환합니다.

platform.**python_implementation**()

파이썬 구현을 식별하는 문자열을 반환합니다. 가능한 반환 값은 이렇습니다: 'CPython', 'IronPython', 'Jython', 'PyPy'.

platform.**python_revision**()

파이썬 구현 SCM 리비전을 식별하는 문자열을 반환합니다.

platform.**python_version**()

파이썬 버전을 문자열 `'major.minor.patchlevel'`로 반환합니다.

파이썬 `sys.version`과 달리, 반환 값은 항상 patchlevel을 포함함에 유의하십시오 (기본값은 0입니다).

platform.**python_version_tuple**()

파이썬 버전을 문자열의 튜플 (major, minor, patchlevel)로 반환합니다.

파이썬 sys.version과 달리, 반환 값은 항상 patchlevel을 포함함에 유의하십시오 (기본값은 '0'입니다).

platform.**release**()

시스템의 릴리스를 반환합니다, 예를 들어 '2.2.0'이나 'NT'. 값을 판별할 수 없으면 빈 문자열이 반환됩니다.

platform.**system**()

시스템/OS 이름을 반환합니다, 가령 'Linux', 'Darwin', 'Java', 'Windows'. 값을 판별할 수 없으면 빈 문자열이 반환됩니다.

On iOS and Android, this returns the user-facing OS name (i.e, 'iOS, 'iPadOS' or 'Android'). To obtain the kernel name ('Darwin' or 'Linux'), use os.uname().

platform.**system_alias**(*system*, *release*, *version*)

일부 시스템에서 사용되는 상용 마케팅 이름으로 별칭 된 (system, release, version)을 반환합니다. 혼동을 일으킬 수 있는 일부 경우에 정보의 순서를 변경하기도 합니다.

platform.**version**()

시스템의 릴리스 버전을 반환합니다, 예를 들어 '#3 on degas'. 값을 판별할 수 없으면 빈 문자열이 반환됩니다.

On iOS and Android, this is the user-facing OS version. To obtain the Darwin or Linux kernel version, use os.uname().

platform.**uname**()

꽤 이식성 있는 uname 인터페이스. system, node, release, version, machine, processor의 6개의 어트리뷰트를 포함한 namedtuple()를 반환합니다.

processor는 요청 시 늦게 확인됩니다.

Note: the first two attribute names differ from the names presented by os.uname(), where they are named sysname and nodename.

결정할 수 없는 항목은 ''로 설정됩니다.

버전 3.3에서 변경: 결과가 튜플에서 namedtuple()로 변경되었습니다.

버전 3.9에서 변경: processor is resolved late instead of immediately.

platform.**invalidate_caches**()

Clear out the internal cache of information, such as the uname(). This is typically useful when the platform's node() is changed by an external process and one needs to retrieve the updated value.

Added in version 3.14.

16.7.2 Java platform

platform.**java_ver**(*release*='', *vendor*='', *vminfo*=('', '', ''), *osinfo*=('', '', ''))

Jython의 버전 인터페이스.

튜플 (release, vendor, vminfo, osinfo)를 반환하는데, *vminfo*는 튜플 (vm_name, vm_release, vm_vendor)이고, *osinfo*는 튜플 (os_name, os_version, os_arch)입니다. 결정할 수 없는 값은 매개 변수로 지정된 기본값으로 설정됩니다 (기본값은 모두 ''입니다).

Deprecated since version 3.13, will be removed in version 3.15: It was largely untested, had a confusing API, and was only useful for Jython support.

16.7.3 Windows platform

platform.**win32_ver**(*release=''*, *version=''*, *csd=''*, *ptype=''*)

윈도우 레지스트리에서 추가 버전 정보를 얻고 OS 릴리스, 버전 번호, CSD 수준 (서비스 팩) 및 OS 유형(다중/단일 프로세서)을 가리키는 튜플 `(release, version, csd, ptype)`를 반환합니다. 파악할 수 없는 값은 매개 변수로 주어지는 기본값(모두 빈 문자열이 기본값입니다)으로 설정됩니다.

힌트: *ptype*은 단일 프로세서 NT 기계에서는 `'Uniprocessor Free'`이고 다중 프로세서 기계에서는 `'Multiprocessor Free'`입니다. `'Free'`는 디버깅 코드가 없는 OS 버전을 나타냅니다. 또한 `'Checked'`를 언급할 수 있는데, OS 버전이 디버깅 코드, 즉 인자, 범위 등을 검사하는 코드를 사용한다는 것을 뜻합니다.

platform.**win32_edition**()

현재 윈도우 에디션을 나타내는 문자열을 반환합니다. 값을 파악할 수 없으면 `None`을 반환합니다. 가능한 값은 `'Enterprise'`, `'IoTUAP'`, `'ServerStandard'` 및 `'nanoserver'`를 포함하지만 이에 국한되지는 않습니다.

Added in version 3.8.

platform.**win32_is_iot**()

*win32_edition()*에 의해 반환된 윈도우 에디션이 IoT 에디션으로 인식되면 `True`를 반환합니다.

Added in version 3.8.

16.7.4 macOS platform

platform.**mac_ver**(*release=''*, *versioninfo=('', '', '')*, *machine=''*)

맥 OS 버전 정보를 얻고 튜플 `(release, versioninfo, machine)`으로 반환하는데, *versioninfo*는 튜플 `(version, dev_stage, non_release_version)`입니다.

결정할 수 없는 항목은 `''`로 설정됩니다. 모든 튜플 항목은 문자열입니다.

16.7.5 iOS platform

platform.**ios_ver**(*system=''*, *release=''*, *model=''*, *is_simulator=False*)

Get iOS version information and return it as a *namedtuple()* with the following attributes:

- `system` is the OS name; either `'iOS'` or `'iPadOS'`.
- `release` is the iOS version number as a string (e.g., `'17.2'`).
- `model` is the device model identifier; this will be a string like `'iPhone13,2'` for a physical device, or `'iPhone'` on a simulator.
- `is_simulator` is a boolean describing if the app is running on a simulator or a physical device.

판단할 수 없는 항목은 매개 변수로 주어지는 기본값으로 설정됩니다.

16.7.6 Unix platforms

platform.**libc_ver**(*executable=sys.executable*, *lib=''*, *version=''*, *chunksize=16384*)

파일 executable(기본값은 파이썬 인터프리터입니다)이 링크된 libc 버전을 확인하려고 시도합니다. 문자열의 튜플 `(lib, version)`을 반환하는데, 조회가 실패하면 지정된 매개 변수를 기본값으로 사용합니다.

다른 libc 버전이 실행 파일에 심볼을 추가하는 방법에 대해 이 함수가 가진 지식은 아마도 **gcc**로 컴파일된 실행 파일에서만 사용 가능하다는 것에 유의하십시오.

파일은 *chunksize* 바이트의 청크 단위로 읽고 스캔됩니다.

16.7.7 Linux platforms

platform.**freedesktop_os_release**()

Get operating system identification from os-release file and return it as a dict. The os-release file is a freedesktop.org standard and is available in most Linux distributions. A noticeable exception is Android and Android-based distributions.

Raises *OSError* or subclass when neither /etc/os-release nor /usr/lib/os-release can be read.

On success, the function returns a dictionary where keys and values are strings. Values have their special characters like " and $ unquoted. The fields NAME, ID, and PRETTY_NAME are always defined according to the standard. All other fields are optional. Vendors may include additional fields.

Note that fields like NAME, VERSION, and VARIANT are strings suitable for presentation to users. Programs should use fields like ID, ID_LIKE, VERSION_ID, or VARIANT_ID to identify Linux distributions.

예제:

```
def get_like_distro():
    info = platform.freedesktop_os_release()
    ids = [info["ID"]]
    if "ID_LIKE" in info:
        # id 들은 공백으로 구분되고 우선 순위에 따라 정렬됩니다
        ids.extend(info["ID_LIKE"].split())
    return ids
```

Added in version 3.10.

16.7.8 Android platform

platform.**android_ver**(*release=''*, *api_level=0*, *manufacturer=''*, *model=''*, *device=''*, *is_emulator=False*)

Get Android device information. Returns a *namedtuple()* with the following attributes. Values which cannot be determined are set to the defaults given as parameters.

- release - Android version, as a string (e.g. "14").
- api_level - API level of the running device, as an integer (e.g. 34 for Android 14). To get the API level which Python was built against, see *sys.getandroidapilevel()*.
- manufacturer - Manufacturer name.
- model - Model name – typically the marketing name or model number.
- device - Device name – typically the model number or a codename.
- is_emulator - True if the device is an emulator; False if it' s a physical device.

Google maintains a list of known model and device names.

Added in version 3.13.

16.7.9 Command-line usage

platform can also be invoked directly using the -m switch of the interpreter:

```
python -m platform [--terse] [--nonaliased] [{nonaliased,terse} ...]
```

The following options are accepted:

--terse

Print terse information about the platform. This is equivalent to calling *platform.platform()* with the *terse* argument set to True.

--nonaliased

Print platform information without system/OS name aliasing. This is equivalent to calling `platform.platform()` with the *aliased* argument set to `True`.

You can also pass one or more positional arguments (`terse`, `nonaliased`) to explicitly control the output format. These behave similarly to their corresponding options.

16.8 errno — 표준 errno 시스템 기호

이 모듈은 표준 `errno` 시스템 기호를 제공합니다. 각 기호의 값은 해당 정숫값입니다. 이름과 설명은 `linux/include/errno.h`에서 빌려 왔는데, 포괄적이어야 합니다.

`errno.`**errorcode**

errno 값에서 하부 시스템의 문자열 이름으로의 매핑을 제공하는 딕셔너리입니다. 예를 들어, `errno.errorcode[errno.EPERM]`는 `'EPERM'`로 매핑됩니다.

숫자 에러 코드를 에러 메시지로 변환하려면, `os.strerror()`를 사용하십시오.

다음 목록에서, 현재 플랫폼에서 사용되지 않는 기호는 모듈에서 정의하지 않습니다. 정의된 기호의 구체적인 목록은 `errno.errorcode.keys()`로 사용 가능합니다. 사용할 수 있는 기호는 다음과 같습니다:

`errno.`**EPERM**

Operation not permitted. 이 에러는 예외 `PermissionError`로 매핑됩니다.

`errno.`**ENOENT**

No such file or directory – 그런 파일이나 디렉터리가 없습니다. 이 에러는 예외 `FileNotFoundError`로 매핑됩니다.

`errno.`**ESRCH**

No such process – 그런 프로세스가 없습니다. 이 에러는 예외 `ProcessLookupError`로 매핑됩니다.

`errno.`**EINTR**

Interrupted system call – 중단된 시스템 호출. 이 에러는 예외 `InterruptedError`로 매핑됩니다.

`errno.`**EIO**

I/O error – I/O 에러

`errno.`**ENXIO**

No such device or address – 그런 장치나 주소가 없습니다.

`errno.`**E2BIG**

Arg list too long – 인자 목록이 너무 깁니다.

`errno.`**ENOEXEC**

Exec format error – Exec 포맷 에러

`errno.`**EBADF**

Bad file number – 잘못된 파일 번호

`errno.`**ECHILD**

No child processes – 그런 자식 프로세스가 없습니다. 이 에러는 예외 `ChildProcessError`로 매핑됩니다.

`errno.`**EAGAIN**

Try again – 다시 시도하십시오. 이 에러는 예외 `BlockingIOError`로 매핑됩니다.

`errno.`**ENOMEM**

Out of memory – 메모리 부족

errno.**EACCES**

Permission denied – 사용 권한이 거부되었습니다. 이 에러는 예외 *PermissionError*로 매핑됩니다.

errno.**EFAULT**

Bad address – 잘못된 주소

errno.**ENOTBLK**

Block device required – 블록 장치가 필요합니다

errno.**EBUSY**

Device or resource busy – 장치나 자원이 사용 중입니다

errno.**EEXIST**

File exists – 파일이 존재합니다. 이 에러는 예외 *FileExistsError*로 매핑됩니다.

errno.**EXDEV**

Cross-device link – 장치 간 링크

errno.**ENODEV**

No such device – 그런 장치가 없습니다

errno.**ENOTDIR**

Not a directory – 디렉터리가 아닙니다. 이 에러는 예외 *NotADirectoryError*로 매핑됩니다.

errno.**EISDIR**

Is a directory – 디렉터리입니다. 이 에러는 예외 *IsADirectoryError*로 매핑됩니다.

errno.**EINVAL**

Invalid argument – 잘못된 인자

errno.**ENFILE**

File table overflow – 파일 테이블 오버플로

errno.**EMFILE**

Too many open files – 열려있는 파일이 너무 많습니다

errno.**ENOTTY**

Not a typewriter – 타자기가 아닙니다

errno.**ETXTBSY**

Text file busy – 텍스트 파일이 사용 중입니다

errno.**EFBIG**

File too large – 파일이 너무 큽니다

errno.**ENOSPC**

No space left on device – 장치에 남은 공간이 없습니다.

errno.**ESPIPE**

Illegal seek – 잘못된 탐색

errno.**EROFS**

Read-only file system – 읽기 전용 파일 시스템

errno.**EMLINK**

Too many links – 링크가 너무 많습니다

errno.**EPIPE**

Broken pipe – 깨진 파이프. 이 에러는 예외 *BrokenPipeError*로 매핑됩니다.

errno.**EDOM**

Math argument out of domain of func – 함수의 범위를 벗어난 수학 인자

errno.**ERANGE**
Math result not representable – 수학 결과를 표현할 수 없습니다

errno.**EDEADLK**
Resource deadlock would occur – 자원 교착 상태가 발생합니다

errno.**ENAMETOOLONG**
File name too long – 파일 이름이 너무 깁니다

errno.**ENOLCK**
No record locks available – 사용 가능한 레코드 록이 없습니다

errno.**ENOSYS**
Function not implemented – 기능이 구현되지 않았습니다

errno.**ENOTEMPTY**
Directory not empty – 디렉터리가 비어 있지 않습니다

errno.**ELOOP**
Too many symbolic links encountered – 마주친 심볼릭 링크가 너무 많습니다

errno.**EWOULDBLOCK**
Operation would block – 연산이 블록 됩니다. 이 에러는 예외 `BlockingIOError`로 매핑됩니다.

errno.**ENOMSG**
No message of desired type – 원하는 유형의 메시지가 없습니다

errno.**EIDRM**
Identifier removed – 식별자가 삭제되었습니다

errno.**ECHRNG**
Channel number out of range – 채널 번호가 범위를 벗어났습니다

errno.**EL2NSYNC**
Level 2 not synchronized – 수준 2가 동기화되지 않았습니다

errno.**EL3HLT**
Level 3 halted – 수준 3이 정지되었습니다

errno.**EL3RST**
Level 3 reset – 수준 3이 재설정되었습니다

errno.**ELNRNG**
Link number out of range – 링크 번호가 범위를 벗어났습니다

errno.**EUNATCH**
Protocol driver not attached – 프로토콜 드라이버가 연결되지 않았습니다

errno.**ENOCSI**
No CSI structure available – 사용 가능한 CSI 구조가 없습니다

errno.**EL2HLT**
Level 2 halted – 수준 2가 중지되었습니다

errno.**EBADE**
Invalid exchange – 잘못된 교환

errno.**EBADR**
Invalid request descriptor – 잘못된 요청 기술자

errno.**EXFULL**
Exchange full – 교환 포화

errno.**ENOANO**
No anode – anode가 없습니다

errno.**EBADRQC**
Invalid request code – 유효하지 않은 요청 코드

errno.**EBADSLT**
Invalid slot – 유효하지 않은 슬롯

errno.**EDEADLOCK**
File locking deadlock error – 파일 잠금 교착 상태 에러

errno.**EBFONT**
Bad font file format – 잘못된 글꼴 파일 형식

errno.**ENOSTR**
Device not a stream – 장치가 스트림이 아닙니다

errno.**ENODATA**
No data available – 데이터가 없습니다

errno.**ETIME**
Timer expired – 타이머가 만료되었습니다

errno.**ENOSR**
Out of streams resources – 스트림 자원 부족

errno.**ENONET**
Machine is not on the network – 기계가 네트워크에 없습니다.

errno.**ENOPKG**
Package not installed – 패키지가 설치되지 않았습니다

errno.**EREMOTE**
Object is remote – 객체가 원격입니다

errno.**ENOLINK**
Link has been severed – 링크가 절단되었습니다

errno.**EADV**
Advertise error – 광고 에러

errno.**ESRMNT**
Srmount error – srmount 에러

errno.**ECOMM**
Communication error on send – 전송 시 통신 에러

errno.**EPROTO**
Protocol error – 프로토콜 에러

errno.**EMULTIHOP**
Multihop attempted – 다중 홉을 시도했습니다

errno.**EDOTDOT**
RFS specific error – RFS 특정 에러

errno.**EBADMSG**
Not a data message – 데이터 메시지가 아닙니다

errno.**EOVERFLOW**
Value too large for defined data type – 정의된 데이터형에 비해 값이 너무 큽니다

errno.**ENOTUNIQ**
Name not unique on network – 이름이 네트워크에서 고유하지 않습니다

errno.**EBADFD**
File descriptor in bad state – 잘못된 상태의 파일 기술자

errno.**EREMCHG**
Remote address changed – 원격 주소가 변경되었습니다

errno.**ELIBACC**
Can not access a needed shared library – 필요한 공유 라이브러리에 액세스할 수 없습니다.

errno.**ELIBBAD**
Accessing a corrupted shared library – 손상된 공유 라이브러리 액세스

errno.**ELIBSCN**
.lib section in a.out corrupted – 손상된 a.out의 .lib 섹션

errno.**ELIBMAX**
Attempting to link in too many shared libraries – 너무 많은 공유 라이브러리 연결 시도

errno.**ELIBEXEC**
Cannot exec a shared library directly – 공유 라이브러리를 직접 실행할 수 없습니다

errno.**EILSEQ**
Illegal byte sequence – 잘못된 바이트 시퀀스

errno.**ERESTART**
Interrupted system call should be restarted – 중단된 시스템 호출을 다시 시작해야 합니다

errno.**ESTRPIPE**
Streams pipe error – 스트림 파이프 에러

errno.**EUSERS**
Too many users – 사용자가 너무 많습니다

errno.**ENOTSOCK**
Socket operation on non-socket – 비 소켓에 대한 소켓 연산

errno.**EDESTADDRREQ**
Destination address required – 목적지 주소가 필요합니다

errno.**EMSGSIZE**
Message too long – 메시지가 너무 깁니다

errno.**EPROTOTYPE**
Protocol wrong type for socket – 소켓에 대한 프로토콜 유형이 잘못되었습니다

errno.**ENOPROTOOPT**
Protocol not available – 프로토콜을 사용할 수 없습니다

errno.**EPROTONOSUPPORT**
Protocol not supported – 지원되지 않는 프로토콜

errno.**ESOCKTNOSUPPORT**
Socket type not supported – 지원되지 않는 소켓 유형

errno.**EOPNOTSUPP**
Operation not supported on transport endpoint – 트랜스포트 끝점에서 지원되지 않는 연산

errno.**ENOTSUP**
Operation not supported – 지원되지 않는 연산

Added in version 3.2.

errno.**EPFNOSUPPORT**

Protocol family not supported – 지원되지 않는 프로토콜 패밀리

errno.**EAFNOSUPPORT**

Address family not supported by protocol – 프로토콜이 지원하지 않는 주소 패밀리

errno.**EADDRINUSE**

Address already in use – 이미 사용 중인 주소

errno.**EADDRNOTAVAIL**

Cannot assign requested address – 요청된 주소를 할당할 수 없습니다.

errno.**ENETDOWN**

Network is down – 네트워크가 다운되었습니다

errno.**ENETUNREACH**

Network is unreachable – 네트워크에 도달할 수 없습니다

errno.**ENETRESET**

Network dropped connection because of reset – 재설정으로 인해 네트워크 연결이 끊겼습니다

errno.**ECONNABORTED**

Software caused connection abort – 소프트웨어로 인해 연결이 중단되었습니다. 이 에러는 예외 `ConnectionAbortedError`로 매핑됩니다.

errno.**ECONNRESET**

Connection reset by peer – 피어에 의한 연결 재설정. 이 에러는 예외 `ConnectionResetError`로 매핑됩니다.

errno.**ENOBUFS**

No buffer space available – 사용 가능한 버퍼 공간이 없습니다.

errno.**EISCONN**

Transport endpoint is already connected – 트랜스포트 끝점이 이미 연결되어 있습니다.

errno.**ENOTCONN**

Transport endpoint is not connected – 트랜스포트 끝점이 연결되어 있지 않습니다.

errno.**ESHUTDOWN**

Cannot send after transport endpoint shutdown – 트랜스포트 끝점 종료 후에 보낼 수 없습니다. 이 에러는 예외 `BrokenPipeError`로 매핑됩니다.

errno.**ETOOMANYREFS**

Too many references: cannot splice – 참조가 너무 많습니다: 연결할 수 없습니다

errno.**ETIMEDOUT**

Connection timed out – 연결 시간이 초과하였습니다. 이 에러는 예외 `TimeoutError`로 매핑됩니다.

errno.**ECONNREFUSED**

Connection refused – 연결이 거부되었습니다. 이 에러는 예외 `ConnectionRefusedError`로 매핑됩니다.

errno.**EHOSTDOWN**

Host is down – 호스트가 다운되었습니다

errno.**EHOSTUNREACH**

No route to host – 호스트로 가는 길이 없습니다

errno.**EHWPOISON**

Memory page has hardware error.

Added in version 3.14.

errno.**EALREADY**

Operation already in progress – 이미 진행 중인 연산. 이 에러는 예외 *BlockingIOError*로 매핑됩니다.

errno.**EINPROGRESS**

Operation now in progress – 이제 연산이 진행 중입니다. 이 에러는 예외 *BlockingIOError*로 매핑됩니다.

errno.**ESTALE**

Stale NFS file handle – 오래된 NFS 파일 핸들

errno.**EUCLEAN**

Structure needs cleaning – 구조 청소 필요

errno.**ENOTNAM**

Not a XENIX named type file – XENIX 이름 붙은 형식 파일이 아닙니다

errno.**ENAVAIL**

No XENIX semaphores available – 사용할 수 있는 XENIX 세마포어가 없습니다

errno.**EISNAM**

Is a named type file – 이름 붙은 형식 파일입니다

errno.**EREMOTEIO**

Remote I/O error – 원격 I/O 에러

errno.**EDQUOT**

Quota exceeded – 할당량 초과

errno.**EQFULL**

Interface output queue is full

Added in version 3.11.

errno.**ENOMEDIUM**

No medium found

errno.**EMEDIUMTYPE**

Wrong medium type

errno.**ENOKEY**

Required key not available

errno.**EKEYEXPIRED**

Key has expired

errno.**EKEYREVOKED**

Key has been revoked

errno.**EKEYREJECTED**

Key was rejected by service

errno.**ERFKILL**

Operation not possible due to RF-kill

errno.**ELOCKUNMAPPED**

Locked lock was unmapped

errno.**ENOTACTIVE**

Facility is not active

errno.**EAUTH**

Authentication error

Added in version 3.2.

errno.**EBADARCH**

Bad CPU type in executable

Added in version 3.2.

errno.**EBADEXEC**

Bad executable (or shared library)

Added in version 3.2.

errno.**EBADMACHO**

Malformed Mach-o file

Added in version 3.2.

errno.**EDEVERR**

Device error

Added in version 3.2.

errno.**EFTYPE**

Inappropriate file type or format

Added in version 3.2.

errno.**ENEEDAUTH**

Need authenticator

Added in version 3.2.

errno.**ENOATTR**

Attribute not found

Added in version 3.2.

errno.**ENOPOLICY**

Policy not found

Added in version 3.2.

errno.**EPROCLIM**

Too many processes

Added in version 3.2.

errno.**EPROCUNAVAIL**

Bad procedure for program

Added in version 3.2.

errno.**EPROGMISMATCH**

Program version wrong

Added in version 3.2.

errno.**EPROGUNAVAIL**

RPC prog. not avail

Added in version 3.2.

errno.**EPWROFF**

Device power is off

Added in version 3.2.

errno.**EBADRPC**

RPC struct is bad

Added in version 3.2.

errno.**ERPCMISMATCH**

RPC version wrong

Added in version 3.2.

errno.**ESHLIBVERS**

Shared library version mismatch

Added in version 3.2.

errno.**ENOTCAPABLE**

Capabilities insufficient. This error is mapped to the exception *PermissionError*.

가용성: WASI, FreeBSD

Added in version 3.11.1.

errno.**ECANCELED**

Operation canceled

Added in version 3.2.

errno.**EOWNERDEAD**

Owner died

Added in version 3.2.

errno.**ENOTRECOVERABLE**

State not recoverable

Added in version 3.2.

16.9 ctypes — A foreign function library for Python

Source code: Lib/ctypes

---

*ctypes*는 파이썬용 외부 함수(foreign function) 라이브러리입니다. C 호환 데이터형을 제공하며, DLL 또는 공유 라이브러리에 있는 함수를 호출할 수 있습니다. 이 라이브러리들을 순수 파이썬으로 감싸는 데 사용할 수 있습니다.

16.9.1 ctypes 자습서

Note: The code samples in this tutorial use *doctest* to make sure that they actually work. Since some code samples behave differently under Linux, Windows, or macOS, they contain doctest directives in comments.

참고: 일부 코드 예제는 ctypes *c_int* 형을 참조합니다. `sizeof(long) == sizeof(int)` 인 플랫폼에서, 이는 *c_long*의 별칭입니다. 따라서 *c_int*를 기대할 때 *c_long*가 인쇄되더라도 혼란스러워하지 않아도 됩니다 — 이것들은 실제로 같은 형입니다.

동적 링크 라이브러리 로드하기

*ctypes*는 동적 링크 라이브러리 로드를 위해 *cdll*을, 그리고 윈도우에서는 *windll* 및 *oledll* 객체를, 노출합니다.

You load libraries by accessing them as attributes of these objects. *cdll* loads libraries which export functions using the standard `cdecl` calling convention, while *windll* libraries call functions using the `stdcall` calling convention. *oledll* also uses the `stdcall` calling convention, and assumes the functions return a Windows `HRESULT` error code. The error code is used to automatically raise an *OSError* exception when the function call fails.

버전 3.3에서 변경: 윈도우 에러는 *WindowsError*를 일으켜왔습니다. 이제는 *OSError*의 별칭입니다.

Here are some examples for Windows. Note that `msvcrt` is the MS standard C library containing most standard C functions, and uses the `cdecl` calling convention:

```
>>> from ctypes import *
>>> print(windll.kernel32)
<WinDLL 'kernel32', handle ... at ...>
>>> print(cdll.msvcrt)
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

윈도우는 일반적인 `.dll` 파일 접미사를 자동으로 추가합니다.

> **참고**
>
> `cdll.msvcrt`를 통해 표준 C 라이브러리에 액세스하면 파이썬에서 사용되는 라이브러리와 호환되지 않는 오래된 라이브러리 버전이 사용됩니다. 가능하면 파이썬 자체의 기능을 사용하거나, `msvcrt` 모듈을 임포트 해서 사용하십시오.

On Linux, it is required to specify the filename *including* the extension to load a library, so attribute access can not be used to load libraries. Either the *LoadLibrary()* method of the dll loaders should be used, or you should load the library by creating an instance of CDLL by calling the constructor:

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

로드된 dll에서 함수에 액세스하기

함수는 dll 객체의 어트리뷰트로 액세스 됩니다:

```
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.GetModuleHandleA)
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.MyOwnFunction)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

Note that win32 system dlls like `kernel32` and `user32` often export ANSI as well as UNICODE versions of a function. The UNICODE version is exported with a `W` appended to the name, while the ANSI version is exported with an `A` appended to the name. The win32 `GetModuleHandle` function, which returns a *module handle* for a given module name, has the following C prototype, and a macro is used to expose one of them as `GetModuleHandle` depending on whether UNICODE is defined or not:

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

*windll*는 마술적으로 이 중 하나를 선택하려고 하지 않으므로, `GetModuleHandleA` 나 `GetModuleHandleW` 를 명시적으로 지정하여 필요한 버전에 액세스해야 하고, 그런 다음 각각 바이트열이나 문자열 객체로 호출해야 합니다.

때때로, dll은 `"??2@YAPAXI@Z"`와 같은 유효한 파이썬 식별자가 아닌 이름으로 함수를 내보냅니다. 이때는 *getattr()*를 사용하여 함수를 조회해야 합니다:

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

윈도우에서, 일부 dll은 이름이 아니라 서수(ordinal)로 함수를 내보냅니다. 이 함수는 서수로 dll 객체를 인덱싱하여 액세스할 수 있습니다:

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

함수 호출하기

You can call these functions like any other Python callable. This example uses the `rand()` function, which takes no arguments and returns a pseudo-random integer:

```
>>> print(libc.rand())
1804289383
```

On Windows, you can call the `GetModuleHandleA()` function, which returns a win32 module handle (passing `None` as single argument to call it with a `NULL` pointer):

```
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
>>>
```

`cdecl` 호출 규칙을 사용하여 `stdcall` 함수를 호출하면 *ValueError*가 발생하고, 그 반대도 마찬가지입니다:

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf(b"spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

올바른 호출 규칙을 찾으려면 C 헤더 파일이나 호출할 함수에 대한 설명서를 살펴봐야 합니다.

윈도우에서, *ctypes*는 함수가 유효하지 않은 인자 값을 사용하여 호출될 때, 일반적인 보호 오류로 인한 충돌을 방지하기 위해 win32 구조적 예외 처리를 사용합니다:

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation reading 0x00000020
>>>
```

그러나, *ctypes*로 파이썬을 충돌시킬 방법이 많으므로, 어쨌든 주의해야 합니다. *faulthandler* 모듈은 충돌을 디버깅하는 데 도움이 될 수 있습니다 (예를 들어, 오류가 있는 C 라이브러리 호출로 인한 세그먼트 오류).

`None`, integers, bytes objects and (unicode) strings are the only native Python objects that can directly be used as parameters in these function calls. `None` is passed as a C `NULL` pointer, bytes objects and strings are passed as pointer to the memory block that contains their data (`char*` or `wchar_t*`). Python integers are passed as the platform' s default C `int` type, their value is masked to fit into the C type.

다른 매개 변수 형으로 함수를 호출하기 전에, *ctypes* 데이터형에 대해 더 알아야 합니다.

기본 데이터형

*ctypes*는 많은 기본적인 C 호환 데이터형을 정의합니다.:

ctypes 형	C 형	파이썬 형
c_bool	`_Bool`	bool (1)
c_char	`char`	1-문자 바이트열 객체
c_wchar	`wchar_t`	1-문자 문자열
c_byte	`char`	int
c_ubyte	`unsigned char`	int
c_short	`short`	int
c_ushort	`unsigned short`	int
c_int	`int`	int
c_int8	`int8_t`	int
c_int16	`int16_t`	int
c_int32	`int32_t`	int
c_int64	`int64_t`	int
c_uint	`unsigned int`	int
c_uint8	`uint8_t`	int
c_uint16	`uint16_t`	int
c_uint32	`uint32_t`	int
c_uint64	`uint64_t`	int
c_long	`long`	int
c_ulong	`unsigned long`	int
c_longlong	`__int64` or `long long`	int
c_ulonglong	`unsigned __int64` or `unsigned long long`	int
c_size_t	`size_t`	int
c_ssize_t	`ssize_t` or `Py_ssize_t`	int
c_time_t	`time_t`	int
c_float	`float`	float
c_double	`double`	float
c_longdouble	`long double`	float
c_char_p	`char*` (NUL terminated)	바이트열 객체나 `None`
c_wchar_p	`wchar_t*` (NUL terminated)	문자열이나 `None`
c_void_p	`void*`	int 나 `None`

(1) 생성자는 논릿값을 가진 모든 객체를 받아들입니다.

Additionally, if IEC 60559 compatible complex arithmetic (Annex G) is supported in both C and `libffi`, the following complex types are available:

ctypes 형	C 형	파이썬 형
c_float_complex	float complex	complex
c_double_complex	double complex	complex
c_longdouble_complex	long double complex	complex

이 모든 형은 올바른 형과 값의 선택적 초기화자로 호출해서 만들어질 수 있습니다:

```
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p(140018365411392)
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

이러한 형은 가변이므로, 값을 나중에 변경할 수도 있습니다:

```
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
42
>>> i.value = -99
>>> print(i.value)
-99
>>>
```

Assigning a new value to instances of the pointer types *c_char_p*, *c_wchar_p*, and *c_void_p* changes the *memory location* they point to, *not the contents* of the memory block (of course not, because Python string objects are immutable):

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p(139966785747344)
>>> print(c_s.value)
Hello World
>>> c_s.value = "Hi, there"
>>> print(c_s)              # the memory location has changed
c_wchar_p(139966783348904)
>>> print(c_s.value)
Hi, there
>>> print(s)                # first object is unchanged
Hello, World
>>>
```

그러나, 이것들을 가변 메모리에 대한 포인터를 예상하는 함수에 전달하지 않도록 주의해야 합니다. 가변 메모리 블록이 필요하다면, ctypes에는 다양한 방법으로 이를 만드는 *create_string_buffer()* 함수가 있습니다. 현재 메모리 블록 내용은 raw 프로퍼티를 사용하여 액세스(또는 변경)할 수 있습니다; NUL 종료 문자열로 액세스하려면 value 프로퍼티를 사용하십시오:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)            # create a 3 byte buffer, initialized␣
↪to NUL bytes
>>> print(sizeof(p), repr(p.raw))
3 b'\x00\x00\x00'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> p = create_string_buffer(b"Hello")      # create a buffer containing a NUL␣
↪terminated string
>>> print(sizeof(p), repr(p.raw))
6 b'Hello\x00'
>>> print(repr(p.value))
b'Hello'
>>> p = create_string_buffer(b"Hello", 10) # create a 10 byte buffer
>>> print(sizeof(p), repr(p.raw))
10 b'Hello\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw))
10 b'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

The *create_string_buffer()* function replaces the old `c_buffer()` function (which is still available as an alias). To create a mutable memory block containing unicode characters of the C type `wchar_t`, use the *create_unicode_buffer()* function.

함수 호출하기, 계속

printf 는 *sys.stdout* 이 아니라 실제 표준 출력으로 인쇄하므로, 이 예제는 콘솔 프롬프트에서만 작동하고 *IDLE* 이나 *PythonWin* 에서는 작동하지 않음에 유의하십시오:

```
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf(b"%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: TypeError: Don't know how to convert parameter 2
>>>
```

이전에 언급했듯이, 정수, 문자열 및 바이트열 객체를 제외한 모든 파이썬 형은 필요한 C 데이터형으로 변환될 수 있도록 해당하는 *ctypes* 형으로 래핑 되어야 합니다:

```
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

Calling variadic functions

On a lot of platforms calling variadic functions through ctypes is exactly the same as calling functions with a fixed number of parameters. On some platforms, and in particular ARM64 for Apple Platforms, the calling convention for variadic functions is different than that for regular functions.

On those platforms it is required to specify the *argtypes* attribute for the regular, non-variadic, function arguments:

```
libc.printf.argtypes = [ctypes.c_char_p]
```

Because specifying the attribute does not inhibit portability it is advised to always specify *argtypes* for all variadic functions.

사용자 정의 데이터형을 사용하여 함수 호출하기

You can also customize *ctypes* argument conversion to allow instances of your own classes be used as function arguments. *ctypes* looks for an _as_parameter_ attribute and uses this as the function argument. The attribute must be an integer, string, bytes, a *ctypes* instance, or an object with an _as_parameter_ attribute:

```
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

If you don' t want to store the instance' s data in the _as_parameter_ instance variable, you could define a *property* which makes the attribute available on request.

필수 인자 형 (함수 프로토타입) 지정하기

It is possible to specify the required argument types of functions exported from DLLs by setting the *argtypes* attribute.

argtypes must be a sequence of C data types (the printf() function is probably not a good example here, because it takes a variable number and different types of parameters depending on the format string, on the other hand this is quite handy to experiment with this feature):

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

포맷을 지정하면 호환되지 않는 인자 형으로부터 보호하고(C 함수의 프로토타입처럼), 유효한 형으로 인자를 변환하려고 시도합니다:

```
>>> printf(b"%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: TypeError: 'int' object cannot be interpreted as␣
↪ctypes.c_char_p
>>> printf(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
13
>>>
```

If you have defined your own classes which you pass to function calls, you have to implement a *from_param()* class method for them to be able to use them in the *argtypes* sequence. The *from_param()* class method receives the Python object passed to the function call, it should do a typecheck or whatever is needed to make sure this object is acceptable, and then return the object itself, its _as_parameter_ attribute, or whatever you want to pass as the C function argument in this case. Again, the result should be an integer, string, bytes, a *ctypes* instance, or an object with an _as_parameter_ attribute.

반환형

By default functions are assumed to return the C `int` type. Other return types can be specified by setting the *restype* attribute of the function object.

The C prototype of `time()` is `time_t time(time_t *)`. Because `time_t` might be of a different type than the default return type `int`, you should specify the `restype` attribute:

```
>>> libc.time.restype = c_time_t
```

The argument types can be specified using *argtypes*:

```
>>> libc.time.argtypes = (POINTER(c_time_t),)
```

To call the function with a `NULL` pointer as first argument, use `None`:

```
>>> print(libc.time(None))
1150640792
```

Here is a more advanced example, it uses the `strchr()` function, which expects a string pointer and a char, and returns a pointer to a string:

```
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p    # c_char_p is a pointer to a string
>>> strchr(b"abcdef", ord("d"))
b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
>>>
```

If you want to avoid the *ord("x")* calls above, you can set the *argtypes* attribute, and the second argument will be converted from a single character Python bytes object into a C char:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
b'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
ctypes.ArgumentError: argument 2: TypeError: one character bytes, bytearray or␣
↪integer expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
b'def'
>>>
```

You can also use a callable Python object (a function or a class for example) as the *restype* attribute, if the foreign function returns an integer. The callable will be called with the *integer* the C function returns, and the result of this call will be used as the result of your function call. This is useful to check for error return values and automatically raise an exception:

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
OSError: [Errno 126] The specified module could not be found.
>>>
```

`WinError`는 윈도우 `FormatMessage()` api를 호출하여 에러 코드의 문자열 표현을 가져오고, 예외를 반환하는 함수입니다. `WinError`는 선택적 에러 코드 매개 변수를 취합니다, 제공하지 않으면 *GetLastError()*를 호출하여 에러 코드를 가져옵니다.

Please note that a much more powerful error checking mechanism is available through the *errcheck* attribute; see the reference manual for details.

포인터 전달하기 (또는: 참조로 매개 변수 전달하기)

때때로 C api 함수는 매개 변수로 데이터형을 가리키는 포인터를 기대합니다, 아마도 해당 위치에 쓰기 위해서, 또는 데이터가 너무 커서 값으로 전달할 수 없어서. 이것은 참조로 매개 변수 전달하기로 알려져 있기도 합니다.

*ctypes*는 매개 변수를 참조로 전달하는 데 사용되는 *byref()* 함수를 내보냅니다. 같은 효과를 *pointer()* 함수로도 얻을 수 있습니다. 하지만 *pointer()*는 실제 포인터 객체를 생성하기 때문에 더 많은 작업을 수행하므로, 파이썬 자체에서 포인터 객체가 필요하지 않으면 *byref()*를 사용하는 것이 더 빠릅니다.:

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc.sscanf(b"1 3.14 Hello", b"%d %f %s",
...             byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
>>>
```

구조체와 공용체

Structures and unions must derive from the *Structure* and *Union* base classes which are defined in the *ctypes* module. Each subclass must define a *_fields_* attribute. _fields_ must be a list of *2-tuples*, containing a *field name* and a *field type*.

필드형은 *c_int*와 같은 *ctypes* 형이거나 다른 파생된 *ctypes* 형(구조체, 공용체, 배열, 포인터)이어야 합니다.

다음은 *x* 및 *y*라는 두 개의 정수가 포함된 POINT 구조체의 간단한 예제이며, 생성자에서 구조체를 초기화하는 방법도 보여줍니다:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10 20
>>> point = POINT(y=5)
>>> print(point.x, point.y)
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: too many initializers
>>>
```

그러나, 훨씬 복잡한 구조를 만들 수 있습니다. 구조체는 필드형으로 구조체를 사용하여 다른 구조체를 포함할 수 있습니다.

다음은 *upperleft* 및 *lowerright*라는 두 개의 POINT를 포함하는 RECT 구조체입니다:

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
0 0
>>>
```

중첩된 구조체는 여러 가지 방법으로 생성자에서 초기화할 수 있습니다:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

Field *descriptor*s can be retrieved from the *class*, they are useful for debugging because they can provide useful information. See `CField`:

```
>>> POINT.x
<ctypes.CField 'x' type=c_int, ofs=0, size=4>
>>> POINT.y
<ctypes.CField 'y' type=c_int, ofs=4, size=4>
>>>
```

경고

`ctypes`는 비트 필드가 있는 공용체나 구조체를 값으로 함수에 전달할 수 없습니다. 32비트 x86에서 작동할 수 있지만, 일반적으로 작동은 라이브러리가 보증하지 않습니다. 비트 필드가 있는 공용체와 구조체는 항상 포인터로 함수에 전달되어야 합니다.

Structure/union layout, alignment and byte order

By default, Structure and Union fields are laid out in the same way the C compiler does it. It is possible to override this behavior entirely by specifying a `_layout_` class attribute in the subclass definition; see the attribute documentation for details.

It is possible to specify the maximum alignment for the fields by setting the `_pack_` class attribute to a positive integer. This matches what `#pragma pack(n)` does in MSVC.

It is also possible to set a minimum alignment for how the subclass itself is packed in the same way `#pragma align(n)` works in MSVC. This can be achieved by specifying a *_align_* class attribute in the subclass definition.

*ctypes*는 구조체와 공용체에 기본(native) 바이트 순서를 사용합니다. 기본이 아닌 바이트 순서로 구조체를 만들려면 *BigEndianStructure*, *LittleEndianStructure*, *BigEndianUnion* 및 *LittleEndianUnion* 베이스 클래스 중 하나를 사용할 수 있습니다. 이러한 클래스들은 포인터 필드를 포함할 수 없습니다.

구조체와 공용체의 비트 필드

It is possible to create structures and unions containing bit fields. Bit fields are only possible for integer fields, the bit width is specified as the third item in the *_fields_* tuples:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print(Int.first_16)
<ctypes.CField 'first_16' type=c_int, ofs=0, bit_size=16, bit_offset=0>
>>> print(Int.second_16)
<ctypes.CField 'second_16' type=c_int, ofs=0, bit_size=16, bit_offset=16>
```

It is important to note that bit field allocation and layout in memory are not defined as a C standard; their implementation is compiler-specific. By default, Python will attempt to match the behavior of a "native" compiler for the current platform. See the *_layout_* attribute for details on the default behavior and how to change it.

배열

배열은 같은 형의 고정 된 수의 인스턴스를 포함하는 시퀀스입니다.

배열형을 만드는 데 권장되는 방법은 데이터형에 양의 정수를 곱하는 것입니다:

```
TenPointsArrayType = POINT * 10
```

다음은 다소 인공적인 데이터형의 예입니다. 다른 항목들과 함께 4개의 POINT를 포함하는 구조체입니다:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print(len(MyStruct().point_array))
4
>>>
```

인스턴스는 클래스를 호출하는 일반적인 방법으로 만들어집니다:

```
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

위 코드는 배열 내용이 0으로 초기화되기 때문에, 일련의 `0 0` 줄을 인쇄합니다.

올바른 형의 초기화자를 지정할 수도 있습니다:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(ii)
<c_long_Array_10 object at 0x...>
>>> for i in ii: print(i, end=" ")
...
1 2 3 4 5 6 7 8 9 10
>>>
```

포인터

포인터 인스턴스는 *ctypes* 형에 *pointer()* 함수를 호출해서 만듭니다:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

포인터 인스턴스는 포인터가 가리키는 객체(위에서는 `i` 객체)를 반환하는 *contents* 어트리뷰트를 가집니다:

```
>>> pi.contents
c_long(42)
>>>
```

*ctypes*에는 OOR(원래 객체 반환, original object return)이 없다는 것에 유의하십시오. 어트리뷰트를 가져올 때마다 (동등하지만) 새로운 객체를 만듭니다:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

다른 *c_int* 인스턴스를 포인터의 contents 어트리뷰트에 대입하면 포인터는 이 인스턴스가 저장되어있는 메모리 위치를 가리키게 됩니다:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

포인터 인스턴스는 정수로도 인덱싱할 수 있습니다.:

```
>>> pi[0]
99
>>>
```

정수 인덱스에 대입하면 가리키고 있는 값이 바뀝니다:

```
>>> print(i)
c_long(99)
>>> pi[0] = 22
>>> print(i)
c_long(22)
>>>
```

0이 아닌 인덱스를 사용할 수도 있지만, C에서와 마찬가지로 자신이 하는 일을 알아야 합니다: 임의의 메모리 위치를 액세스하거나 변경할 수 있습니다. 일반적으로 C 함수에서 포인터를 받고, 포인터가 실제로 단일 항목 대신 배열을 가리키는 것을 알 때만 이 기능을 사용합니다.

장막 뒤에서, *pointer()* 함수는 단순히 포인터 인스턴스를 만드는 것 이상을 수행합니다. 먼저 포인터 형을 만들어야 합니다. 이것은 임의의 *ctypes* 형을 받아들이고, 새로운 형을 반환하는 *POINTER()* 함수로 수행됩니다:

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

인자 없이 포인터형을 호출하면 `NULL` 포인터가 만들어집니다. `NULL` 포인터는 `False` 논릿값을 갖습니다:

```
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
>>>
```

*ctypes*는 포인터를 역참조(dereference)할 때 `NULL`인지 확인합니다(하지만 `NULL`이 아닌 잘못된 포인터를 역참조하면 파이썬을 충돌시킵니다):

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

Thread safety without the GIL

From Python 3.13 onward, the *GIL* can be disabled on *free threaded* builds. In ctypes, reads and writes to a single object concurrently is safe, but not across multiple objects:

```
>>> number = c_int(42)
>>> pointer_a = pointer(number)
>>> pointer_b = pointer(number)
```

In the above, it's only safe for one object to read and write to the address at once if the GIL is disabled. So, `pointer_a` can be shared and written to across multiple threads, but only if `pointer_b` is not also attempting to do the same. If this is an issue, consider using a *threading.Lock* to synchronize access to memory:

```
>>> import threading
>>> lock = threading.Lock()
>>> # Thread 1
>>> with lock:
...     pointer_a.contents = 24
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> # Thread 2
>>> with lock:
...     pointer_b.contents = 42
```

형 변환

Usually, ctypes does strict type checking. This means, if you have `POINTER(c_int)` in the *argtypes* list of a function or as the type of a member field in a structure definition, only instances of exactly the same type are accepted. There are some exceptions to this rule, where ctypes accepts other objects. For example, you can pass compatible array instances instead of pointer types. So, for `POINTER(c_int)`, ctypes accepts an array of c_int:

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print(bar.values[i])
...
1
2
3
>>>
```

In addition, if a function argument is explicitly declared to be a pointer type (such as `POINTER(c_int)`) in *argtypes*, an object of the pointed type (`c_int` in this case) can be passed to the function. ctypes will apply the required *byref()* conversion in this case automatically.

POINTER 형 필드를 `NULL`로 설정하려면, `None`을 대입할 수 있습니다:

```
>>> bar.values = None
>>>
```

때에 따라 호환되지 않는 형의 인스턴스가 있을 수 있습니다. C에서는, 한 형을 다른 형으로 강제 변환(cast)할 수 있습니다. *ctypes*는 같은 방식으로 사용할 수 있는 *cast()* 함수를 제공합니다. 위에 정의된 `Bar` 구조체는 `values` 필드에 대해 `POINTER(c_int)` 포인터나 *c_int* 배열을 받아들이지만 다른 형의 인스턴스는 허용하지 않습니다:

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long␣
↪instance
>>>
```

이럴 때, *cast()* 함수가 편리합니다.

cast() 함수는 ctypes 인스턴스를 다른 ctypes 데이터형에 대한 포인터로 변환하는 데 사용할 수 있습니다. *cast()*는 두 개의 매개 변수, 어떤 종류의 포인터로 변환될 수 있는 ctypes 객체와 ctypes 포인터형을 받아들입니다. 첫 번째 인자와 같은 메모리 블록을 참조하는 두 번째 인자의 인스턴스를 반환합니다:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

따라서, *cast()*는 `Bar` 구조체의 `values` 필드에 대입하는 데 사용할 수 있습니다:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print(bar.values[0])
0
>>>
```

불완전한 형

불완전한 형은 멤버가 아직 지정되지 않은 구조체, 공용체 또는 배열입니다. C에서, 이것들은 나중에 정의되는 전방 선언(forward declaration)으로 지정됩니다:

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

ctypes 코드로 그대로 옮기면 이렇게 되지만, 작동하지는 않습니다:

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

because the new `class cell` is not available in the class statement itself. In *ctypes*, we can define the `cell` class and set the *_fields_* attribute later, after the class statement:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

해 봅시다. 우리는 두 개의 `cell` 인스턴스를 만들고, 서로를 가리키도록 한 다음, 마지막으로 포인터 체인을 몇 번 따라갑니다:

```
>>> c1 = cell()
>>> c1.name = b"foo"
>>> c2 = cell()
>>> c2.name = b"bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

콜백 함수

*ctypes*는 파이썬 콜러블로부터 C에서 호출 가능한 함수 포인터를 만들 수 있습니다. 이들은 때로 콜백 함수(*callback functions*)라고 불립니다.

먼저, 콜백 함수를 위한 클래스를 만들어야 합니다. 클래스는 호출 규칙, 반환형 및 이 함수가 받는 인자의 수와 형을 알고 있습니다.

CFUNCTYPE() 팩토리 함수는 `cdecl` 호출 규칙을 사용하여 콜백 함수의 형을 만듭니다. 윈도우에서, *WINFUNCTYPE()* 팩토리 함수는 `stdcall` 호출 규칙을 사용하여 콜백 함수 형을 만듭니다.

이러한 팩토리 함수는 모두 첫 번째 인자로 결과 형을, 나머지 인자로 콜백 함수가 기대하는 인자 형들로 호출됩니다.

I will present an example here which uses the standard C library' s `qsort()` function, that is used to sort items with the help of a callback function. `qsort()` will be used to sort an array of integers:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

`qsort()` must be called with a pointer to the data to sort, the number of items in the data array, the size of one item, and a pointer to the comparison function, the callback. The callback will then be called with two pointers to items, and it must return a negative integer if the first item is smaller than the second, a zero if they are equal, and a positive integer otherwise.

따라서 콜백 함수는 정수에 대한 포인터들을 받고 정수를 반환해야 합니다. 먼저 콜백 함수를 위한 **형**을 만듭니다:

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

시작하기 위해, 전달된 값을 보여주는 간단한 콜백이 있습니다:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

결과:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

이제 실제로 두 항목을 비교하여 유용한 결과를 반환할 수 있습니다:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>>
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

쉽게 확인할 수 있듯이, 배열은 이제 정렬되었습니다:

```
>>> for i in ia: print(i, end=" ")
...
1 5 7 33 99
>>>
```

함수 팩토리는 데코레이터 팩토리로 사용할 수 있으므로, 다음과 같이 작성할 수도 있습니다:

```
>>> @CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
... def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>> qsort(ia, len(ia), sizeof(c_int), py_cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

참고

C 코드에서 사용되는 동안 *CFUNCTYPE()* 객체에 대한 참조를 유지해야 합니다. *ctypes*가 참조를 유지하지는 않으며, 여러분이 하지 않는다면 가비지 수집되어, 콜백이 발생할 때 프로그램이 충돌할 수 있습니다.

또한, 콜백 함수가 파이썬 제어 바깥에서 만들어진 스레드(예를 들어, 콜백을 호출하는 외부 코드)에서 호출되면, ctypes는 모든 호출에 대해 새로운 더미 파이썬 스레드를 만듭니다. 이 동작은 대부분 적합하지만, *threading.local*에 저장된 값은, 같은 C 스레드에서 호출되는 여러 콜백에서 살아남을 수 없음을 뜻합니다.

dll에서 내 보낸 값을 액세스하기

Some shared libraries not only export functions, they also export variables. An example in the Python library itself is the `Py_Version`, Python runtime version number encoded in a single constant integer.

ctypes can access values like this with the *in_dll()* class methods of the type. *pythonapi* is a predefined symbol giving access to the Python C api:

```
>>> version = ctypes.c_int.in_dll(ctypes.pythonapi, "Py_Version")
>>> print(hex(version.value))
0x30c00a0
```

포인터의 사용법도 보여주는 확장 예제는 파이썬이 내 보낸 `PyImport_FrozenModules` 포인터에 액세스합니다.

해당 값에 대한 문서를 인용하면:

> This pointer is initialized to point to an array of `_frozen` records, terminated by one whose members are all `NULL` or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

따라서, 이 포인터를 조작하는 것이 유용할 수도 있습니다. 예제 크기를 제한하기 위해, *ctypes*로 이 테이블을 읽는 방법만 보여줍니다:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int),
...                 ("get_code", POINTER(c_ubyte)),  # Function pointer
...                ]
...
>>>
```

We have defined the `_frozen` data type, so we can get the pointer to the table:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "_PyImport_FrozenBootstrap")
>>>
```

`table`이 `struct_frozen` 레코드의 배열에 대한 포인터이므로, 이터레이션할 수 있습니다. 하지만 포인터는 크기가 없으므로 루프를 종료하는 방법이 필요합니다. 조만간 액세스 위반 등으로 인해 충돌이 발생할 수 있으므로, `NULL` 엔트리가 발견되자마자 루프에서 벗어나는 것이 좋습니다:

```
>>> for item in table:
...     if item.name is None:
...         break
...     print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
zipimport 12345
>>>
```

표준 파이썬이 프로즌 모듈과 프로즌 패키지(음수 `size` 멤버로 표시됨)를 가지고 있다는 사실은 잘 알려지지 않았으며, 테스트용으로만 사용됩니다. 예를 들어 `import __hello__`를 시도해보십시오.

의외의 것들

*ctypes*에는 여러분이 기대하는 것과 실제로 일어나는 것이 다른 가장자리가 있습니다.

다음 예제를 고려해보십시오:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
1 2 3 4
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
3 4 3 4
>>>
```

흠. 아마도 마지막 문장이 `3 4 1 2`를 인쇄할 것으로 기대했을 겁니다. 어떻게 된 걸까요? 위의 `rc.a, rc.b = rc.b, rc.a` 줄의 단계는 다음과 같습니다:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

`temp0` 과 `temp1`은 여전히 위의 `rc` 객체의 내부 버퍼를 사용하는 객체입니다. 따라서 `rc.a = temp0`를 실행하면 `temp0`의 버퍼 내용이 `rc`의 버퍼로 복사됩니다. 이것은, 결과적으로 `temp1`의 내용을 변경합니다. 따라서 마지막 대입인 `rc.b = temp1`은 기대하는 효과를 주지 못합니다.

Structure, Union 및 Array에서 서브 객체를 가져오는 것은 서브 객체를 복사하지 않고, 대신 루트 객체의 하부 버퍼에 액세스하는 래퍼 객체를 가져온다는 점에 유의하십시오.

예상과 다른 행동을 하는 또 다른 예는 다음과 같습니다:

```
>>> s = c_char_p()
>>> s.value = b"abc def ghi"
>>> s.value
b'abc def ghi'
>>> s.value is s.value
False
>>>
```

참고

*c_char_p*로 인스턴스를 만든 객체는 바이트열이나 정수로 설정된 value만 가질 수 있습니다.

`False`를 인쇄하는 이유는 무엇일까요? ctypes 인스턴스는 메모리 블록과 메모리 내용에 액세스하는 어떤 디스크립터를 포함하는 객체입니다. 메모리 블록에 파이썬 객체를 저장하면 객체 자체를 저장하지 않고, 대신 객체의 **내용**을 저장합니다. 내용에 다시 액세스하면 매번 새로운 파이썬 객체가 생성됩니다!

가변 크기 데이터형

*ctypes*는 가변 크기 배열과 구조체에 대한 일부 지원을 제공합니다.

resize() 함수는 기존 ctypes 객체의 메모리 버퍼 크기를 바꾸는 데 사용할 수 있습니다. 이 함수는 객체를 첫 번째 인자로 가져오고, 바이트 단위의 요청 된 크기를 두 번째 인자로 가져옵니다. 메모리 블록을 객체 형이 지정하는 원래 메모리 블록보다 작게 만들 수 없습니다. 시도하면 *ValueError*가 발생합니다:

```
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> sizeof(type(short_array))
8
>>>
```

훌륭합니다, 하지만 이 배열에 포함된 추가 요소에 어떻게 액세스할 수 있습니까? 형은 여전히 4개의 요소만 알고 있으므로, 다른 요소에 액세스하면 에러가 발생합니다:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

*ctypes*에서 가변 크기 데이터형을 사용하는 또 다른 방법은, 파이썬의 동적 특성을 사용하고 필요한 크기가 이미 알려진 후 매번 데이터형을 (재) 정의하는 것입니다.

16.9.2 ctypes 레퍼런스

공유 라이브러리 찾기

컴파일 언어로 프로그래밍할 때, 공유 라이브러리는 프로그램을 컴파일/링크할 때와 프로그램을 실행할 때 액세스 됩니다.

The purpose of the *find_library()* function is to locate a library in a way similar to what the compiler or runtime loader does (on platforms with several versions of a shared library the most recent should be loaded), while the ctypes library loaders act like when a program is run, and call the runtime loader directly.

The `ctypes.util` module provides a function which can help to determine the library to load.

`ctypes.util.`**`find_library`**(*name*)

> 라이브러리를 찾아서 경로명을 반환하려고 시도합니다. *name*은 *lib* 같은 접두사, `.so`, `.dylib` 또는 버전 번호와 같은 접미사가 없는 라이브러리 이름입니다 (이것은 posix 링커 옵션 `-l`에 사용되는 양식입니다). 라이브러리를 찾을 수 없으면 `None`을 반환합니다.

정확한 기능은 시스템에 따라 다릅니다.

On Linux, *find_library()* tries to run external programs (`/sbin/ldconfig`, `gcc`, `objdump` and `ld`) to find the library file. It returns the filename of the library file.

버전 3.6에서 변경: 리눅스에서, 다른 수단으로 라이브러리를 찾을 수 없으면, 라이브러리 검색 시 환경 변수 `LD_LIBRARY_PATH`의 값이 사용됩니다.

여기 예제가 있습니다:

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

On macOS and Android, *find_library()* uses the system' s standard naming schemes and paths to locate the library, and returns a full pathname if successful:

```
>>> from ctypes.util import find_library
>>> find_library("c")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

On Windows, *find_library()* searches along the system search path, and returns the full pathname, but since there is no predefined naming scheme a call like `find_library("c")` will fail and return `None`.

If wrapping a shared library with *ctypes*, it *may* be better to determine the shared library name at development time, and hardcode that into the wrapper module instead of using *find_library()* to locate the library at runtime.

Listing loaded shared libraries

When writing code that relies on code loaded from shared libraries, it can be useful to know which shared libraries have already been loaded into the current process.

The `ctypes.util` module provides the *dllist()* function, which calls the different APIs provided by the various platforms to help determine which shared libraries have already been loaded into the current process.

The exact output of this function will be system dependent. On most platforms, the first entry of this list represents the current process itself, which may be an empty string. For example, on glibc-based Linux, the return may look like:

```
>>> from ctypes.util import dllist
>>> dllist()
['', 'linux-vdso.so.1', '/lib/x86_64-linux-gnu/libm.so.6', '/lib/x86_64-linux-gnu/
↪libc.so.6', ... ]
```

공유 라이브러리 로드하기

공유 라이브러리를 파이썬 프로세스에 로드하는 방법에는 여러 가지가 있습니다. 한 가지 방법은 다음 클래스 중 하나의 인스턴스를 만드는 것입니다:

class `ctypes.`**`CDLL`**(*name, mode=DEFAULT_MODE, handle=None, use_errno=False, use_last_error=False, winmode=None*)

Instances of this class represent loaded shared libraries. Functions in these libraries use the standard C calling convention, and are assumed to return `int`.

윈도우에서는 DLL 이름이 존재하더라도 *CDLL* 인스턴스 생성이 실패할 수 있습니다. 로드된 DLL의 종속 DLL을 찾을 수 없을 때, "[*WinError 126*] *The specified module could not be found*" 메시지로 *OSError* 에러가 발생합니다. 윈도우 API가 정보를 반환하지 않아서 이 에러 메시지에는 누락된 DLL의 이름이 포함되어 있지 않고, 이 에러를 진단하기 어렵게 만듭니다. 이 에러를 해결하고 찾을 수 없는 DLL을 확인하려면, 윈도우 디버깅과 추적 도구를 사용하여 종속 DLL 목록을 찾고 찾을 수 없는 DLL을 확인해야 합니다.

버전 3.12에서 변경: The *name* parameter can now be a *path-like object*.

더 보기

Microsoft DUMPBIN tool – DLL 종속 항목을 찾는 도구.

class `ctypes.`**`OleDLL`**(*name, mode=DEFAULT_MODE, handle=None, use_errno=False, use_last_error=False, winmode=None*)

Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling

convention, and are assumed to return the windows specific *HRESULT* code. *HRESULT* values contain information specifying whether the function call failed or succeeded, together with additional error code. If the return value signals a failure, an *OSError* is automatically raised.

가용성: Windows

버전 3.3에서 변경: *WindowsError* used to be raised, which is now an alias of *OSError*.

버전 3.12에서 변경: The *name* parameter can now be a *path-like object*.

class ctypes.**WinDLL**(*name, mode=DEFAULT_MODE, handle=None, use_errno=False, use_last_error=False, winmode=None*)

Instances of this class represent loaded shared libraries, functions in these libraries use the `stdcall` calling convention, and are assumed to return `int` by default.

가용성: Windows

버전 3.12에서 변경: The *name* parameter can now be a *path-like object*.

파이썬 전역 인터프리터 록은, 이 라이브러리들이 내보낸 함수를 호출하기 전에 해제되고 나중에 다시 획득됩니다.

class ctypes.**PyDLL**(*name, mode=DEFAULT_MODE, handle=None*)

이 클래스의 인스턴스는 *CDLL* 인스턴스처럼 동작합니다. 단, 파이썬 GIL이 함수 호출 중에 릴리스되지 않고, 함수 실행 후 파이썬 에러 플래그가 확인된다는 점만 다릅니다. 에러 플래그가 설정되면 파이썬 예외가 발생합니다.

따라서, 이것은 파이썬 C API 함수를 직접 호출하는 경우에만 유용합니다.

버전 3.12에서 변경: The *name* parameter can now be a *path-like object*.

All these classes can be instantiated by calling them with at least one argument, the pathname of the shared library. If you have an existing handle to an already loaded shared library, it can be passed as the `handle` named parameter, otherwise the underlying platform' s `dlopen()` or `LoadLibrary()` function is used to load the library into the process, and to get a handle to it.

mode 매개 변수는 라이브러리가 로드되는 방법을 지정하는 데 사용될 수 있습니다. 자세한 내용은, *dlopen(3)* 매뉴얼 페이지를 참조하십시오. 윈도우에서는, *mode*가 무시됩니다. posix 시스템에서는 RTLD_NOW가 항상 추가되며 구성할 수 없습니다.

The *use_errno* parameter, when set to true, enables a ctypes mechanism that allows accessing the system *errno* error number in a safe way. *ctypes* maintains a thread-local copy of the system' s *errno* variable; if you call foreign functions created with `use_errno=True` then the *errno* value before the function call is swapped with the ctypes private copy, the same happens immediately after the function call.

ctypes.get_errno() 함수는 ctypes 내부 사본의 값을 반환하고, *ctypes.set_errno()* 함수는 ctypes 내부 사본을 새 값으로 변경하고 이전 값을 반환합니다.

The *use_last_error* parameter, when set to true, enables the same mechanism for the Windows error code which is managed by the *GetLastError()* and `SetLastError()` Windows API functions; *ctypes.get_last_error()* and *ctypes.set_last_error()* are used to request and change the ctypes private copy of the windows error code.

The *winmode* parameter is used on Windows to specify how the library is loaded (since *mode* is ignored). It takes any value that is valid for the Win32 API `LoadLibraryEx` flags parameter. When omitted, the default is to use the flags that result in the most secure DLL load, which avoids issues such as DLL hijacking. Passing the full path to the DLL is the safest way to ensure the correct library and dependencies are loaded.

버전 3.8에서 변경: *winmode* 매개 변수가 추가되었습니다.

ctypes.**RTLD_GLOBAL**

mode 매개 변수에 사용하는 플래그. 이 플래그를 사용할 수 없는 플랫폼에서는, 정수 0으로 정의됩니다.

ctypes.**RTLD_LOCAL**

mode 매개 변수에 사용하는 플래그. 이 플래그를 사용할 수 없는 플랫폼에서는, *RTLD_GLOBAL*과 같습니다.

ctypes.**DEFAULT_MODE**

공유 라이브러리를 로드하는 데 사용되는 기본 모드. OSX 10.3에서는 *RTLD_GLOBAL*이고, 그렇지 않으면 *RTLD_LOCAL*과 같습니다.

이 클래스들의 인스턴스는 공개 메서드가 없습니다. 공유 라이브러리가 내보낸 함수는 어트리뷰트나 인덱스로 액세스할 수 있습니다. 어트리뷰트를 통해 함수에 액세스하면 결과가 캐시 되므로 반복적으로 액세스할 때 매번 같은 객체가 반환됨에 유의하십시오. 반면에 인덱스를 통해 액세스하면 매번 새로운 객체가 반환됩니다:

```
>>> from ctypes import CDLL
>>> libc = CDLL("libc.so.6")  # On Linux
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

다음 공개 어트리뷰트를 사용할 수 있습니다. 내보낸 함수 이름과의 충돌을 피하고자 이름은 밑줄로 시작합니다:

PyDLL.**_handle**

라이브러리에 액세스하는 데 사용되는 시스템 핸들.

PyDLL.**_name**

생성자에서 전달된 라이브러리의 이름.

Shared libraries can also be loaded by using one of the prefabricated objects, which are instances of the `LibraryLoader` class, either by calling the `LoadLibrary()` method, or by retrieving the library as attribute of the loader instance.

class ctypes.**LibraryLoader**(*dlltype*)

공유 라이브러리를 로드하는 클래스. *dlltype*은 `CDLL`, `PyDLL`, `WinDLL` 또는 `OleDLL` 형 중 하나여야 합니다.

`__getattr__()` has special behavior: It allows loading a shared library by accessing it as attribute of a library loader instance. The result is cached, so repeated attribute accesses return the same library each time.

LoadLibrary(*name*)

공유 라이브러리를 프로세스에 로드하고 반환합니다. 이 메서드는 항상 라이브러리의 새 인스턴스를 반환합니다.

다음과 같은 사전 작성된 로더를 사용할 수 있습니다:

ctypes.**cdll**

`CDLL` 인스턴스를 만듭니다.

ctypes.**windll**

Creates `WinDLL` instances.

가용성: Windows

ctypes.**oledll**

Creates `OleDLL` instances.

가용성: Windows

ctypes.**pydll**

`PyDLL` 인스턴스를 만듭니다.

C 파이썬 API에 직접 액세스하기 위해, 바로 사용할 수 있는 파이썬 공유 라이브러리 객체가 제공됩니다:

ctypes.**pythonapi**

An instance of `PyDLL` that exposes Python C API functions as attributes. Note that all these functions are assumed to return C `int`, which is of course not always the truth, so you have to assign the correct `restype` attribute to use these functions.

이러한 객체를 통해 라이브러리를 로드하면 라이브러리를 로드하는 데 사용된 이름인 `name` 문자열 인자로 감사 이벤트(*auditing event*) `ctypes.dlopen`을 발생시킵니다.

로드된 라이브러리에서 함수에 액세스하면 인자 `library`(라이브러리 객체)와 `name`(문자열이나 정수로 심볼의 이름)으로 감사 이벤트 `ctypes.dlsym`을 발생시킵니다.

객체 대신 라이브러리 핸들만 사용 가능한 경우, 함수에 액세스하면 인자 `handle`(원시 라이브러리 핸들)과 `name`으로 감사 이벤트 `ctypes.dlsym/handle`을 발생시킵니다.

외부 함수

As explained in the previous section, foreign functions can be accessed as attributes of loaded shared libraries. The function objects created in this way by default accept any number of arguments, accept any ctypes data instances as arguments, and return the default result type specified by the library loader.

They are instances of a private local class `_FuncPtr` (not exposed in `ctypes`) which inherits from the private *`_CFuncPtr`* class:

```
>>> import ctypes
>>> lib = ctypes.CDLL(None)
>>> issubclass(lib._FuncPtr, ctypes._CFuncPtr)
True
>>> lib._FuncPtr is ctypes._CFuncPtr
False
```

class ctypes.**_CFuncPtr**

C 호출 가능한 외부 함수의 베이스 클래스.

외부 함수의 인스턴스는 C 호환 데이터형이기도 합니다; C 함수 포인터를 나타냅니다.

이 동작은 외부 함수 객체의 특수 어트리뷰트에 대입하여 사용자 정의할 수 있습니다.

restype

Assign a ctypes type to specify the result type of the foreign function. Use `None` for `void`, a function not returning anything.

It is possible to assign a callable Python object that is not a ctypes type, in this case the function is assumed to return a C `int`, and the callable will be called with this integer, allowing further processing or error checking. Using this is deprecated, for more flexible post processing or error checking use a ctypes data type as `restype` and assign a callable to the *`errcheck`* attribute.

argtypes

ctypes 형의 튜플을 대입하여 함수가 받아들이는 인자 형을 지정합니다. `stdcall` 호출 규칙을 사용하는 함수는 이 튜플의 길이와 같은 수의 인자로만 호출할 수 있습니다; C 호출 규칙을 사용하는 함수는 추가적인 지정되지 않은 인자도 허용합니다.

When a foreign function is called, each actual argument is passed to the *`from_param()`* class method of the items in the *`argtypes`* tuple, this method allows adapting the actual argument to an object that the foreign function accepts. For example, a *`c_char_p`* item in the *`argtypes`* tuple will convert a string passed as argument into a bytes object using ctypes conversion rules.

New: It is now possible to put items in argtypes which are not ctypes types, but each item must have a *`from_param()`* method which returns a value usable as argument (integer, string, ctypes instance). This allows defining adapters that can adapt custom objects as function parameters.

errcheck

이 어트리뷰트에 파이썬 함수나 다른 콜러블을 대입합니다. 콜러블은 3개 이상의 인자로 호출됩니다:

callable(*result*, *func*, *arguments*)

result is what the foreign function returns, as specified by the `restype` attribute.

*func*는 외부 함수 객체 자체이며, 같은 콜러블 객체를 재사용하여 여러 함수의 결과를 확인하거나 사후 처리할 수 있도록 합니다.

*arguments*는 원래 함수 호출에 전달된 매개 변수를 포함하는 튜플입니다. 사용된 인자에 따라 동작을 특수화할 수 있도록 합니다.

이 함수가 반환하는 객체는 외부 함수 호출에서 반환되지만, 결괏값을 확인하고 외부 함수 호출이 실패하면 예외를 발생시킬 수도 있습니다.

On Windows, when a foreign function call raises a system exception (for example, due to an access violation), it will be captured and replaced with a suitable Python exception. Further, an auditing event `ctypes.set_exception` with argument `code` will be raised, allowing an audit hook to replace the exception with its own.

Some ways to invoke foreign function calls as well as some of the functions in this module may raise an auditing event `ctypes.call_function` with arguments `function pointer` and `arguments`.

함수 프로토타입

함수 프로토타입의 인스턴스를 만들어서 외부 함수를 만들 수도 있습니다. 함수 프로토타입은 C의 함수 프로토타입과 비슷합니다; 구현을 정의하지 않고 함수(반환형, 인자형, 호출 규칙)를 설명합니다. 팩토리 함수는 원하는 결과형과 함수의 인자형들로 호출되어야 하며, 데코레이터 팩토리로 사용되어 `@wrapper` 문법을 통해 함수에 적용될 수 있습니다. 예제는 콜백 함수를 참조하십시오.

`ctypes.`**`CFUNCTYPE`**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)

반환된 함수 프로토타입은 표준 C 호출 규칙을 사용하는 함수를 만듭니다. 이 함수는 호출 중에 GIL을 해제합니다. *use_errno*를 참으로 설정하면, 시스템 `errno` 변수의 ctypes 내부 복사본이 호출 전후에 실제 `errno` 값과 교환됩니다; *use_last_error*는 윈도우 에러 코드에 대해 같은 일을 합니다.

`ctypes.`**`WINFUNCTYPE`**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)

The returned function prototype creates functions that use the `stdcall` calling convention. The function will release the GIL during the call. *use_errno* and *use_last_error* have the same meaning as above.

가용성: Windows

`ctypes.`**`PYFUNCTYPE`**(*restype*, **argtypes*)

반환된 함수 프로토타입은 파이썬 호출 규칙을 사용하는 함수를 만듭니다. 이 함수는 호출 도중 GIL을 해제하지 않습니다.

이러한 팩토리 함수로 만들어진 함수 프로토타입은 호출의 매개 변수 형과 수에 따라 다른 방법으로 인스턴스를 만들 수 있습니다:

`prototype`(*address*)

지정된 정수 주소에 있는 외부 함수를 반환합니다.

`prototype`(*callable*)

파이썬 *callable*로 C 호출 가능 함수(콜백 함수)를 만듭니다.

`prototype`(*func_spec*[, *paramflags*])

공유 라이브러리가 내보낸 외부 함수를 반환합니다. *func_spec*은 2-튜플 `(name_or_ordinal, library)` 여야 합니다. 첫 번째 항목은 내보낸 함수의 문자열 이름이거나, 작은 정수로 표현된 내보낸 함수의 서수(ordinal)입니다. 두 번째 항목은 공유 라이브러리 인스턴스입니다.

`prototype`(*vtbl_index*, *name*[, *paramflags*[, *iid*]])

COM 메서드를 호출할 외부 함수를 반환합니다. *vtbl_index*는 가상 함수 테이블에 대한 인덱스이며, 작은 음이 아닌 정수입니다. *name*은 COM 메서드의 이름입니다. *iid*는 확장 에러 보고에 사용되는 인터페이스 식별자를 가리키는 선택적 포인터입니다.

If *iid* is not specified, an `OSError` is raised if the COM method call fails. If *iid* is specified, a `COMError` is raised instead.

COM methods use a special calling convention: They require a pointer to the COM interface as first argument, in addition to those parameters that are specified in the `argtypes` tuple.

가용성: Windows

선택적 *paramflags* 매개 변수는 위에 설명된 기능보다 훨씬 많은 기능을 갖는 외부 함수 래퍼를 만듭니다.

paramflags must be a tuple of the same length as `argtypes`.

이 튜플의 각 항목에는 매개 변수에 대한 추가 정보가 들어 있으며, 한 개, 두 개 또는 세 개의 항목이 들어 있는 튜플이어야 합니다.

첫 번째 항목은 매개 변수의 방향 플래그 조합을 포함하는 정수입니다:

1
함수에 대한 입력 매개 변수를 지정합니다.

2
출력 매개 변수. 외부 함수가 값을 채웁니다.

4
기본값이 정수 0인 입력 매개 변수.

선택적인 두 번째 항목은 문자열 매개 변수 이름입니다. 이것이 지정되면, 이름있는 매개 변수로 외부 함수를 호출할 수 있습니다.

선택적 세 번째 항목은 이 매개 변수의 기본값입니다.

The following example demonstrates how to wrap the Windows `MessageBoxW` function so that it supports default parameters and named arguments. The C declaration from the windows header file is this:

```
WINUSERAPI int WINAPI
MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```

다음은 `ctypes`로 래핑하는 방법입니다:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCWSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCWSTR, LPCWSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", "Hello from␣
↪ctypes"), (1, "flags", 0)
>>> MessageBox = prototype(("MessageBoxW", windll.user32), paramflags)
```

이제 `MessageBox` 외부 함수를 다음과 같이 호출할 수 있습니다.:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

두 번째 예제는 출력 매개 변수를 보여줍니다. win32 `GetWindowRect` 함수는 지정된 창의 크기를 조회하는데, 호출자가 제공해야 하는 `RECT` 구조체로 복사합니다. 다음은 C 선언입니다:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
     HWND hWnd,
     LPRECT lpRect);
```

다음은 `ctypes`로 래핑하는 방법입니다:

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

출력 매개 변수가 있는 함수는, 하나뿐이면 자동으로 출력 매개 변숫값을 반환하고, 여러 개면 출력 매개 변숫값을 포함하는 튜플을 반환하므로, GetWindowRect 함수는 이제 호출되면 RECT 인스턴스를 반환합니다.

Output parameters can be combined with the *errcheck* protocol to do further output processing and error checking. The win32 `GetWindowRect` api function returns a `BOOL` to signal success or failure, so this function could do the error checking, and raises an exception when the api call failed:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

If the *errcheck* function returns the argument tuple it receives unchanged, *ctypes* continues the normal processing it does on the output parameters. If you want to return a tuple of window coordinates instead of a `RECT` instance, you can retrieve the fields in the function and return them instead, the normal processing will no longer take place:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

유틸리티 함수

`ctypes.`**`addressof`**(*obj*)

메모리 버퍼의 주소를 정수로 반환합니다. *obj*는 ctypes 형의 인스턴스여야 합니다.

인자 `obj`로 감사 이벤트 `ctypes.addressof`를 발생시킵니다.

`ctypes.`**`alignment`**(*obj_or_type*)

ctypes 형의 정렬 요구 사항을 반환합니다. *obj_or_type*는 ctypes 형이나 인스턴스여야 합니다.

`ctypes.`**`byref`**(*obj*[, *offset*])

*obj*에 대한 경량 포인터를 반환합니다. *obj*는 ctypes 형의 인스턴스여야 합니다. *offset*의 기본값은 0이며, 내부 포인터 값에 더해질 정수여야 합니다.

`byref(obj, offset)`는 이 C 코드에 해당합니다:

```
(((char *)&obj) + offset)
```

반환된 객체는 외부 함수 호출 매개 변수로만 사용할 수 있습니다. `pointer(obj)`와 비슷하게 동작하지만, 훨씬 빨리 만들어집니다.

`ctypes.`**`CopyComPointer`**(*src*, *dst*)

Copies a COM pointer from *src* to *dst* and returns the Windows specific `HRESULT` value.

If *src* is not `NULL`, its `AddRef` method is called, incrementing the reference count.

In contrast, the reference count of *dst* will not be decremented before assigning the new value. Unless *dst* is `NULL`, the caller is responsible for decrementing the reference count by calling its `Release` method when necessary.

가용성: Windows

Added in version 3.14.

ctypes.**cast**(*obj*, *type*)

이 함수는 C의 형 변환 연산자와 유사합니다. *obj*와 같은 메모리 블록을 가리키는 *type* 형의 새 인스턴스를 반환합니다. *type*은 포인터형이어야 하며, *obj*는 포인터로 해석될 수 있는 객체여야 합니다.

ctypes.**create_string_buffer**(*init*, *size=None*)

ctypes.**create_string_buffer**(*size*)

이 함수는 가변 문자 버퍼를 만듭니다. 반환된 객체는 `c_char`의 ctypes 배열입니다.

If *size* is given (and not `None`), it must be an `int`. It specifies the size of the returned array.

If the *init* argument is given, it must be `bytes`. It is used to initialize the array items. Bytes not initialized this way are set to zero (NUL).

If *size* is not given (or if it is `None`), the buffer is made one element larger than *init*, effectively adding a NUL terminator.

If both arguments are given, *size* must not be less than `len(init)`.

> **경고**
>
> If *size* is equal to `len(init)`, a NUL terminator is not added. Do not treat such a buffer as a C string.

For example:

```
>>> bytes(create_string_buffer(2))
b'\x00\x00'
>>> bytes(create_string_buffer(b'ab'))
b'ab\x00'
>>> bytes(create_string_buffer(b'ab', 2))
b'ab'
>>> bytes(create_string_buffer(b'ab', 4))
b'ab\x00\x00'
>>> bytes(create_string_buffer(b'abcdef', 2))
Traceback (most recent call last):
    ...
ValueError: byte string too long
```

`init`, `size` 인자로 감사 이벤트 `ctypes.create_string_buffer`를 발생시킵니다.

ctypes.**create_unicode_buffer**(*init*, *size=None*)

ctypes.**create_unicode_buffer**(*size*)

이 함수는 가변 유니코드 문자 버퍼를 만듭니다. 반환된 객체는 `c_wchar`의 ctypes 배열입니다.

The function takes the same arguments as `create_string_buffer()` except *init* must be a string and *size* counts `c_wchar`.

`init`, `size` 인자로 감사 이벤트 `ctypes.create_unicode_buffer`를 발생시킵니다.

ctypes.**DllCanUnloadNow**()

This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllCanUnloadNow function that the _ctypes extension dll exports.

가용성: Windows

ctypes.**DllGetClassObject**()

This function is a hook which allows implementing in-process COM servers with ctypes. It is called from the DllGetClassObject function that the `_ctypes` extension dll exports.

가용성: Windows

ctypes.util.**find_library**(*name*)

라이브러리를 찾아서 경로명을 반환하려고 시도합니다. *name*은 lib 같은 접두사, .so, .dylib 또는 버전 번호와 같은 접미사가 없는 라이브러리 이름입니다 (이것은 posix 링커 옵션 -l에 사용되는 양식입니다). 라이브러리를 찾을 수 없으면 None을 반환합니다.

정확한 기능은 시스템에 따라 다릅니다.

ctypes.util.**find_msvcrt**()

Returns the filename of the VC runtime library used by Python, and by the extension modules. If the name of the library cannot be determined, None is returned.

예를 들어, free(void *)에 대한 호출로 확장 모듈에 의해 할당된 메모리를 해제해야 하면, 메모리를 할당한 것과 같은 라이브러리에 있는 함수를 사용하는 것이 중요합니다.

가용성: Windows

ctypes.util.**dllist**()

Try to provide a list of paths of the shared libraries loaded into the current process. These paths are not normalized or processed in any way. The function can raise *OSError* if the underlying platform APIs fail. The exact functionality is system dependent.

On most platforms, the first element of the list represents the current executable file. It may be an empty string.

가용성: Windows, macOS, iOS, glibc, BSD libc, musl

Added in version 3.14.

ctypes.**FormatError**([*code*])

Returns a textual description of the error code *code*. If no error code is specified, the last error code is used by calling the Windows API function *GetLastError()*.

가용성: Windows

ctypes.**GetLastError**()

Returns the last error code set by Windows in the calling thread. This function calls the Windows GetLastError() function directly, it does not return the ctypes-private copy of the error code.

가용성: Windows

ctypes.**get_errno**()

호출하는 스레드에서 시스템 *errno* 변수의 ctypes 내부 복사본의 현재 값을 반환합니다.

인자 없이 감사 이벤트 ctypes.get_errno를 발생시킵니다.

ctypes.**get_last_error**()

Returns the current value of the ctypes-private copy of the system LastError variable in the calling thread.

가용성: Windows

인자 없이 감사 이벤트 ctypes.get_last_error를 발생시킵니다.

ctypes.**memmove**(*dst*, *src*, *count*)

표준 C memmove 라이브러리 함수와 같습니다: *count* 바이트를 *src*에서 *dst*로 복사합니다. *dst* 와 *src*는 정수이거나 포인터로 변환할 수 있는 ctypes 인스턴스여야 합니다.

ctypes.**memset**(*dst*, *c*, *count*)

표준 C memset 라이브러리 함수와 같습니다: 주소 *dst*의 메모리 블록을 값 *c*의 *count* 바이트로 채웁니다. *dst*는 주소를 지정하는 정수거나 ctypes 인스턴스여야 합니다.

ctypes.**POINTER**(*type*, /)

Create or return a ctypes pointer type. Pointer types are cached and reused internally, so calling this function repeatedly is cheap. *type* must be a ctypes type.

CPython 구현 상세: The resulting pointer type is cached in the __pointer_type__ attribute of *type*. It is possible to set this attribute before the first call to POINTER in order to set a custom pointer type. However, doing this is discouraged: manually creating a suitable pointer type is difficult without relying on implementation details that may change in future Python versions.

ctypes.**pointer**(*obj*, /)

Create a new pointer instance, pointing to *obj*. The returned object is of the type `POINTER(type(obj))`.

참고 사항: 객체에 대한 포인터를 단지 외부 함수 호출로 전달하려면 훨씬 빠른 `byref(obj)`를 사용해야 합니다.

ctypes.**resize**(*obj*, *size*)

이 함수는 *obj*의 내부 메모리 버퍼의 크기를 조정합니다. *obj*는 ctypes 형의 인스턴스여야 합니다. `sizeof(type(obj))`로 주어지는 객체 형의 원래 크기보다 버퍼를 작게 만들 수는 없지만, 버퍼를 확대할 수 있습니다.

ctypes.**set_errno**(*value*)

호출 중인 스레드의 시스템 *errno* 변수의 ctypes 내부 복사본의 현재 값을 *value*로 설정하고 이전 값을 반환합니다.

인자 `errno`로 감사 이벤트 `ctypes.set_errno`를 발생시킵니다.

ctypes.**set_last_error**(*value*)

Sets the current value of the ctypes-private copy of the system `LastError` variable in the calling thread to *value* and return the previous value.

가용성: Windows

인자 `error`로 감사 이벤트 `ctypes.set_last_error`를 발생시킵니다.

ctypes.**sizeof**(*obj_or_type*)

ctypes 형이나 인스턴스 메모리 버퍼의 크기를 바이트 단위로 반환합니다. C `sizeof` 연산자와 같은 일을 합니다.

ctypes.**string_at**(*ptr*, *size=-1*)

Return the byte string at *void *ptr*. If *size* is specified, it is used as size, otherwise the string is assumed to be zero-terminated.

Raises an *auditing event* `ctypes.string_at` with arguments `ptr`, `size`.

ctypes.**WinError**(*code=None*, *descr=None*)

Creates an instance of *OSError*. If *code* is not specified, *GetLastError()* is called to determine the error code. If *descr* is not specified, *FormatError()* is called to get a textual description of the error.

가용성: Windows

버전 3.3에서 변경: An instance of *WindowsError* used to be created, which is now an alias of *OSError*.

ctypes.**wstring_at**(*ptr*, *size=-1*)

Return the wide-character string at *void *ptr*. If *size* is specified, it is used as the number of characters of the string, otherwise the string is assumed to be zero-terminated.

Raises an *auditing event* `ctypes.wstring_at` with arguments `ptr`, `size`.

ctypes.**memoryview_at**(*ptr*, *size*, *readonly=False*)

Return a *memoryview* object of length *size* that references memory starting at *void *ptr*.

If *readonly* is true, the returned `memoryview` object can not be used to modify the underlying memory. (Changes made by other means will still be reflected in the returned object.)

This function is similar to *string_at()* with the key difference of not making a copy of the specified memory. It is a semantically equivalent (but more efficient) alternative to `memoryview((c_byte * size).from_address(ptr))`. (While *from_address()* only takes integers, *ptr* can also be given as a *ctypes.POINTER* or a *byref()* object.)

Raises an *auditing event* `ctypes.memoryview_at` with arguments `address`, `size`, `readonly`.

Added in version 3.14.

데이터형

class ctypes.**_CData**

이 비공개 클래스는 모든 ctypes 데이터형의 공통 베이스 클래스입니다. 무엇보다도, 모든 ctypes 형 인스턴스에는 C 호환 데이터를 보관하는 메모리 블록이 포함됩니다; 메모리 블록의 주소는 *addressof()* 도우미 함수에 의해 반환됩니다. 다른 인스턴스 변수는 *_objects*로 노출됩니다; 여기에는 메모리 블록에 포인터가 포함되어있을 때, 살려둘 필요가 있는 다른 파이썬 객체가 포함되어 있습니다.

ctypes 데이터형의 공통 메서드, 이것들은 모두 클래스 메서드입니다 (정확히 말하면, 메타 클래스의 메서드입니다):

from_buffer(*source*[, *offset*])

이 메서드는 *source* 객체의 버퍼를 공유하는 ctypes 인스턴스를 반환합니다. *source* 객체는 쓰기 가능한 버퍼 인터페이스를 지원해야 합니다. 선택적 *offset* 매개 변수는 source 버퍼의 오프셋을 바이트 단위로 지정합니다; 기본값은 0입니다. source 버퍼가 충분히 크지 않으면 *ValueError*가 발생합니다.

인자 `pointer`, `size`, `offset`으로 감사 이벤트 `ctypes.cdata/buffer`를 발생시킵니다.

from_buffer_copy(*source*[, *offset*])

이 메서드는 읽을 수 있어야 하는 *source* 객체 버퍼에서 버퍼를 복사하여 ctypes 인스턴스를 만듭니다. 선택적 *offset* 매개 변수는 원본 버퍼의 오프셋을 바이트 단위로 지정합니다. 기본값은 0입니다. 소스 버퍼가 충분히 크지 않으면 *ValueError*가 발생합니다.

인자 `pointer`, `size`, `offset`으로 감사 이벤트 `ctypes.cdata/buffer`를 발생시킵니다.

from_address(*address*)

이 메서드는 정수 *address*로 지정된 메모리를 사용하여 ctypes 형 인스턴스를 반환합니다.

이 메서드와 이 메서드를 간접적으로 호출하는 다른 것들은 인자 `address`로 감사 이벤트 `ctypes.cdata`를 발생시킵니다.

from_param(*obj*)

This method adapts *obj* to a ctypes type. It is called with the actual object used in a foreign function call when the type is present in the foreign function' s *argtypes* tuple; it must return an object that can be used as a function call parameter.

모든 ctypes 데이터형은 이 클래스 메서드의 기본 구현을 갖는데, *obj* 가 이 형의 인스턴스면 *obj*를 반환합니다. 일부 형은 다른 객체도 허용합니다.

in_dll(*library*, *name*)

이 메서드는 공유 라이브러리가 내보낸 ctypes 형 인스턴스를 반환합니다. *name*은 데이터를 내보내는 심볼의 이름이고, *library*는 로드된 공유 라이브러리입니다.

Common class variables of ctypes data types:

__pointer_type__

The pointer type that was created by calling *POINTER()* for corresponding ctypes data type. If a pointer type was not yet created, the attribute is missing.

Added in version 3.14.

ctypes 데이터형의 공통 인스턴스 변수:

_b_base_

때로 ctypes 데이터 인스턴스는 포함하는 메모리 블록을 소유하지 않고, 베이스 객체의 메모리 블록의 일부를 공유합니다. *_b_base_* 읽기 전용 멤버는 메모리 블록을 소유한 루트 ctypes 객체입니다.

_b_needsfree_

이 읽기 전용 변수는 ctypes 데이터 인스턴스가 메모리 블록을 스스로 할당했을 때 참이고, 그렇지 않으면 거짓입니다.

_objects

이 멤버는 None 이거나 메모리 블록 내용이 계속 유효하도록 유지되어야 하는 파이썬 객체를 포함하는 딕셔너리입니다. 이 객체는 디버깅을 위해서만 노출됩니다; 이 딕셔너리의 내용을 수정하지 마십시오.

기본 데이터형

class ctypes.**_SimpleCData**

이 비공개 클래스는 모든 기본 ctypes 데이터형의 베이스 클래스입니다. 여기에는 기본 ctypes 데이터형의 공통 어트리뷰트가 들어 있으므로 여기에서 언급합니다. *_SimpleCData*는 *_CData*의 서브클래스이므로, 메서드와 어트리뷰트를 상속받습니다. 포인터가 아니고 포인터를 포함하지 않는 ctypes 데이터형을 이제 피클 할 수 있습니다.

인스턴스에는 어트리뷰트가 하나 있습니다:

value

이 어트리뷰트는 인스턴스의 실제 값을 포함합니다. 정수형과 포인터형에서는 정수고, 문자형에서는 단일 문자 바이트열 객체나 문자열이고, 문자 포인터형에서는 파이썬 바이트열 객체나 문자열입니다.

ctypes 인스턴스에서 value 어트리뷰트를 조회하면, 대개 매번 새 객체가 반환됩니다. *ctypes*는 원래의 객체 반환을 구현하지 않습니다. 항상 새로운 객체가 만들어집니다. 다른 모든 ctypes 객체 인스턴스에서도 마찬가지입니다.

Fundamental data types, when returned as foreign function call results, or, for example, by retrieving structure field members or array items, are transparently converted to native Python types. In other words, if a foreign function has a *restype* of *c_char_p*, you will always receive a Python bytes object, *not* a *c_char_p* instance.

Subclasses of fundamental data types do *not* inherit this behavior. So, if a foreign functions restype is a subclass of *c_void_p*, you will receive an instance of this subclass from the function call. Of course, you can get the value of the pointer by accessing the value attribute.

다음은 기본 ctypes 데이터형입니다:

class ctypes.**c_byte**

Represents the C signed char datatype, and interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_char**

Represents the C char datatype, and interprets the value as a single character. The constructor accepts an optional string initializer, the length of the string must be exactly one character.

class ctypes.**c_char_p**

Represents the C char* datatype when it points to a zero-terminated string. For a general character pointer that may also point to binary data, POINTER(c_char) must be used. The constructor accepts an integer address, or a bytes object.

class ctypes.**c_double**

Represents the C double datatype. The constructor accepts an optional float initializer.

class ctypes.**c_longdouble**

Represents the C long double datatype. The constructor accepts an optional float initializer. On platforms where sizeof(long double) == sizeof(double) it is an alias to *c_double*.

class ctypes.**c_float**

Represents the C float datatype. The constructor accepts an optional float initializer.

class ctypes.**c_double_complex**

Represents the C double complex datatype, if available. The constructor accepts an optional *complex* initializer.

Added in version 3.14.

class ctypes.**c_float_complex**

Represents the C `float complex` datatype, if available. The constructor accepts an optional *complex* initializer.

Added in version 3.14.

class ctypes.**c_longdouble_complex**

Represents the C `long double complex` datatype, if available. The constructor accepts an optional *complex* initializer.

Added in version 3.14.

class ctypes.**c_int**

Represents the C `signed int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias to *c_long*.

class ctypes.**c_int8**

Represents the C 8-bit `signed int` datatype. It is an alias for *c_byte*.

class ctypes.**c_int16**

Represents the C 16-bit `signed int` datatype. Usually an alias for *c_short*.

class ctypes.**c_int32**

Represents the C 32-bit `signed int` datatype. Usually an alias for *c_int*.

class ctypes.**c_int64**

Represents the C 64-bit `signed int` datatype. Usually an alias for *c_longlong*.

class ctypes.**c_long**

Represents the C `signed long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_longlong**

Represents the C `signed long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_short**

Represents the C `signed short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_size_t**

C `size_t` 데이터형을 나타냅니다.

class ctypes.**c_ssize_t**

C `ssize_t` 데이터형을 나타냅니다.

Added in version 3.2.

class ctypes.**c_time_t**

Represents the C `time_t` datatype.

Added in version 3.12.

class ctypes.**c_ubyte**

Represents the C `unsigned char` datatype, it interprets the value as small integer. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_uint**

Represents the C `unsigned int` datatype. The constructor accepts an optional integer initializer; no overflow checking is done. On platforms where `sizeof(int) == sizeof(long)` it is an alias for *c_ulong*.

class ctypes.**c_uint8**

Represents the C 8-bit `unsigned int` datatype. It is an alias for *c_ubyte*.

class ctypes.**c_uint16**

Represents the C 16-bit `unsigned int` datatype. Usually an alias for `c_ushort`.

class ctypes.**c_uint32**

Represents the C 32-bit `unsigned int` datatype. Usually an alias for `c_uint`.

class ctypes.**c_uint64**

Represents the C 64-bit `unsigned int` datatype. Usually an alias for `c_ulonglong`.

class ctypes.**c_ulong**

Represents the C `unsigned long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_ulonglong**

Represents the C `unsigned long long` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_ushort**

Represents the C `unsigned short` datatype. The constructor accepts an optional integer initializer; no overflow checking is done.

class ctypes.**c_void_p**

Represents the C `void*` type. The value is represented as integer. The constructor accepts an optional integer initializer.

class ctypes.**c_wchar**

C `wchar_t` 데이터형을 나타내고, 값을 단일 문자 유니코드 문자열로 해석합니다. 생성자는 선택적 문자열 초기화자를 받아들입니다, 문자열의 길이는 정확히 한 문자여야 합니다.

class ctypes.**c_wchar_p**

Represents the C `wchar_t*` datatype, which must be a pointer to a zero-terminated wide character string. The constructor accepts an integer address, or a string.

class ctypes.**c_bool**

Represent the C `bool` datatype (more accurately, `_Bool` from C99). Its value can be `True` or `False`, and the constructor accepts any object that has a truth value.

class ctypes.**HRESULT**

Represents a `HRESULT` value, which contains success or error information for a function or method call.

가용성: Windows

class ctypes.**py_object**

Represents the C `PyObject*` datatype. Calling this without an argument creates a `NULL` `PyObject*` pointer.

버전 3.14에서 변경: `py_object` is now a *generic type*.

The `ctypes.wintypes` module provides quite some other Windows specific data types, for example `HWND`, `WPARAM`, or `DWORD`. Some useful structures like `MSG` or `RECT` are also defined.

구조화된 데이터형

class ctypes.**Union**(**args*, ***kw*)

네이티브 바이트 순서의 공용체를 위한 추상 베이스 클래스.

Unions share common attributes and behavior with structures; see `Structure` documentation for details.

class ctypes.**BigEndianUnion**(**args*, ***kw*)

Abstract base class for unions in *big endian* byte order.

Added in version 3.11.

class ctypes.**LittleEndianUnion**(**args*, ***kw*)

Abstract base class for unions in *little endian* byte order.

Added in version 3.11.

class ctypes.**BigEndianStructure**(**args*, ***kw*)

빅엔디안(*big endian*) 바이트 순서의 구조체를 위한 추상 베이스 클래스.

class ctypes.**LittleEndianStructure**(**args*, ***kw*)

리틀엔디안(*little endian*) 바이트 순서로의 구조체를 위한 추상 베이스 클래스.

Structures and unions with non-native byte order cannot contain pointer type fields, or any other data types containing pointer type fields.

class ctypes.**Structure**(**args*, ***kw*)

네이티브 바이트 순서의 구조체를 위한 추상 베이스 클래스.

구상 구조체와 공용체 형은 이 형 중 하나를 서브 클래싱하고 적어도 *_fields_* 클래스 변수를 정의해서 만들어야 합니다. *ctypes*는 직접 어트리뷰트 액세스로 필드를 읽고 쓸 수 있는 디스크립터를 만듭니다. 이것들은

fields

구조체 필드를 정의하는 시퀀스. 항목은 2-튜플이나 3-튜플이어야 합니다. 첫 번째 항목은 필드의 이름이고, 두 번째 항목은 필드의 형을 지정합니다; 모든 ctypes 데이터형이 될 수 있습니다.

*c_int*와 같은 정수형 필드에서는, 세 번째 선택적 항목을 지정할 수 있습니다. 필드의 비트 폭을 정의하는 작은 양의 정수여야 합니다.

필드 이름은 하나의 구조체나 공용체 내에서 고유해야 합니다. 이것은 검사되지 않습니다, 이름이 중복되면 하나의 필드만 액세스할 수 있습니다.

*_fields_* 클래스 변수를, Structure 서브 클래스를 정의하는 클래스 문 뒤에서 정의할 수 있습니다. 직접 또는 간접적으로 자신을 참조하는 데이터형을 만들 수 있게 합니다:

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...
                ]
```

The _fields_ class variable can only be set once. Later assignments will raise an *AttributeError*.

Additionally, the _fields_ class variable must be defined before the structure or union type is first used: an instance or subclass is created, *sizeof()* is called on it, and so on. Later assignments to _fields_ will raise an *AttributeError*. If _fields_ has not been set before such use, the structure or union will have no own fields, as if _fields_ was empty.

Sub-subclasses of structure types inherit the fields of the base class plus the *_fields_* defined in the sub-subclass, if any.

pack

An optional small integer that allows overriding the alignment of structure fields in the instance. *_pack_* must already be defined when *_fields_* is assigned, otherwise it will have no effect. Setting this attribute to 0 is the same as not setting it at all.

This is only implemented for the MSVC-compatible memory layout.

Deprecated since version 3.14, will be removed in version 3.19: For historical reasons, if _pack_ is non-zero, the MSVC-compatible layout will be used by default. On non-Windows platforms, this default is deprecated and is slated to become an error in Python 3.19. If it is intended, set *_layout_* to 'ms' explicitly.

align

An optional small integer that allows overriding the alignment of the structure when being packed or unpacked to/from memory. Setting this attribute to 0 is the same as not setting it at all.

Added in version 3.13.

layout

An optional string naming the struct/union layout. It can currently be set to:

- `"ms"`: the layout used by the Microsoft compiler (MSVC). On GCC and Clang, this layout can be selected with `__attribute__((ms_struct))`.
- `"gcc-sysv"`: the layout used by GCC with the System V or "SysV-like" data model, as used on Linux and macOS. With this layout, *_pack_* must be unset or zero.

If not set explicitly, `ctypes` will use a default that matches the platform conventions. This default may change in future Python releases (for example, when a new platform gains official support, or when a difference between similar platforms is found). Currently the default will be:

- On Windows: `"ms"`
- When *_pack_* is specified: `"ms"`. (This is deprecated; see *_pack_* documentation.)
- Otherwise: `"gcc-sysv"`

`_layout_` must already be defined when *_fields_* is assigned, otherwise it will have no effect.

Added in version 3.14.

anonymous

이름 없는(익명) 필드의 이름을 나열하는 선택적 시퀀스. *_fields_*가 대입될 때 *_anonymous_*는 이미 정의되어 있어야 합니다. 그렇지 않으면 아무 효과가 없습니다.

이 변수에 나열된 필드는 구조체나 공용체 형 필드여야 합니다. *ctypes*는 구조체나 공용체 필드를 만들 필요 없이, 중첩된 필드에 직접 액세스할 수 있는 디스크립터를 구조체 형에 만듭니다.

다음은 예제 형입니다 (윈도우):

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

`TYPEDESC` 구조체는 COM 데이터형을 설명합니다. `vt` 필드는 공용체 필드 중 어느 것이 유효한지 지정합니다. `u` 필드가 익명 필드로 정의되었으므로, 이제 TYPEDESC 인스턴스에서 멤버에 직접 액세스할 수 있습니다. `td.lptdesc`와 `td.u.lptdesc`는 동등하지만, 앞에 있는 것이 임시 공용체 인스턴스를 만들 필요가 없으므로 더 빠릅니다:

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

구조체 형의 서브-서브 클래스를 정의할 수 있으며, 베이스 클래스의 필드를 상속합니다. 서브 클래스 정의에 별도의 *_fields_* 변수가 있으면, 여기에 지정된 필드가 베이스 클래스의 필드에 추가됩니다.

구조체와 공용체 생성자는 위치와 키워드 인자를 모두 받아들입니다. 위치 인자는 *_fields_*에 나타나는 순서대로 멤버 필드를 초기화하는 데 사용됩니다. 생성자의 키워드 인자는 어트리뷰트 대입으로 해석되므로, *_fields_*를 같은 이름으로 초기화하거나, *_fields_*에 없는 이름에 대한 새 어트리뷰트를 만듭니다.

class ctypes.**CField**(**args*, ***kw*)

Descriptor for fields of a `Structure` and `Union`. For example:

```
>>> class Color(Structure):
...     _fields_ = (
...         ('red', c_uint8),
...         ('green', c_uint8),
...         ('blue', c_uint8),
...         ('intense', c_bool, 1),
...         ('blinking', c_bool, 1),
...     )
...
>>> Color.red
<ctypes.CField 'red' type=c_ubyte, ofs=0, size=1>
>>> Color.green.type
<class 'ctypes.c_ubyte'>
>>> Color.blue.byte_offset
2
>>> Color.intense
<ctypes.CField 'intense' type=c_bool, ofs=3, bit_size=1, bit_offset=0>
>>> Color.blinking.bit_offset
1
```

All attributes are read-only.

`CField` objects are created via `_fields_`; do not instantiate the class directly.

Added in version 3.14: Previously, descriptors only had `offset` and `size` attributes and a readable string representation; the `CField` class was not available directly.

name

Name of the field, as a string.

type

Type of the field, as a *ctypes class*.

offset

byte_offset

Offset of the field, in bytes.

For bitfields, this is the offset of the underlying byte-aligned *storage unit*; see `bit_offset`.

byte_size

Size of the field, in bytes.

For bitfields, this is the size of the underlying *storage unit*. Typically, it has the same size as the bitfield's type.

size

For non-bitfields, equivalent to `byte_size`.

For bitfields, this contains a backwards-compatible bit-packed value that combines `bit_size` and `bit_offset`. Prefer using the explicit attributes instead.

is_bitfield

True if this is a bitfield.

bit_offset

bit_size

The location of a bitfield within its *storage unit*, that is, within `byte_size` bytes of memory starting at `byte_offset`.

To get the field' s value, read the storage unit as an integer, shift left by `bit_offset` and take the `bit_size` least significant bits.

For non-bitfields, `bit_offset` is zero and `bit_size` is equal to `byte_size * 8`.

is_anonymous

True if this field is anonymous, that is, it contains nested sub-fields that should be merged into a containing structure or union.

배열과 포인터

class ctypes.**Array**(**args*)

배열의 추상 베이스 클래스.

The recommended way to create concrete array types is by multiplying any `ctypes` data type with a non-negative integer. Alternatively, you can subclass this type and define `_length_` and `_type_` class variables. Array elements can be read and written using standard subscript and slice accesses; for slice reads, the resulting object is *not* itself an `Array`.

length

배열의 요소 수를 지정하는 양의 정수. 범위를 벗어나는 서브 스크립트는 `IndexError`를 일으킵니다. `len()`에 의해 반환됩니다.

type

배열의 각 요소 형을 지정합니다.

Array 서브 클래스 생성자는 요소를 순서대로 초기화하는 데 사용되는 위치 인자를 받아들입니다.

ctypes.**ARRAY**(*type*, *length*)

Create an array. Equivalent to `type * length`, where *type* is a `ctypes` data type and *length* an integer.

This function is *soft deprecated* in favor of multiplication. There are no plans to remove it.

class ctypes.**_Pointer**

포인터를 위한 내부 추상 베이스 클래스.

구상 포인터형은 가리킬 형으로 `POINTER()`를 호출해서 만들어집니다; 이것은 `pointer()`에 의해 자동으로 수행됩니다.

포인터가 배열을 가리키면, 그것의 요소는 표준 서브 스크립트 및 슬라이스 액세스를 사용하여 읽고 쓸 수 있습니다. 포인터 객체는 크기가 없으므로, `len()`는 `TypeError`를 발생시킵니다. 음수 서브 스크립트는 (C처럼) 포인터 앞의 메모리를 읽을 것이고, 범위를 벗어나는 서브 스크립트는 (운이 좋다면) 액세스 위반으로 인해 충돌을 일으킬 것입니다.

type

가리키는 형을 지정합니다.

contents

포인터가 가리키는 객체를 반환합니다. 이 어트리뷰트에 대입하면 대입된 객체를 가리키도록 포인터가 변경됩니다.

Exceptions

exception ctypes.**ArgumentError**

외부 함수 호출이 전달된 인자 중 하나를 변환할 수 없을 때 발생하는 예외.

exception ctypes.**COMError**(*hresult*, *text*, *details*)

This exception is raised when a COM method call failed.

hresult

The integer value representing the error code.

text

The error message.

details

The 5-tuple `(descr, source, helpfile, helpcontext, progid)`.

descr is the textual description. *source* is the language-dependent `ProgID` for the class or application that raised the error. *helpfile* is the path of the help file. *helpcontext* is the help context identifier. *progid* is the `ProgID` of the interface that defined the error.

가용성: Windows

Added in version 3.14.

CHAPTER 17

Command-line interface libraries

The modules described in this chapter assist with implementing command line and terminal interfaces for applications.

Here' s an overview:

17.1 argparse — Parser for command-line options, arguments and subcommands

Added in version 3.2.

소스 코드: Lib/argparse.py

> **참고**
>
> While *argparse* is the default recommended standard library module for implementing basic command line applications, authors with more exacting requirements for exactly how their command line applications behave may find it doesn' t provide the necessary level of control. Refer to *Choosing an argument parsing library* for alternatives to consider when argparse doesn' t support behaviors that the application requires (such as entirely disabling support for interspersed options and positional arguments, or accepting option parameter values that start with – even when they correspond to another defined option).

> **자습서**
>
> 이 페이지는 API 레퍼런스 정보를 담고 있습니다. 파이썬 명령행 파싱에 대한 더 친절한 소개를 원하시면, *argparse* 자습서 를 보십시오.

The argparse module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and argparse will figure out how to parse those out of *sys.argv*. The argparse module also automatically generates help and usage messages. The module will also issue errors when users give the program invalid arguments.

The argparse module' s support for command-line interfaces is built around an instance of *argparse.ArgumentParser*. It is a container for argument specifications and has options that apply to the parser as whole:

```
parser = argparse.ArgumentParser(
                    prog='ProgramName',
                    description='What the program does',
                    epilog='Text at the bottom of help')
```

The `ArgumentParser.add_argument()` method attaches individual argument specifications to the parser. It supports positional arguments, options that accept values, and on/off flags:

```
parser.add_argument('filename')           # positional argument
parser.add_argument('-c', '--count')      # option that takes a value
parser.add_argument('-v', '--verbose',
                    action='store_true')  # on/off flag
```

The `ArgumentParser.parse_args()` method runs the parser and places the extracted data in a `argparse.Namespace` object:

```
args = parser.parse_args()
print(args.filename, args.count, args.verbose)
```

> **참고**
>
> If you' re looking for a guide about how to upgrade `optparse` code to `argparse`, see *Upgrading Optparse Code*.

17.1.1 ArgumentParser 객체

class argparse.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=argparse.HelpFormatter*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*, *exit_on_error=True*, **, suggest_on_error=False*, *color=True*)

새로운 `ArgumentParser` 객체를 만듭니다. 모든 매개 변수는 키워드 인자로 전달되어야 합니다. 매개 변수마다 아래에서 더 자세히 설명되지만, 요약하면 다음과 같습니다:

- *prog* - The name of the program (default: generated from the `__main__` module attributes and `sys.argv[0]`)
- *usage* - 프로그램 사용법을 설명하는 문자열 (기본값: 파서에 추가된 인자로부터 만들어지는 값)
- *description* - Text to display before the argument help (by default, no text)
- *epilog* - Text to display after the argument help (by default, no text)
- *parents* - `ArgumentParser` 객체들의 리스트이고, 이 들의 인자들도 포함된다
- *formatter_class* - 도움말 출력을 사용자 정의하기 위한 클래스
- *prefix_chars* - 선택 인자 앞에 붙는 문자 집합 (기본값: '-').
- *fromfile_prefix_chars* - 추가 인자를 읽어야 하는 파일 앞에 붙는 문자 집합 (기본값: `None`).
- *argument_default* - 인자의 전역 기본값 (기본값: `None`)
- *conflict_handler* - 충돌하는 선택 사항을 해결하기 위한 전략 (일반적으로 불필요함)
- *add_help* - 파서에 `-h/--help` 옵션을 추가합니다 (기본값: `True`)
- *allow_abbrev* - Allows long options to be abbreviated if the abbreviation is unambiguous (default: `True`)
- *exit_on_error* - Determines whether or not `ArgumentParser` exits with error info when an error occurs. (default: `True`)
- *suggest_on_error* - Enables suggestions for mistyped argument choices and subparser names (default: `False`)

- *color* - Allow color output (default: `True`)

버전 3.5에서 변경: *allow_abbrev* 매개 변수가 추가되었습니다.

버전 3.8에서 변경: 이전 버전에서는, *allow_abbrev*는 `-vv`가 `-v -v`를 뜻하는 것과 같은 짧은 플래그의 그룹화도 비활성화했습니다.

버전 3.9에서 변경: *exit_on_error* 매개 변수가 추가되었습니다.

버전 3.14에서 변경: *suggest_on_error* and *color* parameters were added.

다음 절에서는 이들 각각의 사용 방법에 관해 설명합니다.

prog

By default, `ArgumentParser` calculates the name of the program to display in help messages depending on the way the Python interpreter was run:

- The `base name` of `sys.argv[0]` if a file was passed as argument.
- The Python interpreter name followed by `sys.argv[0]` if a directory or a zipfile was passed as argument.
- The Python interpreter name followed by `-m` followed by the module or package name if the `-m` option was used.

This default is almost always desirable because it will make the help messages match the string that was used to invoke the program on the command line. However, to change this default behavior, another value can be supplied using the `prog=` argument to `ArgumentParser`:

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

options:
 -h, --help  show this help message and exit
```

Note that the program name, whether determined from `sys.argv[0]`, from the `__main__` module attributes or from the `prog=` argument, is available to help messages using the `%(prog)s` format specifier.

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

options:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

버전 3.14에서 변경: The default `prog` value now reflects how `__main__` was actually executed, rather than always being `os.path.basename(sys.argv[0])`.

usage

By default, `ArgumentParser` calculates the usage message from the arguments it contains. The default message can be overridden with the `usage=` keyword argument:

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
 bar           bar help

options:
 -h, --help    show this help message and exit
 --foo [FOO]   foo help
```

`%(prog)s` 포맷 지정자는 사용법 메시지에서 프로그램 이름을 채울 때 사용할 수 있습니다.

When a custom usage message is specified for the main parser, you may also want to consider passing the `prog` argument to *add_subparsers()* or the `prog` and the `usage` arguments to `add_parser()`, to ensure consistent command prefixes and usage information across subparsers.

description

Most calls to the *ArgumentParser* constructor will use the `description=` keyword argument. This argument gives a brief description of what the program does and how it works. In help messages, the description is displayed between the command-line usage string and the help messages for the various arguments.

기본적으로, 설명은 주어진 공간에 맞도록 줄 바꿈 됩니다. 이 동작을 변경하려면 *formatter_class* 인자를 참조하십시오.

epilog

일부 프로그램은 인자에 대한 설명 뒤에 프로그램에 대한 추가 설명을 표시하려고 합니다. 이러한 텍스트는 `epilog=` 에 대한 인자를 *ArgumentParser* 에 사용하여 지정할 수 있습니다:

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

options:
 -h, --help  show this help message and exit

And that's how you'd foo a bar
```

description 인자와 마찬가지로, `epilog=` 텍스트가 기본적으로 줄 바꿈 됩니다만, 이 동작은 *formatter_class* 인자를 *ArgumentParser* 에 제공해서 조정할 수 있습니다.

parents

때로는 여러 파서가 공통 인자 집합을 공유하는 경우가 있습니다. 이러한 인자의 정의를 반복하는 대신, 모든 공유 인자를 갖는 파서를 *ArgumentParser* 에 `parents=` 인자로 전달할 수 있습니다. `parents=` 인자는 *ArgumentParser* 객체의 리스트를 취하여, 그것들로부터 모든 위치와 선택 액션을 수집해서 새로 만들어지는 *ArgumentParser* 객체에 추가합니다:

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

대부분의 부모 파서는 `add_help=False` 를 지정합니다. 그렇지 않으면, *ArgumentParser* 는 (하나는 부모에, 하나는 자식에 있는) 두 개의 `-h/--help` 옵션을 보게 될 것이고, 에러를 발생시킵니다.

> **참고**
>
> `parents=` 를 통해 전달하기 전에 파서를 완전히 초기화해야 합니다. 자식 파서 다음에 부모 파서를 변경하면 자식에 반영되지 않습니다.

formatter_class

ArgumentParser 객체는 대체 포매팅 클래스를 지정함으로써 도움말 포매팅을 사용자 정의 할 수 있도록 합니다. 현재 네 가지 클래스가 있습니다:

class argparse.**RawDescriptionHelpFormatter**

class argparse.**RawTextHelpFormatter**

class argparse.**ArgumentDefaultsHelpFormatter**

class argparse.**MetavarTypeHelpFormatter**

RawDescriptionHelpFormatter 와 *RawTextHelpFormatter* 는 텍스트 설명이 표시되는 방법을 더 제어할 수 있도록 합니다. 기본적으로, *ArgumentParser* 객체는 명령행 도움말 메시지에서 *description* 및 *epilog* 텍스트를 줄 바꿈 합니다.:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...             but that is okay''',
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay

options:
 -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

RawDescriptionHelpFormatter 를 `formatter_class=` 로 전달하는 것은 *description* 과 *epilog* 가 이미 올바르게 포맷되어 있어서 줄 바꿈 되어서는 안 된다는 것을 가리킵니다:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
   I have indented it
   exactly the way
   I want it

options:
 -h, --help  show this help message and exit
```

RawTextHelpFormatter maintains whitespace for all sorts of help text, including argument descriptions. However, multiple newlines are replaced with one. If you wish to preserve multiple blank lines, add spaces between the newlines.

ArgumentDefaultsHelpFormatter 는 기본값에 대한 정보를 각각의 인자 도움말 메시지에 자동으로 추가합니다:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar ...]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

options:
 -h, --help  show this help message and exit
 --foo FOO   FOO! (default: 42)
```

MetavarTypeHelpFormatter 는 각 인자 값의 표시 이름으로 (일반 포매터처럼 *dest* 를 사용하는 대신에) *type* 인자의 이름을 사용합니다:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

options:
  -h, --help  show this help message and exit
  --foo int
```

prefix_chars

Most command-line options will use - as the prefix, e.g. `-f/--foo`. Parsers that need to support different or additional prefix characters, e.g. for options like `+f` or `/foo`, may specify them using the `prefix_chars=` argument to the *ArgumentParser* constructor:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

`prefix_chars=` 인자의 기본값은 `'-'` 입니다. - 를 포함하지 않는 문자 집합을 제공하면 `-f/--foo` 옵션이 허용되지 않게 됩니다.

fromfile_prefix_chars

Sometimes, when dealing with a particularly long argument list, it may make sense to keep the list of arguments in a file rather than typing it out at the command line. If the `fromfile_prefix_chars=` argument is given to the *ArgumentParser* constructor, then arguments that start with any of the specified characters will be treated as files, and will be replaced by the arguments they contain. For example:

```
>>> with open('args.txt', 'w', encoding=sys.getfilesystemencoding()) as fp:
...     fp.write('-f\nbar')
...
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

Arguments read from a file must be one per line by default (but see also *convert_arg_line_to_args()*) and are treated as if they were in the same place as the original file referencing argument on the command line. So in the example above, the expression `['-f', 'foo', '@args.txt']` is considered equivalent to the expression `['-f', 'foo', '-f', 'bar']`.

> **참고**
>
> Empty lines are treated as empty strings (`''`), which are allowed as values but not as arguments. Empty lines that are read as arguments will result in an "unrecognized arguments" error.

ArgumentParser uses *filesystem encoding and error handler* to read the file containing arguments.

`fromfile_prefix_chars=` 인자의 기본값은 `None` 입니다. 이것은 인자가 절대로 파일 참조로 취급되지 않는다는 것을 의미합니다.

버전 3.12에서 변경: *ArgumentParser* changed encoding and errors to read arguments files from default (e.g. *locale.getpreferredencoding(False)* and `"strict"`) to the *filesystem encoding and error handler*. Arguments file should be encoded in UTF-8 instead of ANSI Codepage on Windows.

argument_default

일반적으로 인자의 기본값은 *add_argument()* 에 기본값을 전달하거나 특정 이름-값 쌍 집합을 사용하여 *set_defaults()* 메서드를 호출하여 지정됩니다. 그러나 때로는, 파서 전체에 적용되는 단일 기본값을 지정하는 것이 유용 할 수 있습니다. 이것은 `argument_default=` 키워드 인자를 *ArgumentParser* 에 전달함으로써 이루어질 수 있습니다. 예를 들어, *parse_args()* 호출에서 어트리뷰트 생성을 전역적으로 억제하려면, `argument_default=SUPPRESS` 를 제공합니다:

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

allow_abbrev

일반적으로 *ArgumentParser* 의 *parse_args()* 메서드에 인자 리스트를 건네주면 긴 옵션의 약어를 인식합니다.

`allow_abbrev` 를 `False` 로 설정하면 이 기능을 비활성화 할 수 있습니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

Added in version 3.5.

conflict_handler

ArgumentParser 객체는 같은 옵션 문자열을 가진 두 개의 액션을 허용하지 않습니다. 기본적으로 *ArgumentParser* 객체는 이미 사용 중인 옵션 문자열로 인자를 만들려고 시도하면 예외를 발생시킵니다

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
 ..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

때로는 (예를 들어 *parents* 를 사용하는 경우) 같은 옵션 문자열을 갖는 예전의 인자들을 간단히 대체하는 것이 유용 할 수 있습니다. 이 동작을 얻으려면, *ArgumentParser* 의 `conflict_handler=` 인자에 `'resolve'` 값을 제공합니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

options:
 -h, --help  show this help message and exit
 -f FOO      old foo help
 --foo FOO   new foo help
```

ArgumentParser 객체는 모든 옵션 문자열이 재정의된 경우에만 액션을 제거합니다. 위의 예에서, 이전의 `-f/--foo` 액션은 `--foo` 옵션 문자열만 재정의되었기 때문에 `-f` 액션으로 유지됩니다.

add_help

By default, *ArgumentParser* objects add an option which simply displays the parser' s help message. If `-h` or `--help` is supplied at the command line, the `ArgumentParser` help will be printed.

때에 따라, 이 도움말 옵션을 추가하지 않도록 설정하는 것이 유용 할 수 있습니다. `add_help=` 인자를 `False` 로 *ArgumentParser* 에 전달하면 됩니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

options:
 --foo FOO  foo help
```

도움말 옵션은 일반적으로 `-h/--help` 입니다. 예외는 `prefix_chars=` 가 지정되고 `-` 을 포함하지 않는 경우입니다. 이 경우 `-h` 와 `--help` 는 유효한 옵션이 아닙니다. 이 경우, `prefix_chars` 의 첫 번째 문자가 도움말 옵션 접두어로 사용됩니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

options:
  +h, ++help  show this help message and exit
```

exit_on_error

Normally, when you pass an invalid argument list to the *parse_args()* method of an *ArgumentParser*, it will print a *message* to *sys.stderr* and exit with a status code of 2.

If the user would like to catch errors manually, the feature can be enabled by setting `exit_on_error` to `False`:

```
>>> parser = argparse.ArgumentParser(exit_on_error=False)
>>> parser.add_argument('--integers', type=int)
_StoreAction(option_strings=['--integers'], dest='integers', nargs=None,␣
↪const=None, default=None, type=<class 'int'>, choices=None, help=None,␣
↪metavar=None)
>>> try:
...     parser.parse_args('--integers a'.split())
... except argparse.ArgumentError:
...     print('Catching an argumentError')
...
Catching an argumentError
```

Added in version 3.9.

suggest_on_error

By default, when a user passes an invalid argument choice or subparser name, *ArgumentParser* will exit with error info and list the permissible argument choices (if specified) or subparser names as part of the error message.

If the user would like to enable suggestions for mistyped argument choices and subparser names, the feature can be enabled by setting `suggest_on_error` to `True`. Note that this only applies for arguments when the choices specified are strings:

```
>>> parser = argparse.ArgumentParser(description='Process some integers.',
                                     suggest_on_error=True)
>>> parser.add_argument('--action', choices=['sum', 'max'])
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.parse_args(['--action', 'sumn', 1, 2, 3])
tester.py: error: argument --action: invalid choice: 'sumn', maybe you meant 'sum'?
↪ (choose from 'sum', 'max')
```

If you' re writing code that needs to be compatible with older Python versions and want to opportunistically use `suggest_on_error` when it' s available, you can set it as an attribute after initializing the parser instead of using the keyword argument:

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
>>> parser.suggest_on_error = True
```

Added in version 3.14.

color

By default, the help message is printed in color using ANSI escape sequences. If you want plain text help messages, you can disable this in your local environment, or in the argument parser itself by setting `color` to `False`:

```
>>> parser = argparse.ArgumentParser(description='Process some integers.',
...                                  color=False)
>>> parser.add_argument('--action', choices=['sum', 'max'])
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.parse_args(['--help'])
```

Added in version 3.14.

17.1.2 add_argument() 메서드

`ArgumentParser.`**`add_argument`**(*name or flags...*, *[, *action*] [, *nargs*] [, *const*] [, *default*] [, *type*] [, *choices*] [, *required*] [, *help*] [, *metavar*] [, *dest*] [, *deprecated*])

단일 명령행 인자를 구문 분석하는 방법을 정의합니다. 매개 변수마다 아래에서 더 자세히 설명되지만, 요약하면 다음과 같습니다:

- *name or flags* - Either a name or a list of option strings, e.g. `'foo'` or `'-f', '--foo'`.
- *action* - 명령행에서 이 인자가 발견될 때 수행 할 액션의 기본형.
- *nargs* - 소비되어야 하는 명령행 인자의 수.
- *const* - 일부 *action* 및 *nargs* 를 선택할 때 필요한 상숫값.
- *default* - 인자가 명령행에 없고 namespace 객체에 없으면 생성되는 값.
- *type* - 명령행 인자가 변환되어야 할 형.
- *choices* - A sequence of the allowable values for the argument.
- *required* - 명령행 옵션을 생략 할 수 있는지 아닌지 (선택적일 때만).
- *help* - 인자가 하는 일에 대한 간단한 설명.
- *metavar* - 사용 메시지에 사용되는 인자의 이름.
- *dest* - `parse_args()` 가 반환하는 객체에 추가될 어트리뷰트의 이름.
- *deprecated* - Whether or not use of the argument is deprecated.

다음 절에서는 이들 각각의 사용 방법에 관해 설명합니다.

name or flags

The `add_argument()` method must know whether an optional argument, like `-f` or `--foo`, or a positional argument, like a list of filenames, is expected. The first arguments passed to `add_argument()` must therefore be either a series of flags, or a simple argument name.

For example, an optional argument could be created like:

```
>>> parser.add_argument('-f', '--foo')
```

반면에 위치 인자는 이렇게 만들어집니다:

```
>>> parser.add_argument('bar')
```

parse_args() 가 호출되면, 선택 인자는 - 접두사로 식별되고, 그 밖의 인자는 위치 인자로 간주합니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: the following arguments are required: bar
```

By default, `argparse` automatically handles the internal naming and display names of arguments, simplifying the process without requiring additional configuration. As such, you do not need to specify the *dest* and *metavar* parameters. The *dest* parameter defaults to the argument name with underscores _ replacing hyphens - . The *metavar* parameter defaults to the upper-cased name. For example:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo-bar')
>>> parser.parse_args(['--foo-bar', 'FOO-BAR']
Namespace(foo_bar='FOO-BAR')
>>> parser.print_help()
usage:  [-h] [--foo-bar FOO-BAR]

optional arguments:
 -h, --help  show this help message and exit
 --foo-bar FOO-BAR
```

action

ArgumentParser 객체는 명령행 인자를 액션과 연관시킵니다. 대부분의 액션은 단순히 *parse_args()* 에 의해 반환된 객체에 어트리뷰트를 추가하기만 하지만, 액션은 관련된 명령행 인자로 무엇이든 할 수 있습니다. `action` 키워드 인자는 명령행 인자의 처리 방법을 지정합니다. 제공되는 액션은 다음과 같습니다:

- `'store'` - This just stores the argument' s value. This is the default action.
- `'store_const'` - This stores the value specified by the *const* keyword argument; note that the *const* keyword argument defaults to `None`. The `'store_const'` action is most commonly used with optional arguments that specify some sort of flag. For example:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_const', const=42)
  >>> parser.parse_args(['--foo'])
  Namespace(foo=42)
  ```

- `'store_true'` and `'store_false'` - These are special cases of `'store_const'` used for storing the values `True` and `False` respectively. In addition, they create default values of `False` and `True` respectively:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_true')
  >>> parser.add_argument('--bar', action='store_false')
  >>> parser.add_argument('--baz', action='store_false')
  ```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> parser.parse_args('--foo --bar'.split())
Namespace(foo=True, bar=False, baz=True)
```

- `'append'` - This appends each argument value to a list. It is useful for allowing an option to be specified multiple times. If the default value is a non-empty list, the parsed value will start with the default list' s elements and any values from the command line will be appended after those default values. Example usage:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append', default=['0'])
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['0', '1', '2'])
```

- `'append_const'` - This stores a list, and appends the value specified by the *const* keyword argument to the list; note that the *const* keyword argument defaults to `None`. The `'append_const'` action is typically useful when multiple arguments need to store constants to the same list. For example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const',␣
↪const=str)
>>> parser.add_argument('--int', dest='types', action='append_const',␣
↪const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

- `'extend'` - This stores a list and appends each item from the multi-value argument list to it. The `'extend'` action is typically used with the *nargs* keyword argument value `'+'` or `'*'`. Note that when *nargs* is `None` (the default) or `'?'`, each character of the argument string will be appended to the list. Example usage:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument("--foo", action="extend", nargs="+", type=str)
>>> parser.parse_args(["--foo", "f1", "--foo", "f2", "f3", "f4"])
Namespace(foo=['f1', 'f2', 'f3', 'f4'])
```

  Added in version 3.8.

- `'count'` - 키워드 인자가 등장한 횟수를 계산합니다. 예를 들어, 상세도를 높이는 데 유용합니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count', default=0)
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

  참고, 명시적으로 *0*으로 설정되지 않으면 *default*는 `None`이 됩니다.

- `'help'` - 현재 파서의 모든 옵션에 대한 완전한 도움말 메시지를 출력하고 종료합니다. 기본적으로 help 액션은 자동으로 파서에 추가됩니다. 출력이 만들어지는 방법에 대한 자세한 내용은 `ArgumentParser`를 보세요.

- `'version'` - `add_argument()` 호출에서 `version=` 키워드 인자를 기대하고, 호출되면 버전 정보를 출력하고 종료합니다:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

You may also specify an arbitrary action by passing an `Action` subclass (e.g. `BooleanOptionalAction`) or other object that implements the same interface. Only actions that consume command-line arguments (e.g. `'store'`, `'append'`, `'extend'`, or custom actions with non-zero `nargs`) can be used with positional arguments.

The recommended way to create a custom action is to extend *Action*, overriding the `__call__()` method and optionally the `__init__()` and `format_usage()` methods. You can also register custom actions using the *register()* method and reference them by their registered name.

사용자 정의 액션의 예:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super().__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print('%r %r %r' % (namespace, values, option_string))
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

자세한 내용은 *Action* 을 참조하십시오.

nargs

ArgumentParser objects usually associate a single command-line argument with a single action to be taken. The `nargs` keyword argument associates a different number of command-line arguments with a single action. See also *Specifying ambiguous arguments*. The supported values are:

- `N` (정수). 명령행에서 `N` 개의 인자를 함께 모아서 리스트에 넣습니다. 예를 들면:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs=2)
  >>> parser.add_argument('bar', nargs=1)
  >>> parser.parse_args('c --foo a b'.split())
  Namespace(bar=['c'], foo=['a', 'b'])
  ```

  `nargs=1` 은 하나의 항목을 갖는 리스트를 생성합니다. 이는 항목 그대로 생성되는 기본값과 다릅니다.

- `'?'`. 가능하다면 한 인자가 명령행에서 소비되고 단일 항목으로 생성됩니다. 명령행 인자가 없으면 *default* 의 값이 생성됩니다. 선택 인자의 경우 추가적인 경우가 있습니다 - 옵션 문자열은 있지만, 명령행 인자가 따라붙지 않는 경우입니다. 이 경우 *const* 의 값이 생성됩니다. 이것을 보여주기 위해 몇 가지 예를 듭니다:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs='?', const='c', default='d')
  >>> parser.add_argument('bar', nargs='?', default='d')
  >>> parser.parse_args(['XX', '--foo', 'YY'])
  Namespace(bar='XX', foo='YY')
  >>> parser.parse_args(['XX', '--foo'])
  Namespace(bar='XX', foo='c')
  >>> parser.parse_args([])
  Namespace(bar='d', foo='d')
  ```

  `nargs='?'` 의 흔한 사용법 중 하나는 선택적 입출력 파일을 허용하는 것입니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', nargs='?')
>>> parser.add_argument('outfile', nargs='?')
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile='input.txt', outfile='output.txt')
>>> parser.parse_args(['input.txt'])
Namespace(infile='input.txt', outfile=None)
>>> parser.parse_args([])
Namespace(infile=None, outfile=None)
```

- `'*'`. 모든 명령행 인자를 리스트로 수집합니다. 일반적으로 두 개 이상의 위치 인자에 대해 `nargs='*'` 를 사용하는 것은 별로 의미가 없지만, `nargs='*'` 를 갖는 여러 개의 선택 인자는 가능합니다. 예를 들면:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- `'+'`. Just like `'*'`, all command-line arguments present are gathered into a list. Additionally, an error message will be generated if there wasn' t at least one command-line argument present. For example:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: the following arguments are required: foo
```

If the `nargs` keyword argument is not provided, the number of arguments consumed is determined by the *action*. Generally this means a single command-line argument will be consumed and a single item (not a list) will be produced. Actions that do not consume command-line arguments (e.g. `'store_const'`) set `nargs=0`.

const

`add_argument()` 의 `const` 인자는 명령행에서 읽지는 않지만 다양한 *`ArgumentParser`* 액션에 필요한 상숫값을 저장하는 데 사용됩니다. 가장 흔한 두 가지 용도는 다음과 같습니다:

- When *`add_argument()`* is called with `action='store_const'` or `action='append_const'`. These actions add the `const` value to one of the attributes of the object returned by *`parse_args()`*. See the *action* description for examples. If `const` is not provided to *`add_argument()`*, it will receive a default value of `None`.
- When *`add_argument()`* is called with option strings (like `-f` or `--foo`) and `nargs='?'`. This creates an optional argument that can be followed by zero or one command-line arguments. When parsing the command line, if the option string is encountered with no command-line argument following it, the value from `const` will be used. See the *nargs* description for examples.

버전 3.11 에서 변경: `const=None` by default, including when `action='append_const'` or `action='store_const'`.

default

모든 선택 인자와 일부 위치 인자는 명령행에서 생략될 수 있습니다. *`add_argument()`* 의 `default` 키워드 인자는, 기본값은 `None` 입니다, 명령행 인자가 없을 때 어떤 값을 사용해야 하는지를 지정합니다. 선택 인자의 경우, 옵션 문자열이 명령행에 없을 때 `default` 값이 사용됩니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

If the target namespace already has an attribute set, the action *default* will not overwrite it:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args([], namespace=argparse.Namespace(foo=101))
Namespace(foo=101)
```

`default` 값이 문자열이면, 파서는 마치 명령행 인자인 것처럼 파싱합니다. 특히, 파서는 *Namespace* 반환 값에 어트리뷰트를 설정하기 전에, 제공된 모든 *type* 변환 인자를 적용합니다. 그렇지 않은 경우, 파서는 값을 있는 그대로 사용합니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

nargs 가 `?` 또는 `*` 인 위치 인자의 경우, 명령행 인자가 없을 때 `default` 값이 사용됩니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

For *required* arguments, the `default` value is ignored. For example, this applies to positional arguments with *nargs* values other than `?` or `*`, or optional arguments marked as `required=True`.

`default=argparse.SUPPRESS` 를 지정하면 명령행 인자가 없는 경우 어트리뷰트가 추가되지 않습니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

type

기본적으로, 파서는 명령행 인자를 간단한 문자열로 읽습니다. 그러나 꽤 자주 명령행 문자열은 *float* 나 *int* 와 같은 다른 형으로 해석되어야 합니다. *add_argument()* 의 `type` 키워드는 필요한 형 검사와 형 변환이 수행되도록 합니다.

type 키워드가 *default* 키워드와 함께 사용되면, 형 변환기는 기본값이 문자열일 때만 적용됩니다.

The argument to `type` can be a callable that accepts a single string or the name of a registered type (see *register()*) If the function raises *ArgumentTypeError*, *TypeError*, or *ValueError*, the exception is caught and a nicely formatted error message is displayed. Other exception types are not handled.

일반적인 내장형과 함수는 형 변환기로 사용될 수 있습니다:

```
import argparse
import pathlib

parser = argparse.ArgumentParser()
parser.add_argument('count', type=int)
parser.add_argument('distance', type=float)
parser.add_argument('street', type=ascii)
parser.add_argument('code_point', type=ord)
parser.add_argument('datapath', type=pathlib.Path)
```

사용자 정의 함수도 사용할 수 있습니다:

```
>>> def hyphenated(string):
...     return '-'.join([word[:4] for word in string.casefold().split()])
...
>>> parser = argparse.ArgumentParser()
>>> _ = parser.add_argument('short_title', type=hyphenated)
>>> parser.parse_args(['"The Tale of Two Cities"'])
Namespace(short_title='"the-tale-of-two-citi')
```

bool() 함수는 형 변환기로 권장되지 않습니다. 빈 문자열을 `False`로, 비어 있지 않은 문자열을 `True`로 변환하는 것뿐입니다. 이것은 일반적으로 원하는 것이 아닙니다.

일반적으로, `type` 키워드는 지원되는 세 가지 예외 중 하나만 발생할 수 있는 간단한 변환에만 사용해야 하는 편의 기능입니다. 더 흥미로운 에러 처리나 리소스 관리가 필요한 모든 작업은 인자가 구문 분석된 후에 수행되어야 합니다.

For example, JSON or YAML conversions have complex error cases that require better reporting than can be given by the `type` keyword. A *JSONDecodeError* would not be well formatted and a *FileNotFoundError* exception would not be handled at all.

Even *FileType* has its limitations for use with the `type` keyword. If one argument uses *FileType* and then a subsequent argument fails, an error is reported but the file is not automatically closed. In this case, it would be better to wait until after the parser has run and then use the `with`-statement to manage the files.

고정된 값 집합에 대해 단순히 확인하는 형 검사기의 경우, 대신 *choices* 키워드를 사용하는 것을 고려하십시오.

choices

Some command-line arguments should be selected from a restricted set of values. These can be handled by passing a sequence object as the *choices* keyword argument to *add_argument()*. When the command line is parsed, argument values will be checked, and an error message will be displayed if the argument was not one of the acceptable values:

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

Any sequence can be passed as the *choices* value, so *list* objects, *tuple* objects, and custom sequences are all supported.

*enum.Enum*은 사용법, 도움말 및 에러 메시지에서 나타나는 방식을 제어하기 어렵기 때문에 사용하지 않는 것이 좋습니다.

Note that *choices* are checked after any *type* conversions have been performed, so objects in *choices* should match the *type* specified. This can make *choices* appear unfamiliar in usage, help, or error messages.

To keep *choices* user-friendly, consider a custom type wrapper that converts and formats values, or omit *type* and handle conversion in your application code.

Formatted choices override the default *metavar* which is normally derived from *dest*. This is usually what you want because the user never sees the *dest* parameter. If this display isn' t desirable (perhaps because there are many choices), just specify an explicit *metavar*.

required

In general, the `argparse` module assumes that flags like `-f` and `--bar` indicate *optional* arguments, which can always be omitted at the command line. To make an option *required*, `True` can be specified for the `required=` keyword argument to *add_argument()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: [-h] --foo FOO
: error: the following arguments are required: --foo
```

예에서 보듯이, 옵션이 `required` 로 표시되면, *parse_args()* 는 그 옵션이 명령행에 없을 때 에러를 보고합니다.

> **참고**
>
> 필수 옵션은 사용자가 옵션 이 선택적 일 것으로 기대하기 때문에 일반적으로 나쁜 형식으로 간주하므로 가능하면 피해야 합니다.

help

The `help` value is a string containing a brief description of the argument. When a user requests help (usually by using `-h` or `--help` at the command line), these `help` descriptions will be displayed with each argument.

`help` 문자열은 프로그램 이름이나 인자 *default* 와 같은 것들의 반복을 피하고자 다양한 포맷 지정자를 포함 할 수 있습니다. 사용할 수 있는 지정자는 프로그램 이름, `%(prog)s` 와 *add_argument()* 의 대부분의 키워드 인자, `%(default)s`, `%(type)s` 등을 포함합니다.:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
 bar     the bar to frobble (default: 42)

options:
 -h, --help  show this help message and exit
```

도움말 문자열이 %-포매팅을 지원하기 때문에, 도움말 문자열에 리터럴 `%` 을 표시하려면, `%%` 로 이스케이프 처리해야 합니다.

`argparse` supports silencing the help entry for certain options, by setting the `help` value to `argparse.SUPPRESS`:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
options:
  -h, --help  show this help message and exit
```

metavar

When *ArgumentParser* generates help messages, it needs some way to refer to each expected argument. By default, ArgumentParser objects use the *dest* value as the "name" of each object. By default, for positional argument actions, the *dest* value is used directly, and for optional argument actions, the *dest* value is uppercased. So, a single positional argument with dest='bar' will be referred to as bar. A single optional argument --foo that should be followed by a single command-line argument will be referred to as FOO. An example:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar

positional arguments:
 bar

options:
 -h, --help  show this help message and exit
 --foo FOO
```

다른 이름은 metavar 로 지정할 수 있습니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
 XXX

options:
 -h, --help  show this help message and exit
 --foo YYY
```

metavar 는 표시되는 이름 만 변경합니다 - *parse_args()* 객체의 어트리뷰트 이름은 여전히 *dest* 값에 의해 결정됩니다.

nargs 값이 다르면 metavar 가 여러 번 사용될 수 있습니다. metavar 에 튜플을 제공하면 인자마다 다른 디스플레이가 지정됩니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

options:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
 -h, --help      show this help message and exit
 -x X X
 --foo bar baz
```

dest

대부분 *ArgumentParser* 액션은 *parse_args()* 에 의해 반환된 객체의 어트리뷰트로 어떤 값을 추가합니다. 이 어트리뷰트의 이름은 *add_argument()* 의 `dest` 키워드 인자에 의해 결정됩니다. 위치 인자 액션의 경우, `dest` 는 일반적으로 *add_argument()* 에 첫 번째 인자로 제공됩니다.:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

선택 인자 액션의 경우, `dest` 의 값은 보통 옵션 문자열에서 유추됩니다. *ArgumentParser* 는 첫 번째 긴 옵션 문자열을 취하고 앞의 `--` 문자열을 제거하여 `dest` 의 값을 만듭니다. 긴 옵션 문자열이 제공되지 않았다면 `dest` 는 첫 번째 짧은 옵션 문자열에서 앞의 `-` 문자를 제거하여 만듭니다. 문자열이 항상 유효한 어트리뷰트 이름이 되도록 만들기 위해 중간에 나오는 `-` 문자는 `_` 문자로 변환됩니다. 아래 예제는 이 동작을 보여줍니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

`dest` 는 사용자 정의 어트리뷰트 이름을 지정할 수 있게 합니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

deprecated

During a project' s lifetime, some arguments may need to be removed from the command line. Before removing them, you should inform your users that the arguments are deprecated and will be removed. The `deprecated` keyword argument of *add_argument()*, which defaults to `False`, specifies if the argument is deprecated and will be removed in the future. For arguments, if `deprecated` is `True`, then a warning will be printed to *sys.stderr* when the argument is used:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='snake.py')
>>> parser.add_argument('--legs', default=0, type=int, deprecated=True)
>>> parser.parse_args([])
Namespace(legs=0)
>>> parser.parse_args(['--legs', '4'])
snake.py: warning: option '--legs' is deprecated
Namespace(legs=4)
```

Added in version 3.13.

Action 클래스

`Action` classes implement the Action API, a callable which returns a callable which processes arguments from the command-line. Any object which follows this API may be passed as the `action` parameter to `add_argument()`.

class argparse.**Action**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

`Action` objects are used by an `ArgumentParser` to represent the information needed to parse a single argument from one or more strings from the command line. The `Action` class must accept the two positional arguments plus any keyword arguments passed to `ArgumentParser.add_argument()` except for the `action` itself.

Instances of `Action` (or return value of any callable to the `action` parameter) should have attributes `dest`, `option_strings`, `default`, `type`, `required`, `help`, etc. defined. The easiest way to ensure these attributes are defined is to call `Action.__init__()`.

__call__(*parser*, *namespace*, *values*, *option_string=None*)

`Action` instances should be callable, so subclasses must override the `__call__()` method, which should accept four parameters:

- *parser* - The `ArgumentParser` object which contains this action.
- *namespace* - The `Namespace` object that will be returned by `parse_args()`. Most actions add an attribute to this object using `setattr()`.
- *values* - The associated command-line arguments, with any type conversions applied. Type conversions are specified with the *type* keyword argument to `add_argument()`.
- *option_string* - The option string that was used to invoke this action. The `option_string` argument is optional, and will be absent if the action is associated with a positional argument.

The `__call__()` method may perform arbitrary actions, but will typically set attributes on the `namespace` based on `dest` and `values`.

format_usage()

`Action` subclasses can define a `format_usage()` method that takes no argument and return a string which will be used when printing the usage of the program. If such method is not provided, a sensible default will be used.

class argparse.**BooleanOptionalAction**

A subclass of `Action` for handling boolean flags with positive and negative options. Adding a single argument such as `--foo` automatically creates both `--foo` and `--no-foo` options, storing `True` and `False` respectively:

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=argparse.BooleanOptionalAction)
>>> parser.parse_args(['--no-foo'])
Namespace(foo=False)
```

Added in version 3.9.

17.1.3 parse_args() 메서드

ArgumentParser.**parse_args**(*args=None*, *namespace=None*)

인자 문자열을 객체로 변환하고 namespace의 어트리뷰트로 설정합니다. 값들이 설정된 namespace를 돌려줍니다.

Previous calls to `add_argument()` determine exactly what objects are created and how they are assigned. See the documentation for `add_argument()` for details.

- *args* - 구문 분석할 문자열 리스트. 기본값은 `sys.argv` 에서 취합니다.
- *namespace* - 어트리뷰트가 대입될 객체. 기본값은 새로 만들어지는 빈 `Namespace` 객체입니다.

옵션값 문법

parse_args() 메서드는 (취할 것이 있다면) 옵션의 값을 지정하는 몇 가지 방법을 지원합니다. 가장 단순한 경우, 옵션과 그 값은 두 개의 독립적인 인자로 전달됩니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

긴 옵션(단일 문자보다 긴 이름을 가진 옵션)의 경우, 옵션과 값을 = 로 구분하여 단일 명령행 인자로 전달할 수도 있습니다:

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

짧은 옵션(한 문자 길이의 옵션)의 경우, 옵션과 해당 값을 이어붙일 수 있습니다:

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

여러 개의 짧은 옵션은 마지막 옵션만 값을 요구하는 한 (또는 그들 중 아무것도 값을 요구하지 않거나), 하나의 - 접두어를 사용하여 함께 결합 할 수 있습니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

잘못된 인자

명령행을 파싱할 때, *parse_args()* 는 모호한 옵션, 유효하지 않은 형, 유효하지 않은 옵션, 잘못된 위치 인자의 수 등을 포함한 다양한 에러를 검사합니다. 이런 에러가 발생하면, 사용 메시지와 함께 에러를 인쇄합니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

- 를 포함하는 인자들

parse_args() 메서드는 사용자가 분명히 실수했을 때마다 에러를 주려고 하지만, 어떤 상황은 본질에서 모호합니다. 예를 들어, 명령행 인자 `-1` 은 옵션을 지정하려는 시도이거나 위치 인자를 제공하려는 시도일 수 있습니다. *parse_args()* 메서드는 이럴 때 신중합니다: 위치 인자는 음수처럼 보이고 파서에 음수처럼 보이는 옵션이 없을 때만 - 로 시작할 수 있습니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

- 로 시작해야 하고, 음수처럼 보이지 않는 위치 인자가 있는 경우, *parse_args()* 에 다음과 같은 의사 인자 `'--'` 를 삽입 할 수 있습니다. 그 이후의 모든 것은 위치 인자입니다:

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

See also *the argparse howto on ambiguous arguments* for more details.

인자 약어 (접두사 일치)

parse_args() 메서드는 기본적으로 약어가 모호하지 않으면 (접두사가 오직 하나의 옵션과 일치합니다) 긴 옵션을 접두사로 축약 할 수 있도록 합니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

둘 이상의 옵션과 일치하는 인자는 에러를 일으킵니다. 이 기능은 *allow_abbrev*를 `False` 로 설정함으로써 비활성화시킬 수 있습니다.

`sys.argv` 너머

Sometimes it may be useful to have an `ArgumentParser` parse arguments other than those of `sys.argv`. This can be accomplished by passing a list of strings to `parse_args()`. This is useful for testing at the interactive prompt:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

Namespace 객체

class argparse.**Namespace**

`parse_args()` 가 어트리뷰트를 저장하고 반환할 객체를 만드는 데 기본적으로 사용하는 간단한 클래스.

이 클래스는 의도적으로 단순한데, 단지 가독성 있는 문자열 표현을 갖는 `object` 의 서브 클래스입니다. 어트리뷰트를 딕셔너리처럼 보기 원한다면, 표준 파이썬 관용구를 사용할 수 있습니다, `vars()`:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

`ArgumentParser` 가 새 `Namespace` 객체가 아니라 이미 존재하는 객체에 어트리뷰트를 대입하는 것이 유용 할 수 있습니다. `namespace=` 키워드 인자를 지정하면 됩니다:

```
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

17.1.4 기타 유틸리티

부속 명령

ArgumentParser.**add_subparsers**(*[, *title*][, *description*][, *prog*][, *parser_class*][, *action*][, *dest*][, *required*][, *help*][, *metavar*])

Many programs split up their functionality into a number of subcommands, for example, the `svn` program can invoke subcommands like `svn checkout`, `svn update`, and `svn commit`. Splitting up functionality this way can be a particularly good idea when a program performs several different functions which require different kinds of command-line arguments. *ArgumentParser* supports the creation of such subcommands with the `add_subparsers()` method. The `add_subparsers()` method is normally called with no arguments and returns a special action object. This object has a single method, `add_parser()`, which takes a command name and any `ArgumentParser` constructor arguments, and returns an `ArgumentParser` object that can be modified as usual.

매개 변수 설명:

- *title* - title for the sub-parser group in help output; by default "subcommands" if description is provided, otherwise uses title for positional arguments
- *description* - description for the sub-parser group in help output, by default `None`
- *prog* - usage information that will be displayed with sub-command help, by default the name of the program and any positional arguments before the subparser argument
- *parser_class* - class which will be used to create sub-parser instances, by default the class of the current parser (e.g. *ArgumentParser*)
- *action* - 이 인자를 명령행에서 만날 때 수행 할 액션의 기본형
- *dest* - 부속 명령 이름을 저장하는 어트리뷰트의 이름. 기본적으로 `None` 이며 값은 저장되지 않습니다.
- *required* - 부속 명령이 꼭 제공되어야 하는지 아닌지, 기본값은 `False` (3.7에서 추가)
- *help* - 도움말 출력의 부속 파서 그룹 도움말, 기본적으로 `None`
- *metavar* - string presenting available subcommands in help; by default it is `None` and presents subcommands in form {cmd1, cmd2, ..}

몇 가지 사용 예:

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='subcommand help')
>>>
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices=('X', 'Y', 'Z'), help='baz help')
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

parse_args() 에 의해 반환된 객체는 주 파서와 명령행에 의해 선택된 부속 파서(다른 부속 파서는 아님)의 어트리뷰트만을 포함한다는 것에 주의하십시오. 그래서 위의 예에서, `a` 명령이 지정되면 `foo` 와 `bar` 어트리뷰트 만 존재하고, `b` 명령이 지정되면 `foo` 와 `baz` 어트리뷰트만 존재합니다.

Similarly, when a help message is requested from a subparser, only the help for that particular parser will be printed. The help message will not include parent parser or sibling parser messages. (A help message for each subparser command, however, can be given by supplying the `help=` argument to `add_parser()` as above.)

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   subcommand help
    a     a help
    b     b help

options:
  -h, --help  show this help message and exit
  --foo   foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help

options:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

options:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

add_subparsers() 메서드는 또한 `title` 과 `description` 키워드 인자를 지원합니다. 둘 중 하나가 있으면 부속 파서의 명령이 도움말 출력에서 자체 그룹으로 나타납니다. 예를 들면:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                    description='valid subcommands',
...                                    help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

options:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

Furthermore, `add_parser()` supports an additional *aliases* argument, which allows multiple strings to refer to the same subparser. This example, like `svn`, aliases `co` as a shorthand for `checkout`:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

`add_parser()` supports also an additional *deprecated* argument, which allows to deprecate the subparser.

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='chicken.py')
>>> subparsers = parser.add_subparsers()
>>> run = subparsers.add_parser('run')
>>> fly = subparsers.add_parser('fly', deprecated=True)
>>> parser.parse_args(['fly'])
chicken.py: warning: command 'fly' is deprecated
Namespace()
```

Added in version 3.13.

One particularly effective way of handling subcommands is to combine the use of the *add_subparsers()* method with calls to *set_defaults()* so that each subparser knows which Python function it should execute. For example:

```
>>> # subcommand functions
>>> def foo(args):
...     print(args.x * args.y)
...
>>> def bar(args):
...     print('((%s))' % args.z)
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(required=True)
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

이렇게 하면 *parse_args()* 가 파싱이 완료된 후 적절한 함수를 호출하게 할 수 있습니다. 이처럼 액션과 함수를 연결하는 것이, 일반적으로 각 부속 파서가 서로 다른 액션을 처리하도록 하는 가장 쉬운 방법입니다. 그러나 호출된 부속 파서의 이름을 확인해야 하는 경우 *add_subparsers()* 호출에 `dest` 키워드 인자를 제공합니다:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

버전 3.7에서 변경: New *required* keyword-only parameter.

버전 3.14에서 변경: Subparser' s *prog* is no longer affected by a custom usage message in the main parser.

FileType 객체

class argparse.**FileType**(*mode='r'*, *bufsize=-1*, *encoding=None*, *errors=None*)

`FileType` 팩토리는 `ArgumentParser.add_argument()` 의 type 인자로 전달될 수 있는 객체를 만듭니다. type으로 `FileType` 객체를 사용하는 인자는 명령행 인자를 요청된 모드, 버퍼 크기, 인코딩 및 오류 처리의 파일로 엽니다 (자세한 내용은 `open()` 함수를 참조하세요):

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w', encoding='UTF-8'))
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>,␣
↪raw=<_io.FileIO name='raw.dat' mode='wb'>)
```

FileType objects understand the pseudo-argument `'-'` and automatically convert this into `sys.stdin` for readable `FileType` objects and `sys.stdout` for writable `FileType` objects:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

> **참고**
>
> If one argument uses *FileType* and then a subsequent argument fails, an error is reported but the file is not automatically closed. This can also clobber the output files. In this case, it would be better to wait until after the parser has run and then use the `with`-statement to manage the files.

버전 3.4에서 변경: Added the *encodings* and *errors* parameters.

버전 3.14부터 폐지됨.

인자 그룹

ArgumentParser.**add_argument_group**(*title=None*, *description=None*, **[, argument_default][, conflict_handler]*)

By default, `ArgumentParser` groups command-line arguments into "positional arguments" and "options" when displaying help messages. When there is a better conceptual grouping of arguments than this default one, appropriate groups can be created using the `add_argument_group()` method:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  bar     bar help
  --foo FOO  foo help
```

The `add_argument_group()` method returns an argument group object which has an `add_argument()` method just like a regular `ArgumentParser`. When an argument is added to the group, the parser treats it just like a normal argument, but displays the argument in a separate group for help messages. The `add_argument_group()` method accepts *title* and *description* arguments which can be used to customize this display:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description

  --bar BAR  bar help
```

The optional, keyword-only parameters *argument_default* and *conflict_handler* allow for finer-grained control of the behavior of the argument group. These parameters have the same meaning as in the `ArgumentParser` constructor, but apply specifically to the argument group rather than the entire parser.

사용자 정의 그룹에 없는 인자는 일반적인 "positional arguments" 및 "optional arguments" 섹션으로 들어갑니다.

Deprecated since version 3.11, removed in version 3.14: Calling `add_argument_group()` on an argument group now raises an exception. This nesting was never supported, often failed to work correctly, and was unintentionally exposed through inheritance.

버전 3.14부터 폐지됨: Passing *prefix_chars* to `add_argument_group()` is now deprecated.

상호 배제

`ArgumentParser.`**`add_mutually_exclusive_group`**(*required=False*)

Create a mutually exclusive group. `argparse` will make sure that only one of the arguments in the mutually exclusive group was present on the command line:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

add_mutually_exclusive_group() 메서드는 상호 배타적 인자 중 적어도 하나가 필요하다는 것을 나타내기 위한 *required* 인자도 받아들입니다:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

Note that currently mutually exclusive argument groups do not support the *title* and *description* arguments of *add_argument_group()*. However, a mutually exclusive group can be added to an argument group that has a title and description. For example:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_argument_group('Group title', 'Group description')
>>> exclusive_group = group.add_mutually_exclusive_group(required=True)
>>> exclusive_group.add_argument('--foo', help='foo help')
>>> exclusive_group.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [-h] (--foo FOO | --bar BAR)

options:
  -h, --help  show this help message and exit

Group title:
  Group description

  --foo FOO   foo help
  --bar BAR   bar help
```

Deprecated since version 3.11, removed in version 3.14: Calling *add_argument_group()* or *add_mutually_exclusive_group()* on a mutually exclusive group now raises an exception. This nesting was never supported, often failed to work correctly, and was unintentionally exposed through inheritance.

파서 기본값

`ArgumentParser.`**`set_defaults`**(***kwargs*)

대부분은, *parse_args()* 에 의해 반환된 객체의 어트리뷰트는 명령행 인자와 인자 액션을 검사하여 완전히 결정됩니다. *set_defaults()* 는 명령행을 검사하지 않고 결정되는 몇 가지 추가적인 어트리뷰트를 추가할 수 있도록 합니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

파서 수준의 기본값은 항상 인자 수준의 기본값보다 우선합니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

파서 수준의 기본값은 여러 파서로 작업 할 때 특히 유용 할 수 있습니다. 이 유형의 예제는 *add_subparsers()* 메서드를 참조하십시오.

ArgumentParser.**get_default**(*dest*)

add_argument() 또는 set_defaults() 에 의해 설정된 이름 공간 어트리뷰트의 기본값을 가져옵니다:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

도움말 인쇄

대부분의 일반적인 응용 프로그램에서 parse_args() 가 사용법이나 오류 메시지를 포매팅하고 인쇄하는 것을 담당합니다. 그러나 여러 가지 포매팅 방법이 제공됩니다:

ArgumentParser.**print_usage**(*file=None*)

ArgumentParser 가 어떻게 명령행에서 호출되어야 하는지에 대한 간단한 설명을 인쇄합니다. *file* 이 None 이면, sys.stdout 이 가정됩니다.

ArgumentParser.**print_help**(*file=None*)

프로그램 사용법과 ArgumentParser 에 등록된 인자에 대한 정보를 포함하는 도움말 메시지를 출력합니다. *file* 이 None 이면, sys.stdout 이 가정됩니다.

인쇄하는 대신 단순히 문자열을 반환하는, 이러한 메서드의 변형도 있습니다:

ArgumentParser.**format_usage**()

ArgumentParser 가 어떻게 명령행에서 호출되어야 하는지에 대한 간단한 설명을 담은 문자열을 반환합니다.

ArgumentParser.**format_help**()

프로그램 사용법과 ArgumentParser 에 등록된 인자에 대한 정보를 포함하는 도움말 메시지를 담은 문자열을 반환합니다.

부분 파싱

ArgumentParser.**parse_known_args**(*args=None, namespace=None*)

Sometimes a script only needs to handle a specific set of command-line arguments, leaving any unrecognized arguments for another script or program. In these cases, the parse_known_args() method can be useful.

This method works similarly to parse_args(), but it does not raise an error for extra, unrecognized arguments. Instead, it parses the known arguments and returns a two item tuple that contains the populated namespace and the list of any unrecognized arguments.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

경고

Prefix matching rules apply to parse_known_args(). The parser may consume an option even if it' s just a prefix of one of its known options, instead of leaving it in the remaining arguments list.

파일 파싱 사용자 정의

ArgumentParser.**convert_arg_line_to_args**(*arg_line*)

파일에서 읽은 인자는 (ArgumentParser 생성자의 *fromfile_prefix_chars* 키워드 인자를 참조하세요) 한 줄에 하나의 인자로 읽습니다. 이 동작을 변경하려면 convert_arg_line_to_args() 를 재정의합니다.

이 메서드는 하나의 인자 *arg_line* 를 받아들이는데, 인자 파일에서 읽어 들인 문자열입니다. 이 문자열에서 파싱된 인자 리스트를 반환합니다. 메서드는 인자 파일에서 읽어 들이는 대로 한 줄에 한 번씩 순서대로 호출됩니다.

이 메서드의 재정의하는 유용한 경우는 스페이스로 분리된 각 단어를 인자로 처리하는 것입니다. 다음 예제는 이렇게 하는 방법을 보여줍니다:

```
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

종료 메서드

`ArgumentParser.`**`exit`**(*status=0, message=None*)

This method terminates the program, exiting with the specified *status* and, if given, it prints a *message* to `sys.stderr` before that. The user can override this method to handle these steps differently:

```
class ErrorCatchingArgumentParser(argparse.ArgumentParser):
    def exit(self, status=0, message=None):
        if status:
            raise Exception(f'Exiting because of an error: {message}')
        exit(status)
```

`ArgumentParser.`**`error`**(*message*)

This method prints a usage message, including the *message*, to `sys.stderr` and terminates the program with a status code of 2.

혼합 파싱

`ArgumentParser.`**`parse_intermixed_args`**(*args=None, namespace=None*)

`ArgumentParser.`**`parse_known_intermixed_args`**(*args=None, namespace=None*)

많은 유닉스 명령은 사용자가 선택 인자와 위치 인자를 섞을 수 있도록 합니다. `parse_intermixed_args()` 와 `parse_known_intermixed_args()` 메서드는 이런 파싱 스타일을 지원합니다.

These parsers do not support all the `argparse` features, and will raise exceptions if unsupported features are used. In particular, subparsers, and mutually exclusive groups that include both optionals and positionals are not supported.

다음 예제는 `parse_known_args()` 와 `parse_intermixed_args()` 의 차이점을 보여줍니다: 전자는 여분의 인자로 `['2', '3']` 을 반환하는 반면, 후자는 모든 위치 인자를 `rest` 로 모읍니다.

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('cmd')
>>> parser.add_argument('rest', nargs='*', type=int)
>>> parser.parse_known_args('doit 1 --foo bar 2 3'.split())
(Namespace(cmd='doit', foo='bar', rest=[1]), ['2', '3'])
>>> parser.parse_intermixed_args('doit 1 --foo bar 2 3'.split())
Namespace(cmd='doit', foo='bar', rest=[1, 2, 3])
```

`parse_known_intermixed_args()` 는 채워진 이름 공간과 잔여 인자 문자열의 리스트를 포함하는 두 항목 튜플을 반환합니다. `parse_intermixed_args()` 는 파싱되지 않은 인자 문자열이 남아 있으면 에러를 발생시킵니다.

Added in version 3.7.

Registering custom types or actions

`ArgumentParser.register(registry_name, value, object)`

Sometimes it' s desirable to use a custom string in error messages to provide more user-friendly output. In these cases, `register()` can be used to register custom actions or types with a parser and allow you to reference the type by their registered name instead of their callable name.

The `register()` method accepts three arguments - a *registry_name*, specifying the internal registry where the object will be stored (e.g., `action`, `type`), *value*, which is the key under which the object will be registered, and object, the callable to be registered.

The following example shows how to register a custom type with a parser:

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.register('type', 'hexadecimal integer', lambda s: int(s, 16))
>>> parser.add_argument('--foo', type='hexadecimal integer')
_StoreAction(option_strings=['--foo'], dest='foo', nargs=None, const=None,␣
↪default=None, type='hexadecimal integer', choices=None, required=False,␣
↪help=None, metavar=None, deprecated=False)
>>> parser.parse_args(['--foo', '0xFA'])
Namespace(foo=250)
>>> parser.parse_args(['--foo', '1.2'])
usage: PROG [-h] [--foo FOO]
PROG: error: argument --foo: invalid 'hexadecimal integer' value: '1.2'
```

17.1.5 Exceptions

exception `argparse.ArgumentError`

An error from creating or using an argument (optional or positional).

The string value of this exception is the message, augmented with information about the argument that caused it.

exception `argparse.ArgumentTypeError`

Raised when something goes wrong converting a command line string to a type.

Guides and Tutorials

Argparse 자습서

저자
Tshepang Mbambo

이 자습서는 파이썬 표준 라이브러리에서 권장하는 명령행 파싱 모듈인 *argparse* 에 대한 소개입니다.

> **참고**
>
> The standard library includes two other libraries directly related to command-line parameter processing: the lower level *optparse* module (which may require more code to configure for a given application, but also allows an application to request behaviors that `argparse` doesn' t support), and the very low level *getopt* (which specifically serves as an equivalent to the `getopt()` family of functions available to C programmers). While neither of those modules is covered directly in this guide, many of the core concepts in `argparse` first originated in `optparse`, so some aspects of this tutorial will also be relevant to `optparse` users.

개념

ls 명령을 사용하여 이 입문서에서 다룰 기능들을 살펴봅시다:

```
$ ls
cpython  devguide  prog.py  pypy  rm-unused-function.patch
$ ls pypy
ctypes_configure  demo  dotviewer  include  lib_pypy  lib-python ...
$ ls -l
total 20
drwxr-xr-x 19 wena wena 4096 Feb 18 18:51 cpython
drwxr-xr-x  4 wena wena 4096 Feb  8 12:04 devguide
-rwxr-xr-x  1 wena wena  535 Feb 19 00:05 prog.py
drwxr-xr-x 14 wena wena 4096 Feb  7 00:59 pypy
-rw-r--r--  1 wena wena  741 Feb 18 01:01 rm-unused-function.patch
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
...
```

네 가지 명령에서 배울 수 있는 몇 가지 개념들입니다:

- **ls** 명령은 옵션 없이 실행될 때도 유용합니다. 기본적으로 현재 디렉터리의 내용을 표시합니다.
- 기본적으로 제공하는 것 이상으로 원한다면, 조금 더 말합니다. 이 경우에는 다른 디렉터리인 `pypy` 를 표시하기를 원합니다. 우리가 한 것은 위치 인자라고 알려진 것을 지정하는 것입니다. 프로그램이 명령행에 표시되는 위치를 기준으로 값을 어떻게 처리해야 하는지를 알아야 하므로 이런 이름이 사용됩니다. 이 개념은 **cp** 와 같은 명령에 더 적절합니다. 가장 기본적인 사용법은 `cp SRC DEST` 입니다. 첫 번째 위치는 복사하고자 하는 것 이고 두 번째 위치는 사본을 저장할 곳 입니다.
- 자, 프로그램의 행동을 바꾸고 싶다고 합시다. 이 예에서는 파일 이름만 표시하는 대신 각 파일에 대한 정보를 더 많이 표시합니다. 이 경우 `-l` 은 옵션 인자로 알려져 있습니다.
- 이것이 도움말 텍스트입니다. 이전에 사용해보지 않은 프로그램을 접했을 때 도움말 텍스트를 읽는 것만으로 작동하는 방식을 이해할 수 있다는 점에서 매우 유용합니다.

기본

(거의) 아무것도 하지 않는 아주 간단한 예제로 시작합시다:

```
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

다음은 코드를 실행한 결과입니다:

```
$ python prog.py
$ python prog.py --help
usage: prog.py [-h]

options:
  -h, --help  show this help message and exit
$ python prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

일어난 일은 이렇습니다:

- 옵션 없이 스크립트를 실행하면 아무것도 표준 출력에 표시되지 않습니다. 별로 유용하지 않습니다.
- 두 번째는 *argparse* 모듈의 쓸모를 보여주기 시작합니다. 거의 아무것도 하지 않았지만 이미 도움말을 얻었습니다.
- `--help` 옵션은, `-h` 로 단축할 수도 있습니다, 무료로 얻을 수 있는 유일한 옵션입니다 (즉, 지정할 필요가 없습니다). 다른 값을 지정하면 오류가 발생합니다. 그러나 그때조차도 우리는 사용 안내를 얻습니다, 여전히 공짜입니다.

위치 인자 소개

예:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
args = parser.parse_args()
print(args.echo)
```

코드를 실행합니다:

```
$ python prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python prog.py --help
usage: prog.py [-h] echo

positional arguments:
  echo

options:
  -h, --help  show this help message and exit
$ python prog.py foo
foo
```

이런 일이 일어났습니다:

- *add_argument()* 메서드를 추가했습니다. 이 메서드는 프로그램이 받고 싶은 명령행 옵션을 지정하기 위해 사용합니다. 이 경우 기능과 일치하도록 `echo` 라고 이름 붙였습니다.
- 이제 프로그램을 호출하려면 옵션을 지정해야 합니다.
- *parse_args()* 메서드는 실제로 지정된 옵션으로부터 온 데이터를 돌려줍니다. 이 경우에는 `echo` 입니다.
- 변수는 *argparse* 이 공짜로 수행하는 일종의 '마법' 입니다 (즉, 값이 저장되는 변수를 지정할 필요가 없습니다). 또한, 그 이름이 메서드에 주어진 문자열 인자 `echo` 와 일치함을 알 수 있습니다.

그러나 도움말이 멋지게 보이지만, 현재로서는 가능한 최선이 아닙니다. 예를 들어 `echo` 가 위치 인자임을 볼 수 있지만, 추측하거나 소스 코드를 읽는 것 외에는 그것이 무엇을 하는지 모릅니다. 그럼 좀 더 유용하게 만들어 봅시다:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use here")
args = parser.parse_args()
print(args.echo)
```

그러면 이렇게 됩니다:

```
$ python prog.py -h
usage: prog.py [-h] echo
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
positional arguments:
  echo        echo the string you use here

options:
  -h, --help  show this help message and exit
```

이제, 뭔가 더 쓸모있는 일을 하는 것은 어떻습니까:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number")
args = parser.parse_args()
print(args.square**2)
```

다음은 코드를 실행한 결과입니다:

```
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 5, in <module>
    print(args.square**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

잘 안됐습니다. 우리가 달리 지시하지 않는다면, *argparse* 는 우리가 준 옵션들을 문자열로 취급하기 때문입니다. 그럼, *argparse* 에게 그 입력을 정수로 취급하라고 알려줍시다:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number",
                    type=int)
args = parser.parse_args()
print(args.square**2)
```

다음은 코드를 실행한 결과입니다:

```
$ python prog.py 4
16
$ python prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four'
```

잘 됩니다. 이제 이 프로그램은 잘못된 입력이 올 때 더 진행하지 않고 종료하기조차 합니다.

옵션 인자 소개

지금까지 우리는 위치 인자를 다뤘습니다. 옵션 인자를 추가하는 방법에 대해 살펴봅시다:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity")
args = parser.parse_args()
if args.verbosity:
    print("verbosity turned on")
```

출력은 이렇습니다:

```
$ python prog.py --verbosity 1
verbosity turned on
$ python prog.py
$ python prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]

options:
  -h, --help            show this help message and exit
  --verbosity VERBOSITY
                        increase output verbosity
$ python prog.py --verbosity
usage: prog.py [-h] [--verbosity VERBOSITY]
prog.py: error: argument --verbosity: expected one argument
```

일어난 일은 이렇습니다:

- 이 프로그램은 `--verbosity` 가 지정되었을 때 어떤 것을 표시하고 그렇지 않을 때는 아무것도 표시하지 않도록 작성되었습니다.
- 옵션이 실제로 선택 사항임을 확인하기 위해, 이 옵션을 사용하지 않고 프로그램을 실행할 때 오류가 없습니다. 기본적으로 옵션 인자가 사용되지 않는다면 관련 변수 (이 경우 `args.verbosity`)는 값으로 `None` 이 주어집니다. 이 때문에 `if` 문의 논리값 검사가 실패합니다.
- 도움말 메시지가 약간 달라졌습니다.
- `--verbosity` 옵션을 사용할 때, 어떤 값을 지정해야 합니다. 어떤 값이건 상관없습니다.

위의 예제는 `--verbosity` 에 임의의 정숫값을 허용하지만, 우리의 간단한 프로그램에서는 실제로 `True` 또는 `False` 두 값만 쓸모 있습니다. 그것에 맞게 코드를 수정합시다:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

출력은 이렇습니다:

```
$ python prog.py --verbose
verbosity turned on
$ python prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python prog.py --help
usage: prog.py [-h] [--verbose]

options:
  -h, --help  show this help message and exit
  --verbose   increase output verbosity
```

일어난 일은 이렇습니다:

- 이 옵션은 이제 값을 요구하는 것이 아니라 플래그입니다. 이 개념과 일치하도록 옵션의 이름을 변경하기까지 했습니다. 새로운 키워드 `action` 을 지정하고, `"store_true"` 값을 지정했습니다. 이것은, 옵션이 지정되면 `args.verbose` 에 값 `True` 를 대입하라는 뜻입니다. 지정하지 않으면 묵시적으로 `False` 입니다.
- 값을 지정하면 불평하는데, 플래그의 정의를 따르고 있습니다.
- 도움말 텍스트가 바뀐 것을 확인하십시오.

짧은 옵션

명령행 사용법에 익숙하다면 짧은 옵션 버전에 관한 내용을 아직 다루지 않았음을 알 수 있을 겁니다. 아주 간단합니다:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-v", "--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

그러면 이렇게 됩니다:

```
$ python prog.py -v
verbosity turned on
$ python prog.py --help
usage: prog.py [-h] [-v]

options:
  -h, --help     show this help message and exit
  -v, --verbose  increase output verbosity
```

새로운 기능은 도움말 텍스트에도 반영됩니다.

위치 및 옵션 인자 결합하기

프로그램이 점점 복잡해지고 있습니다:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbose", action="store_true",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose:
    print(f"the square of {args.square} equals {answer}")
else:
    print(answer)
```

이제 출력은 이렇게 됩니다:

```
$ python prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required: square
$ python prog.py 4
16
$ python prog.py 4 --verbose
the square of 4 equals 16
$ python prog.py --verbose 4
the square of 4 equals 16
```

- 위치 인자를 다시 도입했기 때문에, 불평합니다.
- 순서는 중요하지 않습니다.

이 프로그램에 여러 상세도를 지정할 수 있도록 하는 능력을 다시 부여하고, 실제로 그것을 사용하는 것은 어떨까요:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

출력은 이렇습니다:

```
$ python prog.py 4
16
$ python prog.py 4 -v
usage: prog.py [-h] [-v VERBOSITY] square
prog.py: error: argument -v/--verbosity: expected one argument
$ python prog.py 4 -v 1
4^2 == 16
$ python prog.py 4 -v 2
the square of 4 equals 16
$ python prog.py 4 -v 3
16
```

우리 프로그램의 버그를 드러내는 마지막 것을 제외하고는 그럴듯해 보입니다. `--verbosity` 옵션이 받아들일 수 있는 값을 제한해서 고쳐봅시다:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int, choices=[0, 1, 2],
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

출력은 이렇습니다:

```
$ python prog.py 4 -v 3
usage: prog.py [-h] [-v {0,1,2}] square
prog.py: error: argument -v/--verbosity: invalid choice: 3 (choose from 0, 1, 2)
$ python prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
positional arguments:
  square                display a square of a given number

options:
  -h, --help            show this help message and exit
  -v, --verbosity {0,1,2}
                        increase output verbosity
```

변경 내용은 오류 메시지와 도움말 문자열에도 반영됩니다.

이제 상세도를 다루는 다른 접근법을 사용해 봅시다, 이 방법은 꽤 널리 사용됩니다. 또한, 이 방법은 CPython 실행 파일이 자신의 상세도를 처리하는 방식과도 일치합니다 (`python --help` 의 결과를 확인하십시오):

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display the square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

특정 옵션을 지정한 횟수를 계산하기 위해 "count" 라는 또 다른 액션을 도입했습니다.

```
$ python prog.py 4
16
$ python prog.py 4 -v
4^2 == 16
$ python prog.py 4 -vv
the square of 4 equals 16
$ python prog.py 4 --verbosity --verbosity
the square of 4 equals 16
$ python prog.py 4 -v 1
usage: prog.py [-h] [-v] square
prog.py: error: unrecognized arguments: 1
$ python prog.py 4 -h
usage: prog.py [-h] [-v] square

positional arguments:
  square           display a square of a given number

options:
  -h, --help       show this help message and exit
  -v, --verbosity  increase output verbosity
$ python prog.py 4 -vvv
16
```

- 예, 이제 이전 버전의 스크립트처럼 (`action="store_true"` 와 유사하게) 플래그가 되었습니다. 출력되는 불평이 설명됩니다.
- 또한 "store_true" 액션과 비슷하게 작동하기도 합니다.

- 이제 여기에서 "count" 액션이 제공하는 것을 보여줍니다. 이런 종류의 사용법을 전에도 보았을 것입니다.
- 그리고, `-v` 플래그를 지정하지 않으면 그 플래그는 `None` 값으로 간주합니다.
- 예측하듯이, 플래그의 긴 형식을 지정하면, 같은 출력이 얻어져야 합니다.
- 안타깝게도 스크립트가 얻은 새로운 기능에 대한 도움말 출력은 그다지 유익하지 않지만, 스크립트의 문서를 개선하면 항상 해결할 수 있습니다 (예, `help` 키워드 인자를 사용해서).
- 마지막 출력은 우리 프로그램의 버그를 노출합니다.

고칩시다:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2

# bugfix: replace == with >=
if args.verbosity >= 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

그러면 이렇게 됩니다:

```
$ python prog.py 4 -vvv
the square of 4 equals 16
$ python prog.py 4 -vvvv
the square of 4 equals 16
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 11, in <module>
    if args.verbosity >= 2:
TypeError: '>=' not supported between instances of 'NoneType' and 'int'
```

- 첫 번째 출력은 잘 동작하고, 앞에서 나온 버그를 고칩니다. 즉, 2보다 크거나 같은 (>=) 모든 값을 최대의 상세도로 취급하고 싶습니다.
- 세 번째 결과가 좋지 않습니다.

이 버그를 고쳐 봅시다:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count", default=0,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity >= 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

또 다른 키워드 `default` 를 소개했습니다. 다른 int 값과 비교하기 위해 `0` 으로 설정했습니다. 기본적으로, 옵션 인자가 지정되지 않으면 `None` 값을 갖게 되고, 그것은 int 값과 비교될 수 없음을 (그래서 *TypeError* 예외를 일으킵니다) 기억하십시오.

그리고:

```
$ python prog.py 4
16
```

여러분은 지금까지 배운 것만으로도 아주 멀리 갈 수 있으며, 우리는 단지 표면을 긁었을 뿐입니다. *argparse* 모듈은 매우 강력합니다. 이 자습서를 끝내기 전에 좀 더 탐색해 보겠습니다.

조금 더 발전시키기

우리의 작은 프로그램을 확장하여 제곱만이 아닌 다른 거듭제곱을 수행하기를 원하면 어떻게 될까요:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print(f"{args.x} to the power {args.y} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.x}^{args.y} == {answer}")
else:
    print(answer)
```

출력:

```
$ python prog.py
usage: prog.py [-h] [-v] x y
prog.py: error: the following arguments are required: x, y
$ python prog.py -h
usage: prog.py [-h] [-v] x y

positional arguments:
  x                 the base
  y                 the exponent

options:
  -h, --help        show this help message and exit
  -v, --verbosity
$ python prog.py 4 2 -v
4^2 == 16
```

지금까지는 표시되는 텍스트를 변경 하기 위해 상세도를 사용했습니다. 다음 예제는 대신 더 많은 텍스트를 표시하기 위해 상세도를 사용합니다:

```
import argparse
parser = argparse.ArgumentParser()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print(f"Running '{__file__}'")
if args.verbosity >= 1:
    print(f"{args.x}^{args.y} == ", end="")
print(answer)
```

출력:

```
$ python prog.py 4 2
16
$ python prog.py 4 2 -v
4^2 == 16
$ python prog.py 4 2 -vv
Running 'prog.py'
4^2 == 16
```

Specifying ambiguous arguments

When there is ambiguity in deciding whether an argument is positional or for an argument, `--` can be used to tell *parse_args()* that everything after that is a positional argument:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-n', nargs='+')
>>> parser.add_argument('args', nargs='*')

>>> # ambiguous, so parse_args assumes it's an option
>>> parser.parse_args(['-f'])
usage: PROG [-h] [-n N [N ...]] [args ...]
PROG: error: unrecognized arguments: -f

>>> parser.parse_args(['--', '-f'])
Namespace(args=['-f'], n=None)

>>> # ambiguous, so the -n option greedily accepts arguments
>>> parser.parse_args(['-n', '1', '2', '3'])
Namespace(args=[], n=['1', '2', '3'])

>>> parser.parse_args(['-n', '1', '--', '2', '3'])
Namespace(args=['2', '3'], n=['1'])
```

충돌하는 옵션들

지금까지 우리는 *argparse.ArgumentParser* 인스턴스의 두 가지 메서드로 작업 해왔습니다. 세 번째를 소개합시다, *add_mutually_exclusive_group()*. 이것은 서로 배타적인 옵션을 지정할 수 있도록 합니다. 새로운 기능을 더 잘 이해할 수 있도록 프로그램의 나머지 부분을 변경해 보겠습니다: `--quiet` 옵션을 도입하는데 `--verbose` 의 반대입니다:

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print(f"{args.x} to the power {args.y} equals {answer}")
else:
    print(f"{args.x}^{args.y} == {answer}")
```

프로그램은 이제 더 간단해졌으며, 데모를 위해 일부 기능을 잃어버렸습니다. 어쨌든, 출력은 이렇습니다:

```
$ python prog.py 4 2
4^2 == 16
$ python prog.py 4 2 -q
16
$ python prog.py 4 2 -v
4 to the power 2 equals 16
$ python prog.py 4 2 -vq
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
$ python prog.py 4 2 -v --quiet
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
```

따라가기 쉽습니다. 여러분이 얻는 유연성을 볼 수 있도록 마지막 출력을 추가했습니다. 즉, 긴 형식 옵션을 짧은 형식 옵션과 섞어 쓸 수 있습니다.

결론을 내리기 전에, 여러분은 아마도 사용자들이 모를 경우를 대비해서 프로그램의 주요 목적을 알려주기를 원할 것입니다:

```
import argparse

parser = argparse.ArgumentParser(description="calculate X to the power of Y")
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print(f"{args.x} to the power {args.y} equals {answer}")
else:
    print(f"{args.x}^{args.y} == {answer}")
```

사용법 텍스트의 약간의 차이점에 유의하십시오. `[-v | -q]` 에 주목해야 하는데, `-v` 나 `-q` 를 사용할 수 있지만 동시에 둘 다를 사용할 수는 없다는 뜻입니다:

```
$ python prog.py --help
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x               the base
  y               the exponent

options:
  -h, --help      show this help message and exit
  -v, --verbose
  -q, --quiet
```

How to translate the argparse output

The output of the *argparse* module such as its help text and error messages are all made translatable using the *gettext* module. This allows applications to easily localize messages produced by *argparse*. See also 프로그램과 모듈의 국제화.

For instance, in this *argparse* output:

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x               the base
  y               the exponent

options:
  -h, --help      show this help message and exit
  -v, --verbose
  -q, --quiet
```

The strings `usage:`, `positional arguments:`, `options:` and `show this help message and exit` are all translatable.

In order to translate these strings, they must first be extracted into a `.po` file. For example, using Babel, run this command:

```
$ pybabel extract -o messages.po /usr/lib/python3.12/argparse.py
```

This command will extract all translatable strings from the *argparse* module and output them into a file named `messages.po`. This command assumes that your Python installation is in `/usr/lib`.

You can find out the location of the *argparse* module on your system using this script:

```
import argparse
print(argparse.__file__)
```

Once the messages in the `.po` file are translated and the translations are installed using *gettext*, *argparse* will be able to display the translated messages.

To translate your own strings in the *argparse* output, use *gettext*.

Custom type converters

The *argparse* module allows you to specify custom type converters for your command-line arguments. This allows you to modify user input before it' s stored in the *argparse.Namespace*. This can be useful when you need to pre-process the input before it is used in your program.

When using a custom type converter, you can use any callable that takes a single string argument (the argument value) and returns the converted value. However, if you need to handle more complex scenarios, you can use a custom action class with the **action** parameter instead.

For example, let' s say you want to handle arguments with different prefixes and process them accordingly:

```
import argparse

parser = argparse.ArgumentParser(prefix_chars='-+')

parser.add_argument('-a', metavar='<value>', action='append',
                    type=lambda x: ('-', x))
parser.add_argument('+a', metavar='<value>', action='append',
                    type=lambda x: ('+', x))

args = parser.parse_args()
print(args)
```

출력:

```
$ python prog.py -a value1 +a value2
Namespace(a=[('-', 'value1'), ('+', 'value2')])
```

이 예에서,:

- Created a parser with custom prefix characters using the `prefix_chars` parameter.
- Defined two arguments, `-a` and `+a`, which used the `type` parameter to create custom type converters to store the value in a tuple with the prefix.

Without the custom type converters, the arguments would have treated the `-a` and `+a` as the same argument, which would have been undesirable. By using custom type converters, we were able to differentiate between the two arguments.

맺음말

argparse 모듈은 여기에 제시된 것보다 훨씬 많은 것을 제공합니다. 문서는 아주 상세하고 철저하며 예제가 풍부합니다. 이 자습서를 끝내면 압도감 없이 쉽게 소화할 수 있을 겁니다.

Migrating `optparse` code to `argparse`

The *argparse* module offers several higher level features not natively provided by the *optparse* module, including:

- Handling positional arguments.
- Supporting subcommands.
- Allowing alternative option prefixes like + and /.
- Handling zero-or-more and one-or-more style arguments.
- Producing more informative usage messages.
- Providing a much simpler interface for custom `type` and `action`.

Originally, the *argparse* module attempted to maintain compatibility with *optparse*. However, the fundamental design differences between supporting declarative command line option processing (while leaving positional argument processing to application code), and supporting both named options and positional arguments in the declarative interface mean that the API has diverged from that of `optparse` over time.

As described in *Choosing an argument parsing library*, applications that are currently using `optparse` and are happy with the way it works can just continue to use `optparse`.

Application developers that are considering migrating should also review the list of intrinsic behavioural differences described in that section before deciding whether or not migration is desirable.

For applications that do choose to migrate from `optparse` to `argparse`, the following suggestions should be helpful:

- Replace all `optparse.OptionParser.add_option()` calls with `ArgumentParser.add_argument()` calls.
- Replace `(options, args) = parser.parse_args()` with `args = parser.parse_args()` and add additional `ArgumentParser.add_argument()` calls for the positional arguments. Keep in mind that what was previously called `options`, now in the `argparse` context is called `args`.
- Replace `optparse.OptionParser.disable_interspersed_args()` by using `parse_intermixed_args()` instead of `parse_args()`.
- Replace callback actions and the `callback_*` keyword arguments with `type` or `action` arguments.
- Replace string names for `type` keyword arguments with the corresponding type objects (e.g. int, float, complex, etc).
- Replace `optparse.Values` with `Namespace` and `optparse.OptionError` and `optparse.OptionValueError` with `ArgumentError`.
- Replace strings with implicit arguments such as `%default` or `%prog` with the standard Python syntax to use dictionaries to format strings, that is, `%(default)s` and `%(prog)s`.
- Replace the OptionParser constructor `version` argument with a call to `parser.add_argument('--version', action='version', version='<the version>')`.

17.2 optparse — 명령 줄 옵션용 구문 분석기

소스 코드: Lib/optparse.py

17.2.1 Choosing an argument parsing library

The standard library includes three argument parsing libraries:

- `getopt`: a module that closely mirrors the procedural C `getopt` API. Included in the standard library since before the initial Python 1.0 release.
- `optparse`: a declarative replacement for `getopt` that provides equivalent functionality without requiring each application to implement its own procedural option parsing logic. Included in the standard library since the Python 2.3 release.
- `argparse`: a more opinionated alternative to `optparse` that provides more functionality by default, at the expense of reduced application flexibility in controlling exactly how arguments are processed. Included in the standard library since the Python 2.7 and Python 3.2 releases.

In the absence of more specific argument parsing design constraints, `argparse` is the recommended choice for implementing command line applications, as it offers the highest level of baseline functionality with the least application level code.

`getopt` is retained almost entirely for backwards compatibility reasons. However, it also serves a niche use case as a tool for prototyping and testing command line argument handling in `getopt`-based C applications.

`optparse` should be considered as an alternative to `argparse` in the following cases:

- an application is already using `optparse` and doesn' t want to risk the subtle behavioural changes that may arise when migrating to `argparse`
- the application requires additional control over the way options and positional parameters are interleaved on the command line (including the ability to disable the interleaving feature completely)

- the application requires additional control over the incremental parsing of command line elements (while `argparse` does support this, the exact way it works in practice is undesirable for some use cases)
- the application requires additional control over the handling of options which accept parameter values that may start with `-` (such as delegated options to be passed to invoked subprocesses)
- the application requires some other command line parameter processing behavior which `argparse` does not support, but which can be implemented in terms of the lower level interface offered by `optparse`

These considerations also mean that *optparse* is likely to provide a better foundation for library authors writing third party command line argument processing libraries.

As a concrete example, consider the following two command line argument parsing configurations, the first using `optparse`, and the second using `argparse`:

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option('-o', '--output')
    parser.add_option('-v', dest='verbose', action='store_true')
    opts, args = parser.parse_args()
    process(args, output=opts.output, verbose=opts.verbose)
```

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    parser.add_argument('rest', nargs='*')
    args = parser.parse_args()
    process(args.rest, output=args.output, verbose=args.verbose)
```

The most obvious difference is that in the `optparse` version, the non-option arguments are processed separately by the application after the option processing is complete. In the `argparse` version, positional arguments are declared and processed in the same way as the named options.

However, the `argparse` version will also handle some parameter combination differently from the way the `optparse` version would handle them. For example (amongst other differences):

- supplying `-o -v` gives `output="-v"` and `verbose=False` when using `optparse`, but a usage error with `argparse` (complaining that no value has been supplied for `-o/--output`, since `-v` is interpreted as meaning the verbosity flag)
- similarly, supplying `-o --` gives `output="--"` and `args=()` when using `optparse`, but a usage error with `argparse` (also complaining that no value has been supplied for `-o/--output`, since `--` is interpreted as terminating the option processing and treating all remaining values as positional arguments)
- supplying `-o=foo` gives `output="=foo"` when using `optparse`, but gives `output="foo"` with `argparse` (since `=` is special cased as an alternative separator for option parameter values)

Whether these differing behaviors in the `argparse` version are considered desirable or a problem will depend on the specific command line application use case.

더 보기

click is a third party argument processing library (originally based on `optparse`), which allows command line applications to be developed as a set of decorated command implementation functions.

Other third party libraries, such as typer or msgspec-click, allow command line interfaces to be specified in ways that more effectively integrate with static checking of Python type annotations.

17.2.2 Introduction

*optparse*는 최소한의 기능을 제공하는 *getopt* 모듈보다 명령 줄 옵션을 구문 분석하기 위한 더 편리하고 유연하며 강력한 라이브러리입니다. *optparse*는 더 선언적인 스타일의 명령 줄 구문 분석을 사용합니다: *OptionParser*의 인스턴스를 만들고, 옵션으로 채우고, 명령 줄을 구문 분석합니다. *optparse*를 사용하면 사용자가 전통적인 GNU/POSIX 문법으로 옵션을 지정할 수 있으며, 추가로 사용법과 도움말 메시지를 생성할 수 있습니다.

다음은 간단한 스크립트에서 *optparse*를 사용하는 예입니다:

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

이 몇 줄의 코드로, 스크립트 사용자는 이제 명령 줄에서 "일반적인 작업"을 수행 할 수 있습니다, 예를 들면:

```
<yourscript> --file=outfile -q
```

명령 줄을 구문 분석할 때, *optparse*는 사용자가 제공한 명령 줄 값을 기반으로 *parse_args()*에서 반환한 `options` 객체의 어트리뷰트를 설정합니다. 이 명령 줄 구문 분석에서 *parse_args()*가 반환되면, `options.filename`은 `"outfile"`이 되고 `options.verbose`는 `False`가 됩니다. *optparse*는 긴 옵션과 짧은 옵션을 모두 지원하고, 짧은 옵션을 함께 병합하도록 하며, 다양한 방법으로 옵션을 인자와 연관시킬 수 있습니다. 따라서 다음 명령 줄은 모두 위의 예와 동등합니다:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

또한, 사용자는 다음 중 하나를 실행할 수 있습니다

```
<yourscript> -h
<yourscript> --help
```

그러면 *optparse*는 스크립트 옵션에 대한 간략한 요약을 인쇄합니다:

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

여기서 *yourscript*의 값은 (일반적으로 `sys.argv[0]`에서) 실행 시간에 결정됩니다.

17.2.3 배경

*optparse*는 C 개발자가 사용할 수 있는 `getopt()` 함수 계열이 정립한 관례를 따르는 간결한 명령 줄 인터페이스를 갖는 프로그램을 만들 수 있도록 명시적으로 설계되었습니다. 이를 위해, 유닉스에서 전통적으로 사용되는 가장 일반적인 명령 줄 문법과 의미 체계만 지원합니다. 이러한 관례에 익숙하지 않으면, 이 섹션을 읽는 것으로 익히게 될 것입니다.

용어

인자(argument)
명령 줄에 입력하고, 셸에서 `execl()` 이나 `execv()` 로 전달한 문자열. 파이썬에서, 인자는 `sys.argv[1:]` 의 요소입니다 (`sys.argv[0]` 은 실행 중인 프로그램의 이름입니다). 유닉스 셸은 "워드 (word)" 라는 용어도 사용합니다.

때때로 `sys.argv[1:]` 이외의 인자 리스트로 대체하는 것이 바람직해서, "인자" 를 "`sys.argv[1:]` 이나 `sys.argv[1:]` 의 대체로 제공된 다른 리스트의 요소" 로 읽어야 합니다.

옵션(option)
프로그램 실행을 안내하거나 사용자 정의하기 위해 추가 정보를 제공하는 데 사용되는 인자. 옵션에 대한 다양한 문법이 있습니다; 전통적인 유닉스 문법은 하이픈 ("-") 뒤에 단일 문자가 옵니다, 예를 들어 `-x` 나 `-F`. 또한, 전통적인 유닉스 문법은 여러 옵션을 단일 인자로 병합할 수 있도록 합니다, 예를 들어 `-x -F` 는 `-xF` 와 동등합니다. GNU 프로젝트는 하이픈으로 구분된 일련의 단어가 뒤따르는 `--` 를 도입했습니다, 예를 들어 `--file` 이나 `--dry-run`. 이들이 *optparse* 에서 제공하는 유일한 두 가지 옵션 문법입니다.

세상에 등장했던 다른 옵션 문법은 다음과 같습니다:

- 몇 개의 문자가 뒤따르는 하이픈, 예를 들어 `-pf` (이것은 하나의 인자로 병합된 여러 옵션과 같은 것이 아닙니다)
- 전체 단어가 뒤따르는 하이픈, 예를 들어 `-file` (기술적으로 이전 문법과 동등하지만, 일반적으로 같은 프로그램에서 등장하지 않습니다)
- 단일 문자나 몇 개의 문자 또는 단어가 뒤따르는 더하기 기호, 예를 들어 `+f`, `+rgb`
- 단일 문자나 몇 개의 문자 또는 단어가 뒤따르는 슬래시, 예를 들어 `/f`, `/file`

이러한 옵션 문법들은 *optparse* 에서 지원되지 않으며, 앞으로도 지원되지 않을 것입니다. 이것은 의도적입니다: 처음 세 가지는 모든 환경에서 표준이 아니며, 마지막은 윈도우나 특정 레거시 플랫폼 (예를 들어 VMS, MS-DOS) 을 대상으로 하는 경우에만 의미가 있습니다.

옵션 인자(option argument)
옵션 뒤에 오는 인자는 해당 옵션과 밀접하게 연관되어 있으며, 해당 옵션이 있으면 인자 목록에서 사용됩니다. *optparse* 를 사용하면, 옵션 인자가 해당 옵션과 별도의 인자로 있을 수 있습니다:

```
-f foo
--file foo
```

또는 같은 인자에 포함될 수 있습니다:

```
-ffoo
--file=foo
```

일반적으로, 주어진 옵션은 인자를 취하거나 취하지 않습니다. 많은 사람이 "선택적 옵션 인자" 기능을 원합니다. 즉, 일부 옵션은 있다면 인자를 취하고 그렇지 않으면 취하지 않습니다. 이것은 구문 분석을 모호하게 만들기 때문에 다소 논란의 여지가 있습니다: `-a` 가 선택적 인자를 취하고 `-b` 가 완전히 다른 옵션이면 `-ab` 를 어떻게 해석합니까? 이러한 모호성 때문에, *optparse* 는 이 기능을 지원하지 않습니다.

위치 인자(positional argument)
옵션이 구문 분석된 후, 즉 옵션과 해당 인자가 구문 분석되고 인자 목록에서 제거된 후 인자 목록에 남은 것.

필수 옵션(required option)
명령 줄에서 제공해야 하는 옵션; "필수 옵션" 이라는 문구는 영어에서 모순된다는 점에 유의하십시오. *optparse* 는 필수 옵션을 구현하는 것을 방해하지 않지만, 그다지 도움을 주지도 않습니다.

예를 들어, 다음 가상 명령 줄을 고려하십시오:

```
prog -v --report report.txt foo bar
```

-v와 --report는 둘 다 옵션입니다. --report가 하나의 인자를 취한다고 가정하면, report.txt는 옵션 인자입니다. foo와 bar는 위치 인자입니다.

옵션은 무엇을 위한 것입니까?

옵션은 프로그램 실행을 조정하거나 사용자 정의하기 위한 추가 정보를 제공하는 데 사용됩니다. 명확하지 않은 경우, 옵션은 일반적으로 선택적입니다. 프로그램은 어떤 옵션도 없이 잘 실행될 수 있어야 합니다. (유닉스나 GNU 도구 집합에서 임의의 프로그램을 선택하십시오. 옵션 없이도 실행될 수 있으며 여전히 의미가 있습니까? 주요 예외는 find, tar 및 dd입니다 — 모두 비표준 문법과 혼란스러운 인터페이스 때문에 올바로 비판을 받은 돌연변이 괴짜입니다.)

많은 사람이 프로그램에 "필수 옵션"이 있기를 원합니다. 생각해보십시오. 필수라면, 그것은 선택적(*optional*)이 아닙니다! 당신의 프로그램이 성공적으로 실행하기 위해 절대적으로 필요한 정보가 있다면, 그것이 바로 위치 인자의 목적입니다.

좋은 명령 줄 인터페이스 설계의 예로, 파일 복사를 위한 겸손한 cp 유틸리티를 고려하십시오. 대상과 하나 이상의 소스를 제공하지 않고 파일을 복사하는 것은 의미가 없습니다. 따라서, 인자 없이 실행하면 cp가 실패합니다. 그러나 옵션이 전혀 필수로 요구하지 않는 유연하고 유용한 문법을 갖습니다:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

그것만으로도 꽤 멀리 갈 수 있습니다. 대부분의 cp 구현은 파일이 복사되는 방식을 정확하게 조정할 수 있는 여러 옵션을 제공합니다: 모드 및 수정 시간을 보존하고, 심볼릭 링크를 따르지 않고, 기존 파일을 건드리기 전에 물을 수 있습니다. 하지만 이 중 어느 것도 한 파일을 다른 파일로 복사하거나 여러 파일을 다른 디렉터리로 복사하는 cp의 핵심 임무를 방해하지 않습니다.

위치 인자는 무엇을 위한 것입니까?

위치 인자는 프로그램이 실행하기 위해 절대적으로 필요로하는 정보를 위한 것입니다.

좋은 사용자 인터페이스에는 가능한 한 적은 절대 요구 사항이 있어야 합니다. 프로그램을 성공적으로 실행하기 위해 17개의 개별 정보를 요구한다면, 사용자로부터 해당 정보를 어떻게 얻는지는 중요하지 않습니다—대부분의 사람은 프로그램을 성공적으로 실행하기 전에 포기하고 떠납니다. 이것은 사용자 인터페이스가 명령 줄이든, 구성 파일이든, GUI이든 상관없이 적용됩니다: 사용자에게 그렇게 많은 요구를 하면, 대부분은 단순히 포기할 것입니다.

요컨대, 사용자가 절대적으로 제공해야 하는 정보의 양을 최소화하십시오—가능할 때마다 합리적인 기본값을 사용하십시오. 물론, 프로그램을 합리적으로 유연하게 만들고 싶기도 합니다. 그것이 바로 옵션이 있는 이유입니다. 다시 말하지만, 구성 파일의 항목인지, GUI의 "기본 설정" 대화 상자에 있는 위젯인지, 명령 줄 옵션인지는 중요하지 않습니다—구현하는 옵션이 많을수록, 프로그램이 더 유연해지고, 구현은 더 복잡해집니다. 물론 유연성이 너무 많으면 단점도 있습니다; 너무 많은 옵션은 사용자를 압도하고 코드 유지 관리를 훨씬 더 어렵게 만들 수 있습니다.

17.2.4 자습서

*optparse*는 매우 유연하고 강력하지만, 대부분의 경우 사용하기도 간단합니다. 이 섹션에서는 모든 *optparse* 기반 프로그램에 공통적인 코드 패턴을 다룹니다.

먼저, OptionParser 클래스를 임포트 해야 합니다; 그런 다음 메인 프로그램의 초기에, OptionParser 인스턴스를 만듭니다:

```
from optparse import OptionParser
...
parser = OptionParser()
```

그런 다음 옵션 정의를 시작할 수 있습니다. 기본 문법은 다음과 같습니다:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

각 옵션에는 -f나 --file과 같은 하나 이상의 옵션 문자열이 있고, 명령 줄에서 해당 옵션을 발견했을 때 optparse가 기대하는 것과 수행 할 작업을 알려주는 여러 옵션 어트리뷰트가 있습니다.

일반적으로, 각 옵션에는 하나의 짧은 옵션 문자열과 하나의 긴 옵션 문자열이 있습니다, 예를 들어:

```
parser.add_option("-f", "--file", ...)
```

전체적으로 적어도 하나의 옵션 문자열이 있는 한 원하는 만큼 짧은 옵션 문자열과 긴 옵션 문자열을 (없는 것도 포함합니다) 자유롭게 정의 할 수 있습니다.

OptionParser.add_option()에 전달된 옵션 문자열은 해당 호출에 의해 정의된 옵션에 대한 레이블입니다. 간결함을 위해, 명령 줄에서 옵션을 만난다를 자주 언급할 것입니다; 실제로는, optparse가 옵션 문자열을 만나고 이것으로 옵션을 찾습니다.

일단 모든 옵션이 정의되면, optparse가 프로그램의 명령 줄을 구문 분석하도록 지시합니다:

```
(options, args) = parser.parse_args()
```

(원한다면, 사용자 정의 인자 리스트를 parse_args()에 전달할 수 있지만, 거의 필요하지 않습니다: 기본적으로 sys.argv[1:]을 사용합니다.)

parse_args()는 두 가지 값을 반환합니다:

- options, 모든 옵션에 대한 값을 포함하는 객체—예를 들어 --file이 단일 문자열 인자를 취하면, options.file은 사용자가 제공한 파일명이거나, 사용자가 해당 옵션을 제공하지 않으면 None입니다.
- args, 옵션 구문 분석 후 남은 위치 인자 리스트

이 자습서 섹션에서는 가장 중요한 4가지 옵션 어트리뷰트만 다룹니다: action, type, dest (destination) 및 help만 다룹니다. 이 중, action이 가장 기본입니다.

옵션 액션의 이해

액션은 명령 줄에서 옵션을 발견할 때 수행 할 작업을 optparse에 알려줍니다. optparse에 하드 코딩된 고정된 액션 집합이 있습니다; 새로운 액션 추가는 섹션 *optparse 확장하기*에서 다루는 고급 주제입니다. 대부분의 액션은 optparse에게 어떤 변수에 값을 저장하도록 지시합니다—예를 들어, 명령 줄에서 문자열을 취해서 options의 어트리뷰트에 저장합니다.

옵션 액션을 지정하지 않으면, optparse의 기본값은 store입니다.

store 액션

가장 일반적인 옵션 액션은 store로, optparse에게 다음 인자(또는 현재 인자의 나머지)를 취하고, 올바른 형인지 확인한 다음, 선택한 대상에 저장하도록 지시합니다.

예를 들면:

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

이제 가짜 명령 줄을 만들고 optparse에게 구문 분석을 요청합시다:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

optparse가 옵션 문자열 -f를 보면, 다음 인자인 foo.txt를 소비하고, options.filename에 저장합니다. 따라서, 이 parse_args() 호출 후, options.filename은 "foo.txt"입니다.

optparse에서 지원하는 다른 옵션 형은 int와 float입니다. 정수 인자를 기대하는 옵션은 다음과 같습니다:

```
parser.add_option("-n", type="int", dest="num")
```

이 옵션에는 긴 옵션 문자열이 없는데, 완벽하게 허용됩니다. 또한, 기본값이 store이기 때문에 명시적인 액션이 없습니다.

다른 가짜 명령 줄을 구문 분석해 봅시다. 이번에는, 옵션 인자를 옵션 바로 다음에 붙일 것입니다: -n42(하나의 인자)는 -n 42(두 개의 인자)와 동등하므로, 다음 코드는

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

42를 인쇄합니다.

형을 지정하지 않으면, optparse는 string을 가정합니다. 기본 액션이 store라는 사실과 결합하면, 첫 번째 예제를 훨씬 더 짧게 만들 수 있습니다:

```
parser.add_option("-f", "--file", dest="filename")
```

대상을 제공하지 않으면, optparse는 옵션 문자열에서 합리적인 기본값을 추측합니다: 첫 번째 긴 옵션 문자열이 --foo-bar이면 기본 대상은 foo_bar입니다. 긴 옵션 문자열이 없으면, optparse는 첫 번째 짧은 옵션 문자열을 찾습니다: -f의 기본 대상은 f입니다.

optparse는 내장 complex 형도 포함합니다. 형 추가는 섹션 *optparse 확장하기*에서 다룹니다.

불리언 (플래그) 옵션 처리하기

플래그 옵션(특정 옵션이 발견되면 변수를 참이나 거짓으로 설정합니다)은 매우 흔합니다. optparse는 store_true와 store_false의 두 가지 별도의 액션으로 지원합니다. 예를 들어, -v로 켜고 -q로 끄는 verbose 플래그가 있을 수 있습니다:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

여기에 대상이 같은 두 가지 옵션이 있는데, 완벽하게 괜찮습니다. (단지 기본값을 설정할 때 약간 주의해야 함을 뜻합니다—아래를 참조하십시오.)

optparse가 명령 줄에서 -v를 만나면, options.verbose를 True로 설정합니다; -q를 만나면 options.verbose는 False로 설정됩니다.

다른 액션들

optparse에서 지원하는 다른 액션은 다음과 같습니다:

"store_const"
store a constant value, pre-set via *Option.const*

"append"
이 옵션의 인자를 리스트에 추가합니다

"count"
카운터를 1씩 증가시킵니다

"callback"
지정된 함수를 호출합니다

이들은 섹션 레퍼런스 지침서와 섹션 옵션 콜백에서 다룹니다.

기본값

위의 모든 예에는 특정 명령 줄 옵션을 볼 때 일부 변수 ("대상(destination)") 설정이 수반됩니다. 이러한 옵션이 나타나지 않으면 어떻게 될까요? 기본값을 제공하지 않아서, 모두 None으로 설정됩니다. 이것은 일반적으로 괜찮지만, 때로는 더 많은 제어가 필요합니다. optparse는 각 대상에 대한 기본값을 제공할 수 있도록 하는데, 명령 줄이 구문 분석되기 전에 대입됩니다.

먼저, verbose/quiet 예를 고려하십시오. -q가 나타나지 않는 한 optparse가 verbose를 True로 설정하도록 하려면, 다음과 같이 할 수 있습니다:

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

기본값은 특정 옵션이 아닌 대상(*destination*)에 적용되고, 이 두 옵션은 같은 대상을 갖기 때문에, 다음과 정확히 동등합니다:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

이걸 생각해봅시다:

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

다시, `verbose`의 기본값은 `True`입니다: 특정 대상에 대해 제공되는 마지막 기본값이 사용되는 값입니다.

기본값을 지정하는 더 명확한 방법은 OptionParser의 `set_defaults()` 메서드인데, *parse_args()*를 호출하기 전에 언제든지 호출할 수 있습니다:

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

이전과 마찬가지로, 주어진 옵션 대상에 대해 지정된 마지막 값이 사용됩니다. 명확성을 위해, 둘 다가 아닌 한 가지 방법을 사용하여 기본값을 설정하십시오.

도움말 생성하기

도움말과 사용법 텍스트를 자동으로 생성하는 *optparse*의 기능은 사용자 친화적인 명령 줄 인터페이스를 만드는 데 유용합니다. 각 옵션에 대해 *help* 값을 제공하고, 선택적으로 전체 프로그램에 대한 짧은 사용법 메시지를 제공하기만 하면 됩니다. 다음은 사용자에게 친숙한 (문서화된) 옵션으로 채워진 OptionParser 입니다:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                       "or expert [default: %default]")
```

*optparse*가 명령 줄에서 `-h`나 `--help`를 만나거나, `parser.print_help()`를 호출하면, 다음을 표준 출력에 인쇄합니다:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(도움말 출력이 help 옵션으로 트리거 되면, *optparse*가 도움말 텍스트를 인쇄한 후 종료합니다.)

여기에는 *optparse*가 최상의 도움말 메시지를 생성하는 데 도움이 되는 많은 것들이 있습니다:

- 스크립트는 자체 사용법 메시지를 정의합니다:

  ```
  usage = "usage: %prog [options] arg1 arg2"
  ```

  *optparse*는 사용법 문자열의 `%prog`를 현재 프로그램의 이름, 즉 `os.path.basename(sys.argv[0])`으로 확장합니다. 확장된 문자열은 자세한 옵션 도움말 앞에 인쇄됩니다.

  사용법 문자열을 제공하지 않으면, *optparse*는 단순하지만, 합리적인 기본값을 사용합니다: `"Usage: %prog [options]"`, 이는 스크립트가 위치 인자를 취하지 않는다면 괜찮습니다.

- 모든 옵션은 도움말 문자열을 정의하고, 줄 바꿈에 대해 걱정하지 않습니다—*optparse*는 줄 바꿈을 처리하고 도움말 출력을 보기 좋게 만듭니다.

- 값을 취하는 옵션은 자동으로 생성된 도움말 메시지에 이 사실을 나타냅니다, 예를 들어 "mode" 옵션의 경우:

  ```
  -m MODE, --mode=MODE
  ```

  여기서, "MODE"를 메타 변수(meta-variable)라고 합니다: 사용자가 `-m`/`--mode`에 제공할 것으로 기대되는 인자를 나타냅니다. 기본적으로, *optparse*는 대상 변수 이름을 대문자로 변환하고 이를 메타 변수에 사용합니다. 때로는, 이것이 여러분이 원하는 것이 아닙니다—예를 들어, `--filename` 옵션은 명시적으로 `metavar="FILE"`을 설정하여 다음과 같은 자동 생성 옵션 설명을 생성합니다:

  ```
  -f FILE, --filename=FILE
  ```

  이것은 공간을 절약하는 것 이상으로 중요합니다: 수동으로 작성된 도움말 텍스트는 메타 변수 `FILE`을 사용하여 반 형식 구문 `-f FILE`과 비형식적 의미 설명 "write output to FILE" 사이에 연결이 있다는 단서를 사용자에게 알려줍니다. 이는 최종 사용자에게 도움말 텍스트를 훨씬 더 명확하고 유용하게 만드는 간단하지만, 효과적인 방법입니다.

- 기본값이 있는 옵션은 도움말 문자열에 `%default`를 포함할 수 있습니다—*optparse*는 이를 옵션 기본값의 *str()*로 대체합니다. 옵션에 기본값이 없으면 (또는 기본값이 `None`이면), `%default`는 `none`으로 확장됩니다.

옵션 그룹화하기

많은 옵션을 다룰 때, 더 나은 도움말 출력을 위해 이러한 옵션을 그룹화하는 것이 편리합니다. *OptionParser*에는 여러 옵션 그룹이 포함될 수 있으며 각 그룹에는 여러 옵션이 포함될 수 있습니다.

옵션 그룹은 클래스 *OptionGroup*을 사용하여 얻습니다:

class optparse.**OptionGroup**(*parser*, *title*, *description=None*)

여기서

- parser는 그룹이 삽입될 *OptionParser* 인스턴스입니다
- title은 그룹 제목입니다
- description(선택 사항)은 그룹에 대한 자세한 설명입니다

*OptionGroup*은 (*OptionParser* 처럼) `OptionContainer`에서 상속되므로 `add_option()` 메서드를 사용하여 그룹에 옵션을 추가할 수 있습니다.

일단 모든 옵션이 선언되면, *OptionParser* 메서드 `add_option_group()`을 사용하여 그룹이 이전에 정의된 구문 분석기에 추가됩니다.

이전 섹션에서 정의한 구문 분석기로 계속 진행하면, 구문 분석기에 *OptionGroup*을 쉽게 추가할 수 있습니다:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

그러면 다음과 같은 도움말이 출력됩니다:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.
```

좀 더 완전한 예는 둘 이상의 그룹을 수반할 수 있습니다: 여전히 이전 예를 확장합니다:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                 help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                 help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

다음과 같은 출력을 줍니다:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]

  Dangerous Options:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                    Group option.

  Debug Options:
    -d, --debug           Print debug information
    -s, --sql             Print all SQL statements executed
    -e                    Print every action done
```

특히 옵션 그룹을 프로그래밍 방식으로 작업할 때, 또 다른 흥미로운 메서드는 다음과 같습니다:

OptionParser.**get_option_group**(*opt_str*)

짧거나 긴 옵션 문자열 *opt_str*(예를 들어 `'-o'`나 `'--option'`)이 속한 *OptionGroup*을 반환합니다. 그러한 *OptionGroup*이 없으면, `None`을 반환합니다.

버전 문자열 인쇄하기

간단한 사용법 문자열과 유사하게, *optparse*는 프로그램의 버전 문자열을 인쇄 할 수도 있습니다. OptionParser에 `version` 인자로 문자열을 제공해야 합니다:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

`%prog`는 `usage`에서처럼 확장됩니다. 그 외에도, `version`은 원하는 무엇이든 포함할 수 있습니다. 이를 제공하면, *optparse*는 자동으로 구문 분석기에 `--version` 옵션을 추가합니다. 명령 줄에서 이 옵션을 발견하면, (`%prog`를 대체하여) `version` 문자열을 확장하고, stdout에 인쇄한 다음, 종료합니다.

예를 들어, 스크립트가 `/usr/bin/foo`로 호출되면:

```
$ /usr/bin/foo --version
foo 1.0
```

다음 두 가지 메서드를 사용하여 `version` 문자열을 인쇄하고 가져올 수 있습니다:

OptionParser.**print_version**(*file=None*)

현재 프로그램의 버전 메시지(`self.version`)를 *file*(기본값은 표준 출력)로 인쇄합니다. *print_usage()*와 마찬가지로, `self.version`에 등장하는 `%prog`는 현재 프로그램의 이름으로 대체됩니다. `self.version`이 비어 있거나 정의되지 않았으면 아무 작업도 수행하지 않습니다.

OptionParser.**get_version**()

*print_version()*과 같지만 인쇄하는 대신 버전 문자열을 반환합니다.

optparse가 에러를 처리하는 방법

*optparse*가 걱정해야 할 에러에는 두 가지가 있습니다: 프로그래머 에러와 사용자 에러. 프로그래머 에러는 일반적으로 *OptionParser.add_option()*에 대한 잘못된 호출입니다, 예를 들어 잘못된 옵션 문자열, 알 수 없는 옵션 어트리뷰트, 누락된 옵션 어트리뷰트 등. 이러한 에러는 일반적인 방식으로 처리됩니다: 예외(*optparse.OptionError*나 *TypeError*)를 발생시키고 프로그램이 충돌하도록 합니다.

사용자 에러를 처리하는 것은 훨씬 더 중요합니다, 코드가 아무리 안정적이더라도 발생하는 것이 보장되기 때문입니다. *optparse*는 잘못된 옵션 인자(`-n`이 정수 인자를 취할 때 `-n 4x` 전달), 누락된 인자(`-n`이 모든 형의 인자를 취할 때, 명령 줄 끝의 `-n`)와 같은 일부 사용자 에러를 자동으로 감지할 수 있습니다. 또한, `OptionParser.error()`를 호출하여 응용 프로그램 정의 에러 조건을 알릴 수 있습니다:

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

두 경우 모두, *optparse*는 같은 방식으로 에러를 처리합니다: 프로그램의 사용법 메시지와 에러 메시지를 표준 에러에 인쇄하고 에러 상태 2로 종료합니다.

사용자가 정수를 취하는 옵션에 `4x`를 전달하는 위의 첫 번째 예를 고려하십시오:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

또는 사용자가 값을 전혀 전달하지 못하는 경우:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

optparse-생성 에러 메시지는 항상 에러와 관련된 옵션을 언급하도록 주의를 기울입니다; 응용 프로그램 코드에서 `OptionParser.error()`를 호출할 때도 그래야 합니다.

*optparse*의 기본 에러 처리 동작이 여러분의 요구에 맞지 않으면, OptionParser를 서브 클래스 화하고 `exit()` 및/또는 `error()` 메서드를 재정의해야 합니다.

모두 합치기

일반적으로 *optparse* 기반 스크립트는 다음과 같습니다:

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print("reading %s..." % options.filename)
    ...

if __name__ == "__main__":
    main()
```

17.2.5 레퍼런스 지침서

구문 분석기 만들기

*optparse*를 사용하는 첫 번째 단계는 OptionParser 인스턴스를 만드는 것입니다.

class optparse.**OptionParser**(...)

OptionParser 생성자에는 필수 인자가 없지만, 여러 선택적 키워드 인자가 있습니다. 이들을 항상 키워드 인자로 전달해야 합니다, 즉, 인자가 선언된 순서에 의존하지 마십시오.

usage (기본값: `"%prog [options]"`)

프로그램이 잘못 실행되거나 도움말(help) 옵션을 사용할 때 인쇄할 사용법 요약. *optparse*가

사용법 문자열을 인쇄할 때, `%prog`를 `os.path.basename(sys.argv[0])`(또는 해당 키워드 인자를 전달한 경우 `prog`)으로 확장합니다. 사용법 메시지를 표시하지 않으려면, 특수 값 `optparse.SUPPRESS_USAGE`를 전달하십시오.

option_list (기본값: []）
구문 분석기를 채울 Option 객체 리스트. `option_list`의 옵션은 `standard_option_list`(OptionParser 서브 클래스에서 설정할 수 있는 클래스 어트리뷰트)의 모든 옵션 뒤에 추가되지만, 모든 버전(version) 또는 도움말(help) 옵션 앞에 추가됩니다. 폐지되었습니다; 대신 구문 분석기를 만든 후 *add_option()*을 사용하십시오.

option_class (기본값: optparse.Option)
*add_option()*에서 구문 분석기에 옵션을 추가할 때 사용할 클래스.

version (기본값: None)
사용자가 버전(version) 옵션을 제공할 때 인쇄 할 버전 문자열. `version`에 참값을 제공하면, *optparse*는 단일 옵션 문자열 `--version`으로 버전 옵션을 자동으로 추가합니다. 하위 문자열 `%prog`는 `usage`와 마찬가지로 확장됩니다.

conflict_handler (기본값: "error")
충돌하는 옵션 문자열이 있는 옵션이 구문 분석기에 추가될 때 수행 할 작업을 지정합니다; 섹션 옵션 간의 충돌을 참조하십시오.

description (기본값: None)
프로그램에 대한 간략한 개요를 제공하는 텍스트 단락. *optparse*는 현재 터미널 너비에 맞게 이 단락을 다시 포맷하고 사용자가 도움말을 요청할 때 인쇄합니다 (`usage` 이후, 옵션 목록 이전).

formatter (기본값: 새 IndentedHelpFormatter)
도움말 텍스트를 인쇄하는 데 사용될 optparse.HelpFormatter의 인스턴스. *optparse*는 이 목적으로 두 개의 구상 클래스를 제공합니다: IndentedHelpFormatter 와 TitledHelpFormatter.

add_help_option (기본값: True)
참이면, *optparse*는 구문 분석기에 도움말 옵션(옵션 문자열 `-h`와 `--help`)을 추가합니다.

prog
`usage`와 `version`에서 `%prog`를 확장할 때 `os.path.basename(sys.argv[0])` 대신 사용할 문자열.

epilog (기본값: None)
옵션 도움말 다음에 인쇄할 도움말 텍스트 단락.

구문 분석기 채우기

구문 분석기를 옵션으로 채우는 방법에는 여러 가지가 있습니다. 선호되는 방법은 섹션 자습서에 표시된 대로, *OptionParser.add_option()*을 사용하는 것입니다. `add_option()`은 다음 두 가지 방법의 하나로 호출 할 수 있습니다:

- (`make_option()`에서 반환되는 것과 같은) Option 인스턴스를 전달합니다
- `make_option()`(즉, Option 생성자)에 허용되는 위치와 키워드 인자의 조합을 전달합니다, 그러면 Option 인스턴스를 만듭니다

다른 대안은 다음과 같이 미리 생성된 Option 인스턴스 리스트를 OptionParser 생성자에 전달하는 것입니다:

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
    ]
parser = OptionParser(option_list=option_list)
```

(`make_option()`은 Option 인스턴스를 만들기 위한 팩토리 함수입니다; 현재는 Option 생성자의 별칭입니다. *optparse*의 향후 버전은 Option을 여러 클래스로 나눌 수 있으며, `make_option()`은 인스턴스 화할 올바른 클래스를 선택할 것입니다. Option을 직접 인스턴스 화하지 마십시오.)

옵션 정의하기

각 Option 인스턴스는 동의어 명령 줄 옵션 문자열 집합을 나타냅니다, 예를 들어 `-f`와 `--file`. 짧거나 긴 옵션 문자열을 얼마든지 지정할 수 있지만, 전체적으로 옵션 문자열을 적어도 하나 지정해야 합니다.

Option 인스턴스를 만드는 규범적 방법은 *OptionParser*의 `add_option()` 메서드를 사용하는 것입니다.

`OptionParser.`**`add_option`**(*option*)

`OptionParser.`**`add_option`**(**opt_str, attr=value, ...*)

짧은 옵션 문자열로만 옵션을 정의하려면:

```
parser.add_option("-f", attr=value, ...)
```

그리고 긴 옵션 문자열로만 옵션을 정의하려면:

```
parser.add_option("--foo", attr=value, ...)
```

키워드 인자는 새 Option 객체의 어트리뷰트를 정의합니다. 가장 중요한 옵션 어트리뷰트는 *action*이며, 전체적으로 어떤 어트리뷰트가 관련성이 있거나 필요한지를 결정합니다. 관련 없는 옵션 어트리뷰트를 전달하거나, 필수 어트리뷰트를 전달하지 못하면, *optparse*는 실수를 설명하는 *OptionError* 예외를 발생시킵니다.

옵션의 *action*은 명령 줄에서 이 옵션을 만날 때 *optparse*가 수행하는 작업을 결정합니다. *optparse*에 하드 코딩된 표준 옵션 액션은 다음과 같습니다:

"store"
이 옵션의 인자를 저장합니다 (기본값)

"store_const"
store a constant value, pre-set via *Option.const*

"store_true"
`True`를 저장합니다

"store_false"
`False`를 저장합니다

"append"
이 옵션의 인자를 리스트에 추가합니다

"append_const"
리스트에 상숫값을 추가합니다, *Option.const*로 사전 설정

"count"
카운터를 1씩 증가시킵니다

"callback"
지정된 함수를 호출합니다

"help"
모든 옵션과 해당 설명을 포함하는 사용법 메시지를 인쇄합니다

(액션을 제공하지 않으면, 기본값은 `"store"`입니다. 이 액션의 경우, *type*과 *dest* 옵션 어트리뷰트도 제공할 수 있습니다; 표준 옵션 액션을 참조하십시오.)

보시다시피, 대부분의 액션은 값을 어딘가에 저장하거나 갱신하는 것을 수반합니다. *optparse*는 항상 이를 위해 일반적으로 `options`(*optparse.Values*의 인스턴스)라고 하는 특수 객체를 만듭니다.

class optparse.**Values**

An object holding parsed argument names and values as attributes. Normally created by calling when calling *OptionParser.parse_args()*, and can be overridden by a custom subclass passed to the *values* argument of *OptionParser.parse_args()* (as described in 인자 구문 분석하기).

옵션 인자(및 기타 다양한 값)는 *dest* (destination) 옵션 어트리뷰트에 따라, 이 객체의 어트리뷰트로 저장됩니다.

예를 들어, 다음과 같이 호출할 때

```
parser.parse_args()
```

*optparse*가 하는 첫 번째 작업 중 하나는 `options` 객체를 만드는 것입니다:

```
options = Values()
```

이 구문 분석기의 옵션 중 하나가 다음과 같이 정의되었으면:

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

그리고 구문 분석 중인 명령 줄에는 다음 중 하나가 포함되면:

```
-ffoo
-f foo
--file=foo
--file foo
```

*optparse*는 이 옵션을 볼 때 다음과 동등한 일을 합니다

```
options.filename = "foo"
```

*type*과 *dest* 옵션 어트리뷰트는 *action*만큼 중요하지만, *action*은 모든 옵션에 적합한 유일한 어트리뷰트입니다.

옵션 어트리뷰트

class optparse.**Option**

A single command line argument, with various attributes passed by keyword to the constructor. Normally created with *OptionParser.add_option()* rather than directly, and can be overridden by a custom class via the *option_class* argument to *OptionParser*.

다음 옵션 어트리뷰트는 *OptionParser.add_option()*에 키워드 인자로 전달될 수 있습니다. 특정 옵션과 관련이 없는 옵션 어트리뷰트를 전달하거나, 필수 옵션 어트리뷰트를 전달하지 못하면 *optparse*는 *OptionError*를 발생시킵니다.

Option.**action**

(기본값: `"store"`)

이 옵션이 명령 줄에서 보일 때 *optparse*의 동작을 결정합니다; 사용 가능한 옵션은 여기에 설명되어 있습니다.

Option.**type**

(기본값: `"string"`)

이 옵션이 기대하는 인자 형 (예를 들어, `"string"`이나 `"int"`); 사용 가능한 옵션 형은 여기에 설명되어 있습니다.

Option.**dest**

(기본값: 옵션 문자열에서 파생됩니다)

옵션의 액션이 어딘가에 값을 쓰거나 수정하는 것을 의미하면, 이것은 *optparse*에게 어디에 쓸 것인지 알려줍니다: *dest*는 명령 줄을 구문 분석할 때 *optparse*가 빌드하는 `options` 객체의 어트리뷰트의 이름을 정합니다.

Option.**default**

옵션이 명령 줄에서 보이지 않으면 이 옵션의 대상에 사용할 값. *OptionParser.set_defaults()*도 참조하십시오.

Option.**nargs**

(기본값: 1)

이 옵션이 보일 때 소비되어야 하는 *type* 형의 인자 수입니다. > 1 이면, *optparse*는 값의 튜플을 *dest*에 저장합니다.

Option.**const**

상숫값을 저장하는 액션의 경우, 저장할 상숫값.

Option.**choices**

"choice" 형 옵션의 경우, 사용자가 이 중에서 선택할 수 있는 문자열 리스트.

Option.**callback**

액션 "callback"이 있는 옵션의 경우, 이 옵션이 보일 때 호출 할 콜러블. 콜러블에 전달된 인자에 대한 자세한 내용은 섹션 옵션 콜백을 참조하십시오.

Option.**callback_args**

Option.**callback_kwargs**

4개의 표준 콜백 인자 다음에 callback에 전달할 추가 위치와 키워드 인자.

Option.**help**

사용자가 *help* 옵션을 제공한 (가령 --help) 후 사용 가능한 모든 옵션을 나열할 때 이 옵션에 대해 인쇄할 도움말 텍스트. 도움말 텍스트가 제공되지 않으면, 옵션이 도움말 텍스트 없이 나열됩니다. 이 옵션을 숨기려면, 특수 값 optparse.SUPPRESS_HELP를 사용하십시오.

Option.**metavar**

(기본값: 옵션 문자열에서 파생됩니다)

도움말 텍스트를 인쇄할 때 사용할 옵션 인자를 나타냅니다. 예제는 섹션 자습서를 참조하십시오.

표준 옵션 액션

다양한 옵션 액션은 모두 요구 사항과 효과가 약간 다릅니다. 대부분의 액션에는 *optparse*의 동작을 안내하기 위해 지정할 수 있는 몇 가지 연관된 옵션 어트리뷰트가 있습니다; 일부는 해당 액션을 사용하는 모든 옵션에 대해 지정해야 하는 필수 어트리뷰트가 있습니다.

- "store" [연관된 옵션: *type*, *dest*, *nargs*, *choices*]

  옵션 뒤에 인자가 와야 하며, 인자는 *type*에 따라 값으로 변환되고, *dest*에 저장됩니다. *nargs* > 1 이면, 명령 줄에서 여러 인자가 소비됩니다; 모두 *type*에 따라 변환되고 *dest*에 튜플로 저장됩니다. 표준 옵션 형 섹션을 참조하십시오.

  *choices*가 제공되면 (문자열 리스트나 튜플), 형의 기본값은 "choice"입니다.

  *type*이 제공되지 않으면, 기본값은 "string"입니다.

  *dest*가 제공되지 않으면, *optparse*는 첫 번째 긴 옵션 문자열에서 대상을 파생합니다 (예를 들어, --foo-bar는 foo_bar를 암시합니다). 긴 옵션 문자열이 없으면, *optparse*는 첫 번째 짧은 옵션 문자열에서 대상을 파생합니다 (예를 들어, -f는 f를 암시합니다).

  예:

  ```
  parser.add_option("-f")
  parser.add_option("-p", type="float", nargs=3, dest="point")
  ```

  다음과 같은 명령 줄을 구문 분석할 때

  ```
  -f foo.txt -p 1 -3.5 4 -fbar.txt
  ```

  *optparse*는 다음과 같이 설정합니다

  ```
  options.f = "foo.txt"
  options.point = (1.0, -3.5, 4.0)
  options.f = "bar.txt"
  ```

- "store_const" [필수 옵션: *const*; 연관된 옵션: *dest*]

  값 *const*는 *dest*에 저장됩니다.

  예:

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
```

  --noisy가 보이면, *optparse*는 다음과 같이 설정합니다

```
options.verbose = 2
```

- "store_true" [연관된 옵션: *dest*]

  True를 *dest*에 저장하는 "store_const"의 특별한 경우.

- "store_false" [연관된 옵션: *dest*]

  "store_true"와 비슷하지만, False를 저장합니다.

  예:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append" [연관된 옵션: *type*, *dest*, *nargs*, *choices*]

  옵션 다음에는 *dest*의 리스트에 추가되는 인자가 와야 합니다. *dest*의 기본값이 제공되지 않으면, *optparse*가 명령 줄에서 이 옵션을 처음 발견할 때 빈 리스트가 자동으로 만들어집니다. *nargs* > 1 이면, 여러 인자가 소비되며, *nargs* 길이의 튜플이 *dest*에 추가됩니다.

  *type*과 *dest*의 기본값은 "store" 액션의 경우와 같습니다.

  예:

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

  -t3가 명령 줄에 보이면, *optparse*는 다음과 동등한 것을 수행합니다:

```
options.tracks = []
options.tracks.append(int("3"))
```

  잠시 후, --tracks=4가 보이면, 다음을 수행합니다:

```
options.tracks.append(int("4"))
```

  append 액션은 옵션의 현재 값에 대해 append 메서드를 호출합니다. 이는 지정된 모든 기본값에 append 메서드가 있어야 함을 의미합니다. 또한, 기본값이 비어 있지 않으면, 기본 요소가 옵션의 구문 분석된 값에 존재하며, 명령 줄의 모든 값이 기본값 뒤에 추가됨을 의미합니다:

```
>>> parser.add_option("--files", action="append", default=['~/.mypkg/defaults
↪'])
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
>>> opts.files
['~/.mypkg/defaults', 'overrides.mypkg']
```

- "append_const" [필수 옵션: *const*; 연관된 옵션: *dest*]

  "store_const"와 비슷하지만, *const* 값이 *dest*에 추가됩니다; "append"와 마찬가지로, *dest*의 기본값은 None이며, 옵션이 처음 발견될 때 빈 리스트가 자동으로 만들어집니다.

- "count" [연관된 옵션: *dest*]

  *dest*에 저장된 정수를 증가시킵니다. 기본값이 제공되지 않으면, *dest*는 처음으로 증가하기 전에 0으로 설정됩니다.

  예:

```
parser.add_option("-v", action="count", dest="verbosity")
```

  -v가 명령 줄에서 처음 보이면, *optparse*는 다음과 동등한 작업을 수행합니다:

```
options.verbosity = 0
options.verbosity += 1
```

  이후에 -v가 나타날 때마다 다음과 같이 합니다

```
options.verbosity += 1
```

- "callback" [필수 옵션: *callback*; 연관된 옵션: *type*, *nargs*, *callback_args*, *callback_kwargs*]

  *callback*으로 지정된 함수를 호출합니다, 이 함수는 다음과 같이 호출됩니다

```
func(option, opt_str, value, parser, *args, **kwargs)
```

  자세한 내용은 섹션 옵션 콜백을 참조하십시오.

- "help"

  현재 옵션 구문 분석기의 모든 옵션에 대한 전체 도움말 메시지를 인쇄합니다. 도움말 메시지는 OptionParser의 생성자에 전달된 usage 문자열과 모든 옵션에 전달된 *help* 문자열로 구성됩니다.

  옵션에 *help* 문자열이 제공되지 않아도, 도움말 메시지에 나열됩니다. 옵션을 완전히 생략하려면, 특수 값 optparse.SUPPRESS_HELP를 사용하십시오.

  *optparse*는 모든 OptionParser에 *help* 옵션을 자동으로 추가하므로, 일반적으로 만들 필요가 없습니다.

  예:

```
from optparse import OptionParser, SUPPRESS_HELP

# 일반적으로, 도움말 옵션은 자동으로 추가되지만, add_help_option 인자를
# 사용하여 억제할 수 있습니다
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

  *optparse*가 명령 줄에서 -h나 --help를 보면, 다음 도움말 메시지와 같은 내용을 stdout에 인쇄합니다 (sys.argv[0]이 "foo.py"라고 가정합니다):

```
Usage: foo.py [options]

Options:
  -h, --help        Show this help message and exit
  -v                Be moderately verbose
  --file=FILENAME   Input file to read data from
```

  도움말 메시지를 인쇄한 후, *optparse*는 sys.exit(0)으로 프로세스를 종료합니다.

- "version"

  OptionParser에 제공된 버전 번호를 stdout에 인쇄하고 종료합니다. 버전 번호는 실제로 OptionParser의 print_version() 메서드에 의해 포맷되고 인쇄됩니다. 일반적으로 version 인자가 OptionParser 생성자에 제공되는 경우에만 의미가 있습니다. help 옵션과 마찬가지로, optparse는 필요할 때 자동으로 추가하기 때문에, version 옵션을 거의 만들지 않습니다.

표준 옵션 형

optparse에는 다섯 가지 내장 옵션 형이 있습니다: "string", "int", "choice", "float" 및 "complex". 새로운 옵션 형을 추가해야 하면, 섹션 *optparse 확장하기*를 참조하십시오.

문자열 옵션에 대한 인자는 어떤 방식으로도 검사되거나 변환되지 않습니다: 명령 줄의 텍스트는 있는 그대로 대상에 저장(또는 콜백에 전달)됩니다.

정수 인자("int" 형)는 다음과 같이 구문 분석됩니다:

- 숫자가 0x로 시작하면, 16진수로 구문 분석됩니다
- 숫자가 0으로 시작하면, 8진수로 구문 분석됩니다
- 숫자가 0b로 시작하면, 이진수로 구문 분석됩니다
- 그렇지 않으면, 숫자는 10진수로 구문 분석됩니다

변환은 적절한 진수(2, 8, 10 또는 16)로 int()를 호출하여 수행됩니다. 이것이 실패하면, 더 유용한 에러 메시지를 제공하기는 하지만, optparse도 마찬가지로 실패합니다.

"float"와 "complex" 옵션 인자는 유사한 에러 처리로 float()와 complex()로 직접 변환됩니다.

"choice" 옵션은 "string" 옵션의 서브 형입니다. choices 옵션 어트리뷰트(문자열 시퀀스)는 허용되는 옵션 인자 집합을 정의합니다. optparse.check_choice()는 사용자가 제공한 옵션 인자를 이 마스터 리스트와 비교하고 유효하지 않은 문자열이 주어지면 OptionValueError를 발생시킵니다.

인자 구문 분석하기

OptionParser를 만들고 채우는 것의 요점은 parse_args() 메서드를 호출하는 것입니다.

OptionParser.**parse_args**(*args=None*, *values=None*)

Parse the command-line options found in *args*.

입력 매개 변수는

args
처리할 인자의 리스트 (기본값: sys.argv[1:])

values
옵션 인자를 저장할 Values 객체 (기본값: Values의 새 인스턴스) – 기존 객체를 제공하면, 옵션 기본값이 초기화되지 않습니다.

그리고 반환 값은 쌍 (options, args) 이고

options
*values*로 전달된 것과 같은 객체, 또는 optparse가 만든 optparse.Values 인스턴스

args
모든 옵션이 처리된 후 남은 위치 인자

가장 일반적인 사용법은 두 키워드 인자 중 어느 것도 제공하지 않는 것입니다. values를 제공하면, 반복된 setattr() 호출(옵션 대상에 저장된 모든 옵션 인자에 대해 대략 하나씩)로 수정되고, parse_args()에서 반환됩니다.

parse_args()가 인자 리스트에서 에러를 만나면, 적절한 최종 사용자 에러 메시지와 함께 OptionParser의 error() 메서드를 호출합니다. 이는 궁극적으로 종료 상태 2(명령 줄 에러에 대한 전통적인 유닉스 종료 상태)로 프로세스를 종료합니다.

옵션 구문 분석기를 조회하고 조작하기

옵션 구문 분석기의 기본 동작은 약간 사용자 정의 할 수 있으며, 옵션 구문 분석기를 들여다보고 거기에 무엇이 있는지 볼 수도 있습니다. OptionParser는 다음과 같은 몇 가지 메서드를 제공합니다:

OptionParser.**disable_interspersed_args**()

첫 번째 옵션이 아닌 것에서 중지하도록 구문 분석을 설정합니다. 예를 들어 -a와 -b가 모두 인자를 취하지 않는 간단한 옵션이면, *optparse*는 일반적으로 다음 문법을 허용합니다:

```
prog -a arg1 -b arg2
```

그리고 다음과 동등하게 취급합니다

```
prog -a -b arg1 arg2
```

이 기능을 비활성화하려면, *disable_interspersed_args()*를 호출하십시오. 이렇게 하면 옵션 구문 분석이 첫 번째 옵션이 아닌 인자에서 중지되는, 전통적인 유닉스 문법이 복원됩니다.

자체 옵션이 있는 다른 명령을 실행하는 명령 프로세서가 있고 이러한 옵션이 혼동되지 않도록 하려면 이것을 사용하십시오. 예를 들어, 각 명령에는 다른 옵션 집합이 있을 수 있습니다.

OptionParser.**enable_interspersed_args**()

첫 번째 옵션이 아닌 것에서 구문 분석이 중지되지 않도록 설정하여, 명령 인자와 스위치를 분산시킬 수 있도록 합니다. 이것이 기본 동작입니다.

OptionParser.**get_option**(*opt_str*)

옵션 문자열 *opt_str*을 갖는 Option 인스턴스를 반환합니다, 또는 해당 옵션 문자열을 갖는 옵션이 없으면 None을 반환합니다.

OptionParser.**has_option**(*opt_str*)

OptionParser에 옵션 문자열 *opt_str*을 갖는 옵션이 있으면 True를 반환합니다 (예를 들어, -q나 --verbose).

OptionParser.**remove_option**(*opt_str*)

*OptionParser*에 *opt_str*에 해당하는 옵션이 있으면, 해당 옵션이 제거됩니다. 해당 옵션이 다른 옵션 문자열을 제공하면, 해당 옵션 문자열은 모두 유효하지 않게 됩니다. 이 *OptionParser*에 속하는 옵션에서 *opt_str*이 등장하지 않으면, *ValueError*를 발생시킵니다.

옵션 간의 충돌

주의하지 않으면, 충돌하는 옵션 문자열을 갖는 옵션을 정의하기 쉽습니다:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(일부 표준 옵션을 사용하여 자체 OptionParser 서브 클래스를 정의한 경우 특히 그렇습니다.)

옵션을 추가할 때마다, *optparse*는 기존 옵션과의 충돌을 확인합니다. 발견되면, 현재 충돌 처리 메커니즘을 호출합니다. 충돌 처리 메커니즘을 설정할 수 있는데, 생성자에서:

```
parser = OptionParser(..., conflict_handler=handler)
```

또는 별도의 호출로 가능합니다:

```
parser.set_conflict_handler(handler)
```

사용 가능한 충돌 처리기는 다음과 같습니다:

"error" (기본값)

옵션 충돌이 프로그래밍 에러라고 가정하고 *OptionConflictError*를 발생시킵니다

"resolve"
옵션 충돌을 지능적으로 해결합니다 (아래를 참조하십시오)

예를 들어, 충돌을 지능적으로 해결하고 *OptionParser*를 정의하고 충돌하는 옵션을 추가해 보겠습니다:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

이 시점에서, *optparse*는 이전에 추가된 옵션이 이미 `-n` 옵션 문자열을 사용하고 있음을 감지합니다. `conflict_handler`가 `"resolve"`이므로, 이전 옵션의 옵션 문자열 리스트에서 `-n`을 제거하여 상황을 해결합니다. 이제 `--dry-run`은 사용자가 해당 옵션을 활성화하는 유일한 방법입니다. 사용자가 help를 요청하면, 도움말 메시지에 이것이 반영됩니다:

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

남아있는 옵션이 없을 때까지 이전에 추가된 옵션에 대한 옵션 문자열을 제거 할 수 있고, 사용자는 명령 줄에서 해당 옵션을 호출할 방법이 없을 가능성이 있습니다. 이 경우, *optparse*는 해당 옵션을 완전히 제거해서, 도움말 텍스트나 다른 곳에 표시되지 않습니다. 기존 OptionParser를 계속 사용해서:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

이 시점에서, 원래 `-n/--dry-run` 옵션에 더는 액세스할 수 없어서, *optparse*는 해당 옵션을 제거하고, 다음 도움말 텍스트를 남깁니다:

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

정리

OptionParser 인스턴스에는 여러 순환 참조가 있습니다. 이것은 파이썬의 가비지 수거기에선 문제가 되지 않지만, 작업이 끝나면 OptionParser에서 `destroy()`를 호출하여 순환 참조를 명시적으로 끊을 수 있습니다. 이것은 OptionParser에서 큰 객체 그래프에 도달할 수 있는 장기 실행 응용 프로그램에서 특히 유용합니다.

기타 메서드

OptionParser는 몇 가지 다른 공용 메서드를 지원합니다:

`OptionParser.`**`set_usage`**(*usage*)

`usage` 생성자 키워드 인자에 대해 위에서 설명한 규칙에 따라 사용법 문자열을 설정합니다. `None`을 전달하면 기본 사용법 문자열이 설정됩니다; 사용법 메시지를 억제하려면 `optparse.SUPPRESS_USAGE`를 사용하십시오.

`OptionParser.`**`print_usage`**(*file=None*)

현재 프로그램의 사용법 메시지(`self.usage`)를 *file*(기본값 stdout)로 인쇄합니다. `self.usage`에 등장하는 문자열 `%prog`는 현재 프로그램의 이름으로 대체됩니다. `self.usage`가 비어 있거나 정의되지 않았으면 아무 작업도 수행하지 않습니다.

`OptionParser.`**`get_usage`**()

*print_usage()*와 같지만 인쇄하는 대신 사용법 문자열을 반환합니다.

`OptionParser.`**`set_defaults`**(*dest=value, ...*)

한 번에 여러 옵션 대상에 대한 기본값을 설정합니다. 여러 옵션이 같은 대상을 공유 할 수 있기 때문에, *set_defaults()*를 사용하여 옵션의 기본값을 설정하는 것이 좋습니다. 예를 들어, 여러 "mode" 옵션이 모두 같은 대상을 설정하면, 그중 하나가 기본값을 설정할 수 있으며 마지막 옵션이 이깁니다:

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")     # 아래에서 재정의됩니다
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")   # 위의 설정을 재정의합니다
```

이러한 혼동을 피하려면, *set_defaults()*를 사용하십시오:

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

17.2.6 옵션 콜백

*optparse*의 내장 액션과 형이 여러분의 필요에 충분하지 않으면, 두 가지 선택이 있습니다: *optparse*를 확장하거나 콜백 옵션을 정의합니다. *optparse*를 확장하는 것이 더 일반적이지만, 많은 간단한 경우에는 과합니다. 종종 간단한 콜백만 있으면 됩니다.

콜백 옵션을 정의하는 두 단계가 있습니다:

- `"callback"` 액션을 사용하여 옵션 자체를 정의합니다
- 콜백을 작성합니다; 이것은 아래에 설명된 대로, 최소한 4개의 인자를 취하는 함수(또는 메서드)입니다

콜백 옵션 정의하기

항상 그렇듯이, 콜백 옵션을 정의하는 가장 쉬운 방법은 *OptionParser.add_option()* 메서드를 사용하는 것입니다. *action*과 별도로 지정해야 하는 유일한 옵션 어트리뷰트는 호출할 함수인 `callback`입니다:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

`callback`은 함수(또는 다른 콜러블 객체)이므로, 이 콜백 옵션을 만들 때 `my_callback()`을 이미 정의했어야 합니다. 이 간단한 경우, *optparse*는 `-c`가 인자를 취하는지조차 알지 못합니다. 이는 일반적으로 옵션이 인자를 받지 않음을 의미합니다—명령 줄에 `-c`가 있다는 것만 알 필요가 있습니다. 그러나 일부 상황에서는, 콜백이 임의의 수의 명령 줄 인자를 소비하도록 할 수 있습니다. 콜백 작성이 까다로워지는 곳입니다: 이 섹션의 뒷부분에서 다룹니다.

*optparse*는 항상 4개의 특정 인자를 콜백에 전달하며, *callback_args*와 *callback_kwargs*를 통해 지정하는 경우에만 추가 인자를 전달합니다. 따라서 최소 콜백 함수 서명은 다음과 같습니다:

```
def my_callback(option, opt, value, parser):
```

콜백에 대한 네 가지 인자가 아래에 설명되어 있습니다.

콜백 옵션을 정의할 때 제공할 수 있는 몇 가지 다른 옵션 어트리뷰트가 있습니다:

type
일반적인 의미를 갖습니다: `"store"`나 `"append"` 액션과 마찬가지로, *optparse*에게 하나의 인자를 소비하고 이를 *type*으로 변환하도록 지시합니다. 그러나 *optparse*는 변환된 값을 어딘가에 저장하는 대신 콜백 함수에 전달합니다.

nargs
역시 일반적인 의미를 갖습니다: 제공되고 > 1 이면, *optparse*는 *nargs* 인자를 소비하며 각 인자는 *type*으로 변환 가능해야 합니다. 그런 다음 변환된 값의 튜플을 콜백에 전달합니다.

callback_args
콜백에 전달할 추가 위치 인자의 튜플

callback_kwargs
콜백에 전달할 추가 키워드 인자의 딕셔너리

콜백이 호출되는 방법

모든 콜백은 다음과 같이 호출됩니다:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

여기서

option
콜백을 호출하는 Option 인스턴스입니다

opt_str
콜백을 트리거 하는 명령 줄에 나타나는 옵션 문자열입니다. (축약된 긴 옵션이 사용되면, `opt_str`은 완전한, 규범적인 옵션 문자열이 됩니다—예를 들어 사용자가 `--foobar`의 약어로 명령 줄에 `--foo`를 입력하면, `opt_str`은 `"--foobar"`가 됩니다.)

value
명령 줄에 나타난 이 옵션에 대한 인자입니다. *optparse*는 *type*이 설정되었을 때만 인자를 기대합니다; `value`의 형은 옵션의 형이 암시하는 형입니다. 이 옵션의 *type*이 `None`(기대하는 인자 없음)이면, `value`는 `None`이 됩니다. *nargs* > 1 이면, `value`는 적절한 형의 값의 튜플이 됩니다.

parser
모든 것을 구동하는 OptionParser 인스턴스입니다, 인스턴스 어트리뷰트를 통해 다른 흥미로운 데이터에 액세스 할 수 있어서 주로 유용합니다:

parser.largs
남은 인자의 현재 리스트, 즉, 소비되었지만 옵션이나 옵션 인자가 아닌 인자. 예를 들어 더 많은 인자를 추가하여, `parser.largs`를 자유롭게 수정하십시오. (이 리스트는 *parse_args()*의 두 번째 반환 값인 `args`가 됩니다.)

parser.rargs
나머지 인자의 현재 리스트, 즉, `opt_str`과 `value`(해당한다면)가 제거되고, 그 뒤에 오는 인자만 그대로 남아 있습니다. 예를 들어 더 많은 인자를 소비하여, `parser.rargs`를 자유롭게 수정하십시오.

parser.values
옵션값이 기본적으로 저장되는 객체 (optparse.OptionValues의 인스턴스). 이를 통해 콜백은 옵션값을 저장하기 위해 나머지 *optparse*와 같은 메커니즘을 사용할 수 있습니다; 전역이나 클로저를 엉망으로 만들 필요가 없습니다. 명령 줄에서 이미 발견된 모든 옵션의 값에 액세스하거나 수정할 수도 있습니다.

args
callback_args 옵션 어트리뷰트를 통해 제공되는 임의의 위치 인자의 튜플입니다.

kwargs
*callback_kwargs*를 통해 제공된 임의의 키워드 인자의 딕셔너리입니다.

콜백에서 에러 발생시키기

옵션이나 인자에 문제가 있으면 콜백 함수는 *OptionValueError*를 발생시켜야 합니다. *optparse*는 이것을 포착하고 프로그램을 종료하고, stderr에 여러분이 제공하는 에러 메시지를 인쇄합니다. 메시지는 명확하고 간결하며 정확해야 하며 잘못된 옵션을 언급해야 합니다. 그렇지 않으면, 사용자는 자신이 무엇을 잘못했는지 파악하는 데 어려움을 겪을 것입니다.

콜백 예제 1: 간단한 콜백

다음은 인자를 취하지 않고, 단순히 옵션이 발견되었음을 기록하는 콜백 옵션의 예입니다:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

물론 "store_true" 액션으로 그렇게 할 수 있습니다.

콜백 예제 2: 옵션 순서 확인

여기에 약간 더 흥미로운 예가 있습니다: -a가 발견되었다는 사실을 기록하지만, 명령 줄에서 -b 뒤에 오면 폭발합니다.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

콜백 예제 3: 옵션 순서 확인 (일반화)

이 콜백을 몇 가지 유사한 옵션에 다시 사용하려면 (플래그를 설정하지만, -b가 이미 보였으면 폭발함), 약간의 작업이 필요합니다: 에러 메시지와 설정하는 플래그를 일반화해야 합니다.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

콜백 예제 4: 임의 조건 확인

물론, 여기에 어떤 조건도 넣을 수 있습니다—이미 정의된 옵션의 값을 확인하는 데 국한되지 않습니다. 예를 들어, 만월일 때 호출해서는 안 되는 옵션이 있으면, 다음과 같이 하면 됩니다:

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(is_moon_full()의 정의는 독자를 위한 연습 문제로 남겨 둡니다.)

콜백 예제 5: 고정 인자

고정된 수의 인자를 사용하는 콜백 옵션을 정의하면 상황이 약간 더 흥미로워집니다. 콜백 옵션이 인자를 받도록 지정하는 것은 "store"나 "append" 옵션을 정의하는 것과 유사합니다: *type*을 정의하면, 옵션은 해당 형으로 변환 할 수 있어야 하는 하나의 인자를 취합니다; *nargs*를 추가로 정의하면, 옵션은 *nargs* 인자를 취합니다.

다음은 표준 "store" 액션을 흉내 내는 예입니다:

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

*optparse*가 3개의 인자를 소비하고 이를 정수로 변환하는 작업을 처리함에 유의하십시오; 여러분이 해야 할 일은 그것들을 저장하는 것뿐입니다. (또는 무엇이든; 분명히 이 예제에서는 콜백이 필요하지 않습니다.)

콜백 예제 6: 가변 인자

가변적인 수의 인자를 취하는 옵션을 원할 때 상황이 복잡해집니다. 이 경우, *optparse*는 이것을 위한 내장 기능을 제공하지 않아서 콜백을 작성해야 합니다. 그리고 *optparse*가 일반적으로 처리하는 전통적인 유닉스 명령 줄 구문 분석의 복잡한 문제를 여러분이 처리해야 합니다. 특히, 콜백은 날(bare) `--`와 `-` 인자에 대한 전통적인 규칙을 구현해야 합니다:

- `--`나 `-`는 옵션 인자가 될 수 있습니다
- 날 `--` (어떤 옵션에 대한 인자가 아닌 경우): 명령 줄 처리를 중단하고 `--`를 버립니다
- 날 `-` (어떤 옵션에 대한 인자가 아닌 경우): 명령 줄 처리를 중지하지만 `-`는 유지합니다 (`parser.largs`에 추가합니다)

가변적인 수의 인자를 취하는 옵션을 원한다면, 몇 가지 미묘하고 까다로운 문제가 있습니다. 여러분이 선택하는 정확한 구현은 여러분이 여러분의 응용 프로그램을 위해 취하고자 하는 절충을 기반으로 할 것입니다 (이것이 *optparse*가 이런 종류의 것을 직접 지원하지 않는 이유입니다).

그런데도, 가변 인자가 있는 옵션에 대한 콜백에는 가시가 있습니다:

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
        # --foo 와 같은 옵션에서 멈춥니다
        if arg[:2] == "--" and len(arg) > 2:
            break
        # -a 에서 멈추지만, -3 이나 -3.0 에서는 멈추지 않습니다
        if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
            break
        value.append(arg)

    del parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)
```

17.2.7 optparse 확장하기

*optparse*가 명령 줄 옵션을 해석하는 방법의 두 가지 주요 제어 요소는 각 옵션의 액션과 형이므로, 확장 방향은 새 액션과 새 형을 추가하는 것입니다.

새로운 형 추가하기

새로운 형을 추가하려면, *optparse*의 *Option* 클래스의 여러분 자신의 서브 클래스를 정의해야 합니다. 이 클래스에는 *optparse*의 형을 정의하는 몇 가지 어트리뷰트가 있습니다: *TYPES*와 *TYPE_CHECKER*.

Option.**TYPES**

형 이름의 튜플; 여러분의 서브 클래스에서, 표준 튜플을 기반으로 하는 새 튜플 *TYPES*를 정의하십시오.

Option.**TYPE_CHECKER**

형 이름을 형 검사 함수에 매핑하는 딕셔너리. 형 검사 함수는 다음과 같은 서명을 갖습니다:

```
def check_mytype(option, opt, value)
```

여기서 `option`은 *Option* 인스턴스이고, `opt`는 옵션 문자열(예를 들어, `-f`)이며, `value`는 검사되고 원하는 형으로 변환되어야 하는 명령 줄의 문자열입니다. `check_mytype()`은 가상의 형 `mytype`의 객체를 반환해야 합니다. 형 검사 함수가 반환한 값은 *OptionParser.parse_args()*에서 반환한 OptionValues 인스턴스에 포함되거나 `value` 매개 변수로 콜백에 전달됩니다.

형 검사 함수는 문제가 발생하면 *OptionValueError*를 발생시켜야 합니다. *OptionValueError*는 *OptionParser*의 `error()` 메서드에 있는 그대로 전달되는 단일 문자열 인자를 취하며, 이는 차례로 프로그램 이름과 문자열 `"error:"`를 앞에 붙이고 프로세스를 종료하기 전에 모든 것을 stderr에 인쇄합니다.

다음은 명령 줄에서 파이썬 스타일 복소수를 구문 분석하기 위해 `"complex"` 옵션 형을 추가하는 것을 보여주는 우스꽝스러운 예입니다. (*optparse* 1.3은 복소수에 대한 기본 지원을 추가했기 때문에 전보다 훨씬 우스꽝스럽지만, 신경 쓰지 마십시오.)

첫째, 필요한 임포트:

```
from copy import copy
from optparse import Option, OptionValueError
```

나중에 (Option 서브 클래스의 *TYPE_CHECKER* 클래스 어트리뷰트에서) 참조되므로, 형 검사기를 먼저 정의해야 합니다:

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

마지막으로, Option 서브 클래스:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(*Option.TYPE_CHECKER*의 *copy()*를 만들지 않았다면, *optparse*의 Option 클래스의 *TYPE_CHECKER* 어트리뷰트를 수정하게 될 것입니다. 이것은 파이썬이기 때문에, 좋은 태도와 상식을 제외하고는 아무것도 이렇게 하는 것을 막을 수 없습니다.)

이것이 전부입니다! 이제 OptionParser가 Option 대신 MyOption을 사용하도록 지시해야 한다는 점을 제외하고는, 다른 *optparse* 기반 스크립트와 마찬가지로 새 옵션 형을 사용하는 스크립트를 작성할 수 있습니다:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

또는, 여러분 자신만의 옵션 목록을 만들어 OptionParser에 전달할 수 있습니다; 위의 방법으로 `add_option()`을 사용하지 않으면, OptionParser에 사용할 옵션 클래스를 알려줄 필요가 없습니다:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

새로운 액션 추가하기

*optparse*에는 액션에 대한 몇 가지 분류가 있음을 이해해야 해서, 새 액션을 추가하는 것은 약간 까다롭습니다:

"저장" 액션
: *optparse*가 현재 OptionValues 인스턴스의 어트리뷰트에 값을 저장하는 결과를 주는 액션; 이러한 옵션을 사용하려면 Option 생성자에 *dest* 어트리뷰트를 제공해야 합니다.

"형이 있는" 액션
: 명령 줄에서 값을 취하고 이것이 특정 형일 것으로 기대하는 액션; 또는, 특정 형으로 변환할 수 있는 문자열. 이러한 옵션을 사용하려면 Option 생성자에 *type* 어트리뷰트를 제공해야 합니다.

이들은 겹치는 집합입니다: 일부 기본 "저장" 액션은 `"store"`, `"store_const"`, `"append"` 및 `"count"`이고, 기본 "형이 있는" 액션은 `"store"`, `"append"` 및 `"callback"`입니다.

액션을 추가할 때, Option의 다음 클래스 어트리뷰트 중 하나 이상에 나열하여 분류해야 합니다 (모두 문자열 리스트입니다):

`Option.`**`ACTIONS`**
: 모든 액션은 ACTIONS에 나열되어야 합니다.

`Option.`**`STORE_ACTIONS`**
: "저장" 액션이 여기에 추가로 나열됩니다.

`Option.`**`TYPED_ACTIONS`**
: "형이 있는" 액션이 여기에 추가로 나열됩니다.

`Option.`**`ALWAYS_TYPED_ACTIONS`**
: 항상 형을 취하는 액션(즉, 옵션이 항상 값을 취하는 액션)이 여기에 추가로 나열됩니다. 이것의 유일한 효과는 *optparse*가 *ALWAYS_TYPED_ACTIONS*에 액션이 나열되는 명시적인 형이 없는 옵션에 기본형인 `"string"`을 대입한다는 것입니다.

새 액션을 실제로 구현하려면, Option의 `take_action()` 메서드를 재정의하고 액션을 인식하는 케이스를 추가해야 합니다.

예를 들어, `"extend"` 액션을 추가해 보겠습니다. 이것은 표준 `"append"` 액션과 유사하지만, 명령 줄에서 단일 값을 취해서 기존 리스트에 추가하는 대신, `"extend"`는 단일 쉼표로 구분된 문자열에서 여러 값을 취해서 기존 리스트를 확장합니다. 즉, `--names`가 `"string"` 형의 `"extend"` 옵션이면, 다음과 같은 명령 줄은

```
--names=foo,bar --names blah --names ding,dong
```

다음과 같은 리스트를 만듭니다

```
["foo", "bar", "blah", "ding", "dong"]
```

다시 Option의 서브 클래스를 정의합니다:

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

참고할만한 특징:

- "extend"는 명령 줄에서 값을 기대하기도 하고 그 값을 어딘가에 저장하기도 하므로, *STORE_ACTIONS*와 *TYPED_ACTIONS* 모두에 들어갑니다.
- *optparse*가 기본형 "string"을 "extend" 액션에 대입하도록 하기 위해, *ALWAYS_TYPED_ACTIONS*에도 "extend" 액션을 넣습니다.
- MyOption.take_action()은 이 하나의 새로운 액션만 구현하고, 표준 *optparse* 액션을 위해 제어를 Option.take_action()으로 되돌립니다.
- values는 매우 유용한 ensure_value() 메서드를 제공하는 optparse_parser.Values 클래스의 인스턴스입니다. ensure_value()는 본질적으로 안전밸브가 있는 *getattr()*입니다; 다음과 같이 호출됩니다

  ```
  values.ensure_value(attr, value)
  ```

  values의 attr 어트리뷰트가 존재하지 않거나 None이면, ensure_value()는 먼저 이를 value로 설정한 다음, value를 반환합니다. 이것은 "extend", "append" 및 "count"와 같은 액션에 매우 편리합니다. 이 액션들은 모두 변수에 데이터를 누적하고 해당 변수가 특정 형(처음 두 개는 리스트, 마지막은 정수)이 될 것으로 기대합니다. ensure_value()를 사용한다는 것은 액션을 사용하는 스크립트가 문제의 옵션 대상에 대한 기본값 설정에 대해 걱정할 필요가 없음을 의미합니다; 기본값을 None으로 그대로 둘 수 있으며 ensure_value()는 필요할 때 올바르게 처리합니다.

17.2.8 예외

exception optparse.**OptionError**

Raised if an *Option* instance is created with invalid or inconsistent arguments.

exception optparse.**OptionConflictError**

Raised if conflicting options are added to an *OptionParser*.

exception optparse.**OptionValueError**

Raised if an invalid option value is encountered on the command line.

exception optparse.**BadOptionError**

명령줄에서 잘못된 옵션이 전달되면 발생합니다.

exception optparse.**AmbiguousOptionError**

Raised if an ambiguous option is passed on the command line.

17.3 getpass — 이식성 있는 암호 입력

소스 코드: Lib/getpass.py

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

getpass 모듈은 두 함수를 제공합니다:

getpass.**getpass**(*prompt='Password: ', stream=None, *, echo_char=None*)

에코 없이 사용자에게 암호를 묻습니다. 사용자는 *prompt* 문자열을 사용하여 프롬프트 됩니다. 기본값은 `'Password: '`입니다. 유닉스에서 프롬프트는 필요하다면 replace 에러 처리기를 사용하여 파일류 객체 *stream*에 기록됩니다. *stream*는 제어 터미널(`/dev/tty`)이나 사용할 수 없으면 `sys.stderr`를 기본값으로 사용합니다 (이 인자는 윈도우에서 무시됩니다).

The *echo_char* argument controls how user input is displayed while typing. If *echo_char* is `None` (default), input remains hidden. Otherwise, *echo_char* must be a single printable ASCII character and each typed character is replaced by it. For example, `echo_char='*'` will display asterisks instead of the actual input.

에코가 없는 입력을 사용할 수 없으면, getpass()는 *stream*에 경고 메시지를 인쇄하고, `sys.stdin`에서 읽고, *GetPassWarning*을 방출하는 것으로 돌아갑니다.

> **참고**
>
> IDLE 내에서 getpass를 호출하면, 대기 중인 창 자체가 아닌 IDLE을 시작한 터미널에서 입력이 수행될 수 있습니다.

> **참고**
>
> On Unix systems, when *echo_char* is set, the terminal will be configured to operate in *noncanonical mode*. In particular, this means that line editing shortcuts such as `Ctrl+U` will not work and may insert unexpected characters into the input.

버전 3.14에서 변경: Added the *echo_char* parameter for keyboard feedback.

exception getpass.**GetPassWarning**

패스워드 입력이 에코 될 때 방출되는 *UserWarning* 서브 클래스.

getpass.**getuser**()

사용자의 "로그인 이름"을 반환합니다.

이 함수는 환경 변수 `LOGNAME`, `USER`, `LNAME` 및 `USERNAME`를 순서대로 검사하고, 비어 있지 않은 문자열로 설정된 첫 번째 값을 반환합니다. 아무것도 설정되지 않았으면, *pwd* 모듈을 지원하는 시스템에서는 암호 데이터베이스의 로그인 이름이 반환되고, 그렇지 않으면 *OSError*가 발생합니다.

일반적으로, 이 함수는 *os.getlogin()*보다 선호됩니다.

버전 3.13에서 변경: Previously, various exceptions beyond just *OSError* were raised.

17.4 fileinput — 여러 입력 스트림에서 줄을 이터레이트 하기

소스 코드: Lib/fileinput.py

이 모듈은 표준 입력이나 파일 목록에 대한 루프를 빠르게 작성하기 위한 도우미 클래스와 함수를 구현합니다. 단지 하나의 파일을 읽거나 쓰려면 *open()*을 참조하십시오.

일반적인 사용법은 다음과 같습니다:

```
import fileinput
for line in fileinput.input(encoding="utf-8"):
    process(line)
```

이것은 sys.argv[1:]에 나열된 모든 파일의 줄을 이터레이트 하며, 목록이 비어 있으면 기본값은 sys.stdin입니다. 파일 이름이 '-'이면, 이 또한 sys.stdin으로 대체되고 선택적 인자 *mode*와 *openhook*은 무시됩니다. 대체 파일명 목록을 지정하려면, *input()*의 첫 번째 인자로 전달하십시오. 단일 파일 이름도 허용됩니다.

모든 파일은 기본적으로 텍스트 모드로 열리지만, *input()*이나 *FileInput*을 호출할 때 *mode* 매개 변수를 지정하여 이를 재정의할 수 있습니다. 파일을 열거나 읽는 동안 I/O 에러가 발생하면, *OSError*가 발생합니다.

버전 3.3에서 변경: *IOError*가 발생했었습니다; 이제 이것은 *OSError*의 별칭입니다.

sys.stdin이 두 번 이상 사용되면, 대화식으로 사용되거나 명시적으로 재설정된 경우(예를 들어, sys.stdin.seek(0)을 사용해서)를 제외하고 두 번째와 그 이후의 사용은 줄을 반환하지 않습니다.

빈 파일은 열리고 즉시 닫힙니다; 파일명 목록에 존재함이 인식되는 유일한 시간은 마지막에 열린 파일이 비어있을 때입니다.

줄은 줄 바꿈이 그대로 유지된 채로 반환됩니다. 즉, 파일의 마지막 줄에는 줄 바꿈이 없을 수도 있습니다.

*fileinput.input()*이나 *FileInput()*의 *openhook* 매개 변수를 통해 열기 훅을 제공하여 파일을 여는 방법을 제어할 수 있습니다. 훅은 두 개의 인자 *filename*과 *mode*를 취하고, 그에 따라 열린 파일류 객체를 반환하는 함수여야 합니다. *encoding* 및/또는 *errors*가 지정되면, 추가 키워드 인자로 훅에 전달됩니다. 이 모듈은 압축 파일을 지원하기 위해 *hook_compressed()*를 제공합니다.

다음 함수는 이 모듈의 기본 인터페이스입니다:

fileinput.**input**(*files=None, inplace=False, backup='', *, mode='r', openhook=None, encoding=None, errors=None*)

FileInput 클래스의 인스턴스를 만듭니다. 인스턴스는 이 모듈의 함수에 대한 전역 상태로 사용되며, 이터레이션 중에 사용하기 위해 반환되기도 합니다. 이 함수의 매개 변수는 *FileInput* 클래스의 생성자로 전달됩니다.

FileInput 인스턴스는 with 문에서 컨텍스트 관리자로 사용될 수 있습니다. 이 예제에서, 예외가 발생하더라도 with 문이 종료된 후 *input*이 닫힙니다:

```
with fileinput.input(files=('spam.txt', 'eggs.txt'), encoding="utf-8") as f:
    for line in f:
        process(line)
```

버전 3.2에서 변경: 컨텍스트 관리자로 사용할 수 있습니다.

버전 3.8에서 변경: 키워드 매개 변수 *mode*와 *openhook*은 이제 키워드 전용입니다.

버전 3.10에서 변경: 키워드 전용 매개 변수 *encoding* 와 *errors*를 추가했습니다.

다음 함수는 *fileinput.input()*에 의해 만들어진 전역 상태를 사용합니다; 활성 상태가 없으면, *RuntimeError*가 발생합니다.

fileinput.**filename**()

현재 읽고 있는 파일의 이름을 반환합니다. 첫 번째 줄을 읽기 전에는, None을 반환합니다.

fileinput.**fileno**()

현재 파일의 정수 "파일 기술자"를 반환합니다. 파일이 열리지 않았으면 (첫 번째 줄 전과 파일 사이에), -1을 반환합니다.

fileinput.**lineno**()

방금 읽은 줄의 누적 줄 번호를 반환합니다. 첫 번째 줄을 읽기 전에는, 0을 반환합니다. 마지막 파일의 마지막 줄을 읽은 후에는, 그 줄의 줄 번호를 반환합니다.

fileinput.**filelineno**()

현재 파일의 줄 번호를 반환합니다. 첫 번째 줄을 읽기 전에는, 0을 반환합니다. 마지막 파일의 마지막 줄을 읽은 후에는, 그 줄의 파일 내에서의 줄 번호를 반환합니다.

fileinput.**isfirstline**()

방금 읽은 줄이 파일의 첫 번째 줄이면 True를, 그렇지 않으면 False를 반환합니다.

fileinput.**isstdin**()

마지막 줄을 sys.stdin에서 읽었으면 True를, 그렇지 않으면 False를 반환합니다.

fileinput.**nextfile**()

다음 이터레이션에서 다음 파일(있다면)의 첫 번째 줄을 읽도록 현재 파일을 닫습니다; 파일에서 읽지 않은 줄은 누적 줄 수에 포함되지 않습니다. 파일명은 다음 파일의 첫 번째 줄을 읽을 때까지 변경되지 않습니다. 첫 번째 줄을 읽기 전에는, 이 함수가 효과가 없습니다; 첫 번째 파일을 건너뛰는 데 사용할 수 없습니다. 마지막 파일의 마지막 줄을 읽은 후에는, 이 함수는 효과가 없습니다.

fileinput.**close**()

시퀀스를 닫습니다.

모듈이 제공하는 시퀀스 동작을 구현하는 클래스는 서브 클래싱에도 사용할 수 있습니다:

class fileinput.**FileInput**(*files=None, inplace=False, backup=”, *, mode=’r’, openhook=None, encoding=None, errors=None*)

FileInput 클래스는 구현입니다; 그 메서드 *filename()*, *fileno()*, *lineno()*, *filelineno()*, *isfirstline()*, *isstdin()*, *nextfile()* 및 *close()*는 모듈에 있는 같은 이름의 함수에 해당합니다. 또한 이터러블이고, 다음 입력 줄을 반환하는 *readline()* 메서드가 있습니다. 시퀀스는 엄격하게 순차적으로 액세스해야 합니다; 무작위 액세스와 *readline()*은 혼합될 수 없습니다.

*mode*로 *open()*에 전달할 파일 모드를 지정할 수 있습니다. 'r' 과 'rb' 중 하나여야 합니다.

*openhook*이 제공되면 두 개의 인자 *filename*과 *mode*를 취하고, 이에 따라 열린 파일류 객체를 반환하는 함수여야 합니다. *inplace*와 *openhook*을 함께 사용할 수 없습니다.

You can specify *encoding* and *errors* that is passed to *open()* or *openhook*.

FileInput 인스턴스는 with 문에서 컨텍스트 관리자로 사용될 수 있습니다. 이 예제에서, 예외가 발생하더라도 with 문이 종료된 후 *input*이 닫힙니다:

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

버전 3.2에서 변경: 컨텍스트 관리자로 사용할 수 있습니다.

버전 3.8에서 변경: 키워드 매개 변수 *mode*와 *openhook*은 이제 키워드 전용입니다.

버전 3.10에서 변경: 키워드 전용 매개 변수 *encoding* 와 *errors*를 추가했습니다.

버전 3.11에서 변경: 'rU' 및 'U' 모드와 __getitem__() 메서드는 제거했습니다.

선택적 제자리 필터링(in-place filtering): 키워드 인자 inplace=True가 *fileinput.input()*이나 *FileInput* 생성자로 전달되면, 파일이 백업 파일로 이동되고 표준 출력은 입력 파일로 보내집니다(백업 파일과 같은 이름의 파일이 이미 있으면, 조용히 대체됩니다). 이를 통해 입력 파일을 다시 쓰는 필터를 작성할 수 있습니다. *backup* 매개 변수가 제공되면 (일반적으로 backup='.<some extension>'으로), 백업 파일의 확장자를 지정하고, 백업 파일은 그대로 남아 있습니다; 기본적으로 확장자는 '.bak'이고, 출력 파일을 닫을 때 삭제됩니다. 표준 입력을 읽을 때는 제자리 필터링이 비활성화됩니다.

이 모듈은 다음과 같은 두 개의 열기 훅을 제공합니다:

fileinput.**hook_compressed**(*filename, mode, *, encoding=None, errors=None*)

*gzip*과 *bz2* 모듈을 사용하여 gzip과 bzip2로 압축된 파일(확장자 '.gz'와 '.bz2'로 인식합니다)을 투명하게 엽니다. 파일명 확장자가 '.gz'나 '.bz2'가 아니면, 파일이 정상적으로 열립니다 (즉, 압축 해제 없이 *open()*을 사용합니다).

The *encoding* and *errors* values are passed to *io.TextIOWrapper* for compressed files and open for normal files.

사용 예: fi = fileinput.FileInput(openhook=fileinput.hook_compressed, encoding="utf-8")

버전 3.10에서 변경: 키워드 전용 매개 변수 *encoding* 와 *errors*를 추가했습니다.

`fileinput.hook_encoded(encoding, errors=None)`

주어진 *encoding*과 *errors*를 사용하여 파일을 읽도록 `open()`으로 각 파일을 여는 훅을 반환합니다.

사용 예: `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))`

버전 3.6에서 변경: 선택적 *errors* 매개 변수를 추가했습니다.

버전 3.10부터 폐지됨: This function is deprecated since `fileinput.input()` and `FileInput` now have *encoding* and *errors* parameters.

17.5 curses — 문자 셀 디스플레이를 위한 터미널 처리

소스 코드: Lib/curses

`curses` 모듈은 이식성 있는 고급 터미널 처리를 위한 사실상의 표준인 curses 라이브러리에 대한 인터페이스를 제공합니다.

curses는 유닉스 환경에서 가장 널리 사용되지만, 윈도우, DOS 및 기타 시스템에서도 사용할 수 있는 버전이 있습니다. 이 확장 모듈은 리눅스와 유닉스의 BSD 변형에서 동작하는 오픈 소스 curses 라이브러리인 ncurses의 API와 일치하도록 설계되었습니다.

가용성: not Android, not iOS, not WASI.

이 모듈은 모바일 플랫폼이나 웹어셈블리 플랫폼에서 지원되지 않습니다.

참고

설명서에 문자(*character*)가 언급될 때마다 정수, 한 문자 유니코드 문자열 또는 한 바이트 바이트열로 지정할 수 있습니다.

설명서에 문자 문자열(*character string*)이 언급될 때마다 유니코드 문자열이나 바이트열로 지정할 수 있습니다.

더 보기

모듈 `curses.ascii`
로케일 설정과 관계없이, ASCII 문자로 작업하기 위한 유틸리티.

모듈 `curses.panel`
curses 창에 깊이를 추가하는 패널 스택 확장.

모듈 `curses.textpad`
Emacs와 유사한 바인딩을 지원하는 curses를 위한 편집 가능한 텍스트 위젯.

curses-howto
Andrew Kuchling과 Eric Raymond가 작성한, curses를 파이썬에서 사용하는 것에 대한 자습서 자료.

17.5.1 함수

`curses` 모듈은 다음 예외를 정의합니다:

exception `curses.error`

curses 라이브러리 함수가 에러를 반환할 때 발생하는 예외.

> **참고**
>
> 함수나 메서드에 대한 *x*나 *y* 인자가 선택 사항일 때마다, 기본값은 현재 커서 위치입니다. *attr*이 선택적일 때마다 기본값은 `A_NORMAL`입니다.

`curses` 모듈은 다음 함수를 정의합니다:

`curses.`**`assume_default_colors`**(*fg*, *bg*, /)

Allow use of default values for colors on terminals supporting this feature. Use this to support transparency in your application.

- Assign terminal default foreground/background colors to color number `-1`. So `init_pair(x, COLOR_RED, -1)` will initialize pair *x* as red on default background and `init_pair(x, -1, COLOR_BLUE)` will initialize pair *x* as default foreground on blue.
- Change the definition of the color-pair `0` to `(fg, bg)`.

Added in version 3.14.

`curses.`**`baudrate`**()

터미널의 출력 속도를 초당 비트 수로 반환합니다. 소프트웨어 터미널 에뮬레이터에서는 고정된 큰 값을 갖습니다. 역사적인 이유로 포함되었습니다; 이전에는, 시간 지연에 대한 출력 루프를 작성하는데, 때로는 회선 속도에 따라 인터페이스를 변경하는 데 사용되었습니다.

`curses.`**`beep`**()

짧은 주의 음을 냅니다.

`curses.`**`can_change_color`**()

프로그래머가 터미널에 표시되는 색상을 변경할 수 있는지에 따라 `True`나 `False`를 반환합니다.

`curses.`**`cbreak`**()

cbreak 모드로 들어갑니다. cbreak 모드("드문(rare)" 모드라고도 합니다)에서는 일반 tty 줄 버퍼링이 꺼지고 문자를 하나씩 읽을 수 있습니다. 그러나, 원시(raw) 모드와 달리, 특수 문자(인터럽트(interrupt), 종료(quit), 일시 중단(suspend) 및 흐름 제어(flow control))는 tty 드라이버와 호출하는 프로그램에 영향을 미칩니다. `raw()`를 먼저 호출한 다음 `cbreak()`를 호출하면 터미널이 cbreak 모드로 유지됩니다.

`curses.`**`color_content`**(*color_number*)

`0`과 `COLORS - 1` 사이의 색상 *color_number*에서 빨강, 녹색 및 파랑 (RGB) 구성 요소의 강도(intensity)를 반환합니다. 주어진 색상에 대한 R,G,B 값이 포함된 3-튜플을 반환합니다. 이 값은 `0`(구성 요소 없음)과 `1000`(구성 요소의 최대량) 사이입니다.

`curses.`**`color_pair`**(*pair_number*)

지정된 색상 쌍으로 텍스트를 표시하기 위한 속성값을 반환합니다. 처음 256 색상 쌍만 지원됩니다. 이 속성값은 `A_STANDOUT`, `A_REVERSE` 및 기타 `A_*` 속성과 결합할 수 있습니다. `pair_number()`는 이 함수의 역입니다.

`curses.`**`curs_set`**(*visibility*)

커서 상태를 설정합니다. *visibility*는 `0`, `1` 또는 `2`로 설정될 수 있는데, 각각 보이지 않음(invisible), 보통, 매우 잘 보임(very visible)입니다. 터미널이 요청된 visibility를 지원하면, 이전 커서 상태를 반환합니다; 그렇지 않으면 예외가 발생합니다. 많은 터미널에서, "보이는(visible)" 모드는 밑줄 커서이고 "매우 잘 보이는(very visible)" 모드는 블록 커서입니다.

`curses.`**`def_prog_mode`**()

현재 터미널 모드를 "프로그램(program)" 모드로 저장합니다. 프로그램 모드는 실행 중인 프로그램이 curses를 사용 중인 모드입니다. (반대는 "셸(shell)" 모드이며, 프로그램이 curses를 사용하지 않을 때입니다.) 이후에 `reset_prog_mode()`를 호출하면 이 모드가 복원됩니다.

`curses.`**`def_shell_mode`**()

현재 터미널 모드를 "셸(shell)" 모드로 저장합니다. 셸 모드는 실행 중인 프로그램이 curses를 사용

하지 않는 모드입니다. (반대는 "프로그램(program)" 모드이며, 프로그램이 curses 기능을 사용 중일 때입니다.) 이후에 *reset_shell_mode()*를 호출하면 이 모드가 복원됩니다.

curses.**delay_output**(*ms*)

출력에 *ms* 밀리초 일시 중지를 삽입합니다.

curses.**doupdate**()

물리적 화면을 갱신합니다. curses 라이브러리는 두 개의 데이터 구조를 유지합니다. 하나는 현재 물리적 화면 내용을 표현하고 다른 하나는 원하는 다음 상태를 나타내는 가상 화면을 표현합니다. *doupdate()*는 물리적 화면을 가상 화면과 일치하도록 갱신합니다.

가상 화면은 창에 *addstr()*과 같은 쓰기 연산이 수행된 후 *noutrefresh()* 호출로 갱신될 수 있습니다. 일반적인 *refresh()* 호출은 단순히 `noutrefresh()` 후에 `doupdate()` 하는 것입니다; 여러 개의 창을 갱신해야 하면, 모든 창에서 `noutrefresh()` 호출을 실행한 다음 단일 `doupdate()`를 실행하여 속도 성능을 높이고 아마도 화면 깜박임을 줄일 수 있습니다.

curses.**echo**()

반향(echo) 모드로 들어갑니다. 반향 모드에서는, 각 문자 입력이 입력되는 대로 화면에 반향을 일으킵니다.

curses.**endwin**()

라이브러리를 초기화 해제하고, 터미널을 정상 상태로 되돌립니다.

curses.**erasechar**()

사용자의 현재 지우기 문자(erase character)를 1바이트 바이트열 객체로 반환합니다. 유닉스 운영체제에서 이는 curses 프로그램의 제어 tty의 특성이며, curses 라이브러리 자체에 의해 설정되지 않습니다.

curses.**filter**()

filter() 루틴을 사용하는 경우 *initscr()*을 호출하기 전에 호출해야 합니다. 이러한 호출 중에, 효과는 `LINES`가 `1`로 설정되고; `clear`, `cup`, `cud`, `cud1`, `cuu1`, `cuu`, `vpa` 기능이 비활성화되고; `home` 문자열이 `cr` 값으로 설정됩니다. 결과적으로 커서가 현재 줄에 갇혀서 화면이 갱신됩니다. 화면의 나머지 부분을 건드리지 않고 한 번에 한 글자(character-at-a-time) 줄 편집을 활성화하는데 사용될 수 있습니다.

curses.**flash**()

화면을 깜박입니다. 즉, 반전 비디오로 변경한 다음 짧은 간격으로 되돌립니다. 어떤 사람들은 *beep()*이 만드는 가청 주의 신호보다 이런 '시각적 벨' 을 선호합니다.

curses.**flushinp**()

모든 입력 버퍼를 플러시 합니다. 이렇게 하면 사용자가 입력했지만, 아직 프로그램에서 처리하지 않은 모든 선행 입력(typeahead)을 모두 버립니다.

curses.**getmouse**()

*getch()*가 *KEY_MOUSE*를 반환하여 마우스 이벤트를 알린 후, 이 메서드는 대기 중인 마우스 이벤트를 꺼내기 위해 호출되어야 합니다, 이벤트는 5-튜플 `(id, x, y, z, bstate)`로 표현됩니다. *id*는 여러 장치를 구별하는 데 사용되는 ID 값이며, *x*, *y*, *z*는 이벤트 좌표입니다. (*z*는 현재 사용되지 않습니다.) *bstate*는 비트가 이벤트 유형을 나타내도록 설정되는 정숫값이며, 다음 상수 중 하나 이상의 비트별 OR입니다. 여기서 *n*은 1에서 5까지의 버튼 번호입니다: *BUTTONn_PRESSED*, *BUTTONn_RELEASED*, *BUTTONn_CLICKED*, *BUTTONn_DOUBLE_CLICKED*, *BUTTONn_TRIPLE_CLICKED*, *BUTTON_SHIFT*, *BUTTON_CTRL*, *BUTTON_ALT*.

버전 3.10에서 변경: The `BUTTON5_*` constants are now exposed if they are provided by the underlying curses library.

curses.**getsyx**()

가상 화면 커서의 현재 좌표를 튜플 `(y, x)`로 반환합니다. *leaveok*가 현재 `True`이면, `(-1, -1)`을 반환합니다.

curses.**getwin**(*file*)

이전 *window.putwin()* 호출로 파일에 저장된 창 관련 데이터를 읽습니다. 그런 다음 루틴은 해당 데이터를 사용하여 새 창을 만들고 초기화하여, 새 창 객체를 반환합니다.

curses.**has_colors**()

터미널이 색상을 표시할 수 있으면 True를 반환합니다. 그렇지 않으면 False를 반환합니다.

curses.**has_extended_color_support**()

Return True if the module supports extended colors; otherwise, return False. Extended color support allows more than 256 color pairs for terminals that support more than 16 colors (e.g. xterm-256color).

Extended color support requires ncurses version 6.1 or later.

Added in version 3.10.

curses.**has_ic**()

터미널에 문자 삽입과 삭제 기능이 있으면 True를 반환합니다. 이 함수는 역사적인 이유로만 포함되는데, 모든 최신 소프트웨어 터미널 에뮬레이터에 이러한 기능이 있기 때문입니다.

curses.**has_il**()

터미널에 줄 삽입과 삭제 기능이 있거나, 영역 스크롤을 사용하여 시뮬레이션 할 수 있으면 True를 반환합니다. 이 함수는 역사적인 이유로만 포함되는데, 모든 최신 소프트웨어 터미널 에뮬레이터에 이러한 기능이 있기 때문입니다.

curses.**has_key**(*ch*)

키값 *ch*를 취하고, 현재 터미널 유형이 해당 값을 가진 키를 인식하면 True를 반환합니다.

curses.**halfdelay**(*tenths*)

사용자가 입력 한 문자를 프로그램에서 즉시 사용할 수 있다는 점에서 cbreak 모드와 유사한, 반 지연 (half-delay) 모드에 사용됩니다. 그러나 *tenths* 10분의 1초 동안 블록한 후 아무것도 입력하지 않으면 예외가 발생합니다. *tenths*의 값은 1과 255 사이의 숫자여야 합니다. 반 지연 모드를 종료하려면 *nocbreak()*를 사용하십시오.

curses.**init_color**(*color_number*, *r*, *g*, *b*)

변경될 색상 번호와 뒤따르는 세 RGB 값(빨강, 녹색 및 파랑 성분의 양)을 취해서 색상의 정의를 변경합니다. *color_number*의 값은 0과 COLORS - 1 사이여야 합니다. *r*, *g*, *b*는 각각 0과 1000 사이의 값이어야 합니다. *init_color()*를 사용하면, 화면에서 해당 색상의 모든 항목이 즉시 새 정의로 변경됩니다. 이 함수는 대부분의 터미널에서 효과가 없습니다; *can_change_color()*가 True를 반환할 때만 활성화됩니다.

curses.**init_pair**(*pair_number*, *fg*, *bg*)

Change the definition of a color-pair. It takes three arguments: the number of the color-pair to be changed, the foreground color number, and the background color number. The value of *pair_number* must be between 1 and COLOR_PAIRS - 1 (the 0 color pair can only be changed by *use_default_colors()* and *assume_default_colors()*). The value of *fg* and *bg* arguments must be between 0 and COLORS - 1, or, after calling use_default_colors() or assume_default_colors(), -1. If the color-pair was previously initialized, the screen is refreshed and all occurrences of that color-pair are changed to the new definition.

curses.**initscr**()

라이브러리를 초기화합니다. 전체 화면을 나타내는 창 객체를 반환합니다.

> **참고**
>
> 터미널을 여는 중 에러가 발생하면, 하부 curses 라이브러리가 인터프리터를 종료시킬 수 있습니다.

curses.**is_term_resized**(*nlines*, *ncols*)

*resize_term()*이 창 구조를 수정한다면 True를, 그렇지 않으면 False를 반환합니다.

curses.**isendwin**()

*endwin()*이 호출되었으면 (즉, curses 라이브러리가 초기화 해제되었다면) True를 반환합니다.

curses.**keyname**(*k*)

키 번호 *k*의 이름을 바이트열 객체로 반환합니다. 인쇄 가능한 ASCII 문자를 생성하는 키의 이름은 키의 문자입니다. 제어 키(control-key) 조합의 이름은 캐럿(`b'^'`)과 그 뒤에오는 해당 인쇄 가능한 ASCII 문자로 구성된 2바이트 바이트열 객체입니다. 대체 키(alt-key) 조합(128–255)의 이름은 접두사 `b'M-'`와 그 뒤에 오는 해당 ASCII 문자의 이름으로 구성되는 바이트열 객체입니다.

curses.**killchar**()

사용자의 현재 줄 삭제 문자(line kill character)를 1바이트 바이트열 객체로 반환합니다. 유닉스 운영 체제에서 이는 curses 프로그램의 제어 tty의 특성이며, curses 라이브러리 자체에 의해 설정되지 않습니다.

curses.**longname**()

현재 터미널을 설명하는 terminfo 긴 이름 필드를 포함하는 바이트열 객체를 반환합니다. 자세한 설명의 최대 길이는 128자입니다. *initscr()*을 호출한 후에만 정의됩니다.

curses.**meta**(*flag*)

*flag*가 `True`이면, 8비트 문자 입력을 허용합니다. *flag*가 `False`이면 7비트 문자만 허용합니다.

curses.**mouseinterval**(*interval*)

클릭으로 인식되기 위해 눌림(press)과 해제(release) 이벤트 사이에 지날 수 있는 최대 시간을 밀리초 단위로 설정하고, 이전 간격 값을 반환합니다. 기본값은 200 밀리 초, 즉 1/5초입니다.

curses.**mousemask**(*mousemask*)

마우스 이벤트가 보고되도록 설정하고, 튜플 `(availmask, oldmask)`를 반환합니다. *availmask*는 지정된 마우스 이벤트 중 보고 할 수 있는 것을 나타냅니다; 완전히 실패하면 `0`을 반환합니다. *oldmask*는 주어진 창의 마우스 이벤트 마스크의 이전 값입니다. 이 함수를 한 번도 호출하지 않으면, 마우스 이벤트가 보고되지 않습니다.

curses.**napms**(*ms*)

ms 밀리초 동안 휴면합니다.

curses.**newpad**(*nlines*, *ncols*)

주어진 수의 행과 열로 새로운 패드 데이터 구조에 대한 포인터를 만들고 반환합니다. 패드를 창 객체로 반환합니다.

패드는 화면 크기에 의해 제한되지 않으며, 화면의 특정 부분과 반드시 관련될 필요는 없다는 점을 제외하면 창과 같습니다. 큰 창이 필요할 때 패드를 사용할 수 있으며, 한 번에 창의 일부만 화면에 표시됩니다. 패드의 자동 새로 고침(가령 스크롤이나 입력 반향)은 일어나지 않습니다. 패드의 *refresh()*와 *noutrefresh()* 메서드는 표시할 패드의 부분과 표시할 화면의 위치를 지정하기 위해 6개의 인자가 필요합니다. 인자는 *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*입니다; *p* 인자는 표시할 패드 영역의 왼쪽 위 모서리를 가리키고, *s* 인자는 패드 영역이 표시될 화면 위의 클리핑 상자를 정의합니다.

curses.**newwin**(*nlines*, *ncols*)

curses.**newwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)

왼쪽 위 모서리가 `(begin_y, begin_x)`이고, 높이/너비가 *nlines*/*ncols* 인 새 창을 반환합니다.

기본적으로, 창은 지정된 위치에서 화면 오른쪽 하단으로 확장됩니다.

curses.**nl**()

줄 바꿈(newline) 모드로 들어갑니다. 이 모드는 입력에서 리턴 키를 줄 바꿈으로 변환하고, 출력에서 줄 바꿈을 리턴(return)과 줄 넘김(line-feed)으로 변환합니다. 줄 바꿈 모드는 처음에 켜져 있습니다.

curses.**nocbreak**()

cbreak 모드를 종료합니다. 줄 버퍼링을 사용하는 일반 "요리된(cooked)" 모드로 돌아갑니다.

curses.**noecho**()

반향 모드를 종료합니다. 입력 문자 반향이 꺼집니다.

curses.**nonl**()

줄 바꿈 모드를 종료합니다. 입력에서 리턴을 줄 바꿈으로 변환하는 것을 비활성화하고, 출력에서 줄 바꿈을 줄 바꿈/리턴으로 저수준 변환하는 것을 비활성화합니다 (그러나 이것은 `addch('\n')` 의 동작을 변경하지는 않는데, 항상 가상 화면에서 리턴 (return) 과 줄 넘김 (line feed) 에 동등한 역할을 합니다). 변환이 꺼져 있으면, curses 가 때로 수직 동작 속도를 약간 올릴 수 있습니다; 또한, 입력에서 리턴 키를 감지할 수 있습니다.

curses.**noqiflush**()

`noqiflush()` 루틴이 사용되면, `INTR`, `QUIT` 및 `SUSP` 문자와 연관된 입력과 출력 큐의 일반 플러시가 수행되지 않습니다. 처리기가 종료한 후, 인터럽트가 발생하지 않은 것처럼 출력을 계속하려면 시그널 처리기에서 `noqiflush()` 를 호출할 수 있습니다.

curses.**noraw**()

원시 (raw) 모드를 종료합니다. 줄 버퍼링을 사용하는 일반 "요리된 (cooked)" 모드로 돌아갑니다.

curses.**pair_content**(*pair_number*)

요청된 색상 쌍의 색상이 포함된 튜플 `(fg, bg)` 를 반환합니다. *pair_number* 의 값은 `0` 과 `COLOR_PAIRS - 1` 사이여야 합니다.

curses.**pair_number**(*attr*)

속성값 *attr* 로 설정된 색상 쌍의 번호를 반환합니다. *color_pair()* 는 이 함수의 역입니다.

curses.**putp**(*str*)

`tputs(str, 1, putchar)` 와 동등합니다; 현재 터미널에 대해 지정된 terminfo 기능의 값을 내보냅니다. *putp()* 의 출력은 항상 표준 출력을 향함에 유의하십시오.

curses.**qiflush**([*flag*])

flag 가 `False` 이면, 효과는 *noqiflush()* 를 호출하는 것과 같습니다. *flag* 가 `True` 이거나, 인자가 제공되지 않으면, 이러한 제어 문자를 읽을 때 큐가 플러시 됩니다.

curses.**raw**()

원시 (raw) 모드로 들어갑니다. 원시 모드에서는, 일반 줄 버퍼링과 인터럽트, 종료, 일시 중단 및 흐름 제어 키 처리가 꺼집니다; curses 입력 함수로 문자가 한 번에 하나씩 제시됩니다.

curses.**reset_prog_mode**()

def_prog_mode() 로 이전에 저장한 대로, 터미널을 "프로그램 (program)" 모드로 복원합니다.

curses.**reset_shell_mode**()

def_shell_mode() 로 이전에 저장한 대로, 터미널을 "셸 (shell)" 모드로 복원합니다.

curses.**resetty**()

터미널 모드의 상태를 *savetty()* 에 대한 마지막 호출 때의 상태로 복원합니다.

curses.**resize_term**(*nlines*, *ncols*)

resizeterm() 이 사용하는 백 엔드 함수로, 대부분의 작업을 수행합니다; 창 크기를 조정할 때, *resize_term()* 은 확장되는 영역을 공백으로 채웁니다. 호출하는 응용 프로그램은 이러한 영역을 적절한 데이터로 채워야 합니다. `resize_term()` 함수는 모든 창의 크기를 조정하려고 합니다. 그러나, 패드의 호출 규칙으로 인해, 응용 프로그램과의 추가 상호 작용 없이 이들의 크기를 조정할 수 없습니다.

curses.**resizeterm**(*nlines*, *ncols*)

표준과 현재 창의 크기를 지정된 크기로 조정하고, 창 크기를 기록하는 curses 라이브러리에서 사용하는 다른 관리 데이터를 조정합니다 (특히 SIGWINCH 처리기).

curses.**savetty**()

터미널 모드의 현재 상태를 *resetty()* 에서 사용 가능한 버퍼에 저장합니다.

curses.**get_escdelay**()

set_escdelay() 로 설정된 값을 가져옵니다.

Added in version 3.9.

curses.**set_escdelay**(*ms*)

키보드에 입력된 개별 이스케이프 문자와 커서와 기능키로 전송된 이스케이프 시퀀스를 구별하기 위해, 이스케이프 문자를 읽은 후 기다릴 밀리초 수를 설정합니다.

Added in version 3.9.

curses.**get_tabsize**()

*set_tabsize()*로 설정된 값을 가져옵니다.

Added in version 3.9.

curses.**set_tabsize**(*size*)

탭을 창에 추가해서 탭 문자를 스페이스로 변환할 때 curses 라이브러리가 사용하는 열 수를 설정합니다.

Added in version 3.9.

curses.**setsyx**(*y*, *x*)

가상 화면 커서를 *y*, *x*로 설정합니다. *y*와 *x*가 모두 `-1`이면, *leaveok*는 `True`로 설정됩니다.

curses.**setupterm**(*term=None*, *fd=-1*)

터미널을 초기화합니다. *term*은 터미널 이름을 제공하는 문자열이거나 `None`입니다; 생략되거나 `None`이면, `TERM` 환경 변수의 값이 사용됩니다. *fd*는 초기화 시퀀스가 전송될 파일 기술자입니다; 제공되지 않거나 `-1`이면, `sys.stdout`의 파일 기술자가 사용됩니다.

curses.**start_color**()

프로그래머가 색상을 사용하려면, 다른 색상 조작 루틴을 호출하기 전에 호출해야 합니다. *initscr()* 직후 이 루틴을 호출하는 것이 좋습니다.

*start_color()*는 8개의 기본 색상(검정, 빨강, 녹색, 노랑, 파랑, 마젠타, 시안 및 흰색)과 터미널이 지원할 수 있는 색상과 색상 쌍의 최댓값인 *curses* 모듈의 2개의 전역 변수 *COLORS*와 *COLOR_PAIRS*를 초기화합니다. 또한 터미널의 전원을 켰을 때의 값으로 터미널의 색상을 복원합니다.

curses.**termattrs**()

터미널이 지원하는 모든 비디오 속성의 논리적 OR를 반환합니다. 이 정보는 curses 프로그램이 화면 모양을 완전히 제어해야 할 때 유용합니다.

curses.**termname**()

환경 변수 `TERM`의 값을 14자로 잘린 바이트열 객체로 반환합니다.

curses.**tigetflag**(*capname*)

terminfo 기능 이름 *capname*에 해당하는 불리언 기능의 값을 정수로 반환합니다. *capname*이 불리언 기능이 아니면 `-1` 값을, 터미널 설명에서 취소되었거나 빠졌으면 `0`을 반환합니다.

curses.**tigetnum**(*capname*)

terminfo 기능 이름 *capname*에 해당하는 숫자 기능의 값을 정수로 반환합니다. *capname*이 숫자 기능이 아니면 `-2` 값을, 터미널 설명에서 취소되었거나 빠졌으면 `-1`을 반환합니다.

curses.**tigetstr**(*capname*)

terminfo 기능 이름 *capname*에 해당하는 문자열 기능의 값을 바이트열 객체로 반환합니다. *capname*이 terminfo "문자열 기능"이 아니거나, 터미널 설명에서 취소되었거나 빠졌으면 `None`을 반환합니다.

curses.**tparm**(*str*[, ...])

제공된 매개 변수를 사용하여 바이트열 객체 *str*을 인스턴스 화합니다. 여기서 *str*은 terminfo 데이터베이스에서 얻은 매개 변수화된 문자열이어야 합니다. 예를 들어 `tparm(tigetstr("cup"), 5, 3)`은 `b'\033[6;4H'`가 될 수 있습니다, 정확한 결과는 터미널 유형에 따라 다릅니다.

curses.**typeahead**(*fd*)

파일 기술자 *fd*가 선행 입력 검사(typeahead checking)에 사용되도록 지정합니다. *fd*가 `-1`이면, 선행 입력 검사가 수행되지 않습니다.

curses 라이브러리는 화면을 갱신하는 동안 정기적으로 선행 입력을 들여다보고 "라인 브레이크 아웃 최적화(line-breakout optimization)"를 수행합니다. 입력이 발견되고, 그것이 tty에서 왔으면, refresh

나 doupdate가 다시 호출될 때까지 현재 갱신을 연기해서, 앞서 입력된 명령에 더 빠르게 응답 할 수 있도록 합니다. 이 함수를 사용하면 선행 입력 검사를 위해 다른 파일 기술자를 지정할 수 있습니다.

curses.**unctrl**(*ch*)

문자 *ch*의 인쇄 가능한 표현인 바이트열 객체를 반환합니다. 제어 문자는 캐럿과 그 뒤에 오는 문자로 표시됩니다, 예를 들어 `b'^C'`. 인쇄 문자는 그대로 남아 있습니다.

curses.**ungetch**(*ch*)

다음 `getch()`가 반환하도록 *ch*를 푸시합니다.

> **참고**
>
> `getch()`가 호출되기 전에 하나의 *ch* 만 푸시할 수 있습니다.

curses.**update_lines_cols**()

`LINES`와 `COLS` 모듈 변수를 갱신합니다. 수동 화면 크기 조정을 감지하는 데 유용합니다.

Added in version 3.5.

curses.**unget_wch**(*ch*)

다음 `get_wch()`가 반환하도록 *ch*를 푸시합니다.

> **참고**
>
> `get_wch()`가 호출되기 전에 하나의 *ch* 만 푸시할 수 있습니다.

Added in version 3.3.

curses.**ungetmouse**(*id, x, y, z, bstate*)

주어진 상태 데이터와 결합하여, `KEY_MOUSE` 이벤트를 입력 큐로 푸시합니다.

curses.**use_env**(*flag*)

사용되면, 이 함수는 `initscr()`이나 newterm을 호출하기 전에 호출해야 합니다. *flag*가 `False`이면, 환경 변수 `LINES`와 `COLUMNS`(기본적으로 사용됩니다)가 설정되어 있거나, 창에서 curses가 실행 중인 경우(이때 기본 동작은 `LINES`와 `COLUMNS`가 설정되지 않았으면 창 크기를 사용하는 것입니다)에도 terminfo 데이터베이스에 지정된 행(lines)과 열(columns)의 값이 사용됩니다.

curses.**use_default_colors**()

Equivalent to `assume_default_colors(-1, -1)`.

curses.**wrapper**(*func, /, *args, **kwargs*)

curses를 초기화하고 다른 콜러블 객체 *func*를 호출하는데, 이 콜러블은 curses를 사용하는 응용 프로그램의 나머지 부분이어야 합니다. 응용 프로그램에서 예외가 발생하면, 이 함수는 예외를 다시 발생시키고 트레이스백을 생성하기 전에 터미널을 정상 상태로 복원합니다. 콜러블 객체 *func*는 메인 창 'stdscr' 을 첫 번째 인자로 전달받고 `wrapper()`로 전달된 다른 모든 인자가 그 뒤를 따릅니다. *func*를 호출하기 전에, `wrapper()`는 cbreak 모드를 켜고, 반향을 끄고, 터미널 키패드를 활성화하고, 터미널에 색상이 지원되면 색상을 초기화합니다. 빠져나갈 때 (정상인지 예외로 인한 것인지 관계없이) 요리된(cooked) 모드를 복원하고, 반향을 켜고, 터미널 키패드를 비활성화합니다.

17.5.2 창 객체

class curses.**window**

위의 `initscr()`과 `newwin()`에 의해 반환된 창 객체에는 다음과 같은 메서드와 어트리뷰트가 있습니다:

window.**addch**(*ch*[, *attr*])

window.**addch**(*y*, *x*, *ch*[, *attr*])

속성 *attr*로 (y, x)에 문자 *ch*를 그려서, 그 위치에 이전에 그린 문자를 덮어씁니다. 기본적으로, 문자 위치와 속성은 창 객체의 현재 설정입니다.

> **참고**
>
> 창, 하위 창(subwindow) 또는 패드 외부에 쓰면 `curses.error`가 발생합니다. 창, 하위 창 또는 패드의 오른쪽 하단에 쓰려고 하면 문자가 인쇄된 후 예외가 발생합니다.

window.**addnstr**(*str*, *n*[, *attr*])

window.**addnstr**(*y*, *x*, *str*, *n*[, *attr*])

(y, x)에 문자열 *str*의 최대 *n* 문자를 속성 *attr*로 그려, 이전 디스플레이의 내용을 덮어씁니다.

window.**addstr**(*str*[, *attr*])

window.**addstr**(*y*, *x*, *str*[, *attr*])

(y, x)에 문자열 *str*을 속성 *attr*로 그려, 이전 디스플레이의 내용을 덮어씁니다.

> **참고**
>
> - 창, 하위 창 또는 패드 외부에 쓰면 `curses.error`가 발생합니다. 창, 하위 창 또는 패드의 오른쪽 하단에 쓰려고 하면 문자열이 인쇄된 후 예외가 발생합니다.
> - 이 파이썬 모듈의 백 엔드인 ncurses에 있는 버그가 창 크기를 조정할 때 세그멘테이션 오류를 유발할 수 있습니다. 이것은 ncurses-6.1-20190511에서 수정되었습니다. 이전 ncurses에 갇혀 있다면, 줄 바꿈이 포함된 *str*로 `addstr()`을 호출하지 않으면 트리거를 피할 수 있습니다. 대신, 줄마다 `addstr()`을 따로 호출하십시오.

window.**attroff**(*attr*)

현재 창에 대한 모든 쓰기에 적용된 "배경(background)" 집합에서 속성 *attr*을 제거합니다.

window.**attron**(*attr*)

현재 창에 대한 모든 쓰기에 적용된 "배경(background)" 집합에 속성 *attr*을 추가합니다.

window.**attrset**(*attr*)

속성의 "배경(background)" 집합을 *attr*로 설정합니다. 이 집합은 처음에 0입니다(속성 없음).

window.**bkgd**(*ch*[, *attr*])

창의 배경 속성을 *attr* 속성을 가진 문자 *ch*로 설정합니다. 그런 다음 해당 창의 모든 문자 위치에 변경 사항이 적용됩니다:

- 창의 모든 문자 속성이 새 배경 속성으로 변경됩니다.
- 이전 배경 문자가 나타날 때마다, 새 배경 문자로 변경됩니다.

window.**bkgdset**(*ch*[, *attr*])

창의 배경을 설정합니다. 창의 배경은 문자와 속성의 모든 조합으로 구성됩니다. 배경의 속성 부분은 창에 쓰인 모든 비 공백 문자와 결합(OR)합니다. 배경의 문자와 속성 부분은 모두 공백 문자와 결합합니다. 배경은 문자의 속성이 되고 스크롤과 줄/문자 삽입/삭제 연산을 통해 문자와 함께 이동합니다.

window.**border**([*ls*[, *rs*[, *ts*[, *bs*[, *tl*[, *tr*[, *bl*[, *br*]]]]]]]])

창의 가장자리 주위에 테두리를 그립니다. 각 매개 변수는 테두리의 특정 부분에 사용할 문자를 지정합니다; 자세한 내용은 아래 표를 참조하십시오.

> **참고**

모든 매개 변수의 0 값은 해당 매개 변수에 기본 문자가 사용되도록 합니다. 키워드 매개 변수는 사용할 수 없습니다. 기본값은 이 표에 나열되어 있습니다:

매개 변수	설명	기본값
ls	좌변	ACS_VLINE
rs	우변	ACS_VLINE
ts	상단	ACS_HLINE
bs	하단	ACS_HLINE
tl	왼쪽 위 모서리	ACS_ULCORNER
tr	오른쪽 위 모서리	ACS_URCORNER
bl	왼쪽 아래 모서리	ACS_LLCORNER
br	오른쪽 아래 모서리	ACS_LRCORNER

window.**box**([*vertch*, *horch*])

border() 와 유사하지만, *ls*와 *rs*가 모두 *vertch*이고 *ts*와 *bs*가 모두 *horch*입니다. 이 함수는 항상 기본 모서리 문자를 사용합니다.

window.**chgat**(*attr*)

window.**chgat**(*num*, *attr*)

window.**chgat**(*y*, *x*, *attr*)

window.**chgat**(*y*, *x*, *num*, *attr*)

현재 커서 위치나 제공되면 (y, x) 위치에 *num* 문자의 속성을 설정합니다. *num*이 제공되지 않거나 -1이면, 줄 끝까지의 모든 문자에 속성이 설정됩니다. 이 함수는 제공되면 커서를 (y, x) 위치로 이동합니다. 변경된 줄을 touchline() 메서드를 사용하여 터치해서 다음 창 refresh로 내용이 다시 표시됩니다.

window.**clear**()

erase()와 유사하지만, refresh()를 다음에 호출할 때 전체 창이 다시 그려집니다.

window.**clearok**(*flag*)

*flag*가 True이면, refresh()에 대한 다음 호출은 창을 완전히 지웁니다.

window.**clrtobot**()

커서에서 창끝까지 지웁니다: 커서 아래의 모든 줄이 삭제된 다음, clrtoeol()과 동등한 것이 수행됩니다.

window.**clrtoeol**()

커서에서 줄 끝까지 지웁니다.

window.**cursyncup**()

창의 현재 커서 위치를 반영하도록 창의 모든 조상의 현재 커서 위치를 갱신합니다.

window.**delch**([*y*, *x*])

(y, x)에서 문자를 삭제합니다.

window.**deleteln**()

커서 아래의 줄을 삭제합니다. 다음 줄은 모두 한 줄씩 위로 이동합니다.

window.**derwin**(*begin_y*, *begin_x*)

window.**derwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)

"창 파생(derive window)"의 약어인 derwin()은 subwin()을 호출하는 것과 같지만, *begin_y*와 *begin_x*가 전체 화면에 상대적이 아니라 창의 원점에 상대적이라는 차이가 있습니다. 파생된 창에 대한 창 객체를 반환합니다.

window.**echochar**(*ch*[, *attr*])

attr 속성을 가진 문자 *ch*를 추가하고, 즉시 창에서 refresh()를 호출합니다.

window.**enclose**(*y*, *x*)

주어진 화면 상대적인 문자 셀 좌표 쌍이 주어진 창에 속하는지를 검사하고, `True`나 `False`를 반환합니다. 마우스 이벤트의 위치를 포함하는 화면 창의 부분 집합을 결정하는 데 유용합니다.

버전 3.10에서 변경: Previously it returned `1` or `0` instead of `True` or `False`.

window.**encoding**

메서드 인자(유니코드 문자열과 문자)를 인코딩하는 데 사용되는 인코딩. encoding 어트리뷰트는 하위 창(subwindow)을 만들 때 (예를 들어 *window.subwin()*으로) 부모 창에서 상속됩니다. 기본적으로, 현재 로케일 인코딩이 사용됩니다 (*locale.getencoding()*를 참조하십시오).

Added in version 3.3.

window.**erase**()

창을 지웁니다.

window.**getbegyx**()

왼쪽 위 모서리 좌표의 튜플 `(y, x)`를 반환합니다.

window.**getbkgd**()

주어진 창의 현재 배경 문자/속성 쌍을 반환합니다.

window.**getch**([*y*, *x*])

문자를 얻습니다. 반환된 정수는 ASCII 범위일 필요가 없음에 유의하십시오: 기능키, 키패드 키 등은 255보다 큰 숫자로 표시됩니다. 지연 없는(no-delay) 모드에서, 입력이 없으면 `-1`을 반환하고, 그렇지 않으면 키가 눌릴 때까지 기다립니다.

window.**get_wch**([*y*, *x*])

와이드 문자(wide character)를 얻습니다. 대부분의 키에 대해서는 문자를 반환하고, 기능키, 키패드 키 및 기타 특수키에 대해서는 정수를 반환합니다. 지연 없는(no-delay) 모드에서, 입력이 없으면 예외를 발생시킵니다.

Added in version 3.3.

window.**getkey**([*y*, *x*])

문자를 얻습니다. *getch()*처럼 정수를 반환하는 대신 문자열을 반환합니다. 기능키, 키패드 키 및 기타 특수키는 키 이름이 포함된 멀티 바이트 문자열을 반환합니다. 지연 없는(no-delay) 모드에서, 입력이 없으면 예외를 발생시킵니다.

window.**getmaxyx**()

창의 높이와 너비의 튜플 `(y, x)`를 반환합니다.

window.**getparyx**()

부모 창에 대해 상대적인 이 창의 시작 좌표를 튜플 `(y, x)`로 반환합니다. 이 창에 부모가 없으면 `(-1, -1)`을 반환합니다.

window.**getstr**()

window.**getstr**(*n*)

window.**getstr**(*y*, *x*)

window.**getstr**(*y*, *x*, *n*)

Read a bytes object from the user, with primitive line editing capacity. The maximum value for *n* is 2047.

버전 3.14에서 변경: The maximum value for *n* was increased from 1023 to 2047.

window.**getyx**()

창의 왼쪽 위 모서리에 상대적인 현재 커서 위치의 튜플 `(y, x)`를 반환합니다.

window.**hline**(*ch*, *n*)

window.**hline**(*y*, *x*, *ch*, *n*)

문자 *ch*로 구성된 길이 *n*의 `(y, x)`에서 시작하는 수평선을 표시합니다.

window.**idcok**(*flag*)

*flag*가 False이면, curses는 더는 터미널의 하드웨어 문자 삽입/삭제 기능 사용을 고려하지 않습니다; *flag*가 True이면, 문자 삽입과 삭제 사용이 활성화됩니다. curses가 처음 초기화될 때, 기본적으로 문자 삽입/삭제 사용이 활성화됩니다.

window.**idlok**(*flag*)

*flag*가 True이면, *curses*는 하드웨어 줄 편집 기능을 시도하고 사용합니다. 그렇지 않으면, 줄 삽입/삭제가 비활성화됩니다.

window.**immedok**(*flag*)

*flag*가 True이면, 창 이미지의 모든 변경이 자동으로 창을 새로 고칩니다; 더는 *refresh()*를 직접 호출할 필요가 없습니다. 그러나, wrefresh 호출이 반복되어, 성능이 크게 저하될 수 있습니다. 이 옵션은 기본적으로 비활성화되어 있습니다.

window.**inch**([*y*, *x*])

창의 주어진 위치에 있는 문자를 반환합니다. 하위 8비트는 문자이고, 상위 비트는 속성입니다.

window.**insch**(*ch*[, *attr*])

window.**insch**(*y*, *x*, *ch*[, *attr*])

속성 *attr*로 (y, x)에 문자 *ch*를 그리면서, *x* 위치에서 한 문자씩 오른쪽으로 줄을 이동합니다.

window.**insdelln**(*nlines*)

현재 줄 위의 지정된 창에 *nlines* 줄을 삽입합니다. *nlines* 바닥 줄은 손실됩니다. 음의 *nlines*의 경우, 커서 아래에 있는 줄에서 시작하여 *nlines* 줄을 삭제하고, 나머지 줄을 위로 이동합니다. 바닥 *nlines* 줄이 지워집니다. 현재 커서 위치는 같게 유지됩니다.

window.**insertln**()

커서 아래에 빈 줄을 삽입합니다. 그다음 모든 줄은 한 줄 아래로 이동합니다.

window.**insnstr**(*str*, *n*[, *attr*])

window.**insnstr**(*y*, *x*, *str*, *n*[, *attr*])

커서 아래의 문자 앞에 최대 *n* 문자까지 문자열(줄에 맞는 최대 문자)을 삽입합니다. *n*이 0이거나 음수이면, 전체 문자열이 삽입됩니다. 커서 오른쪽의 모든 문자가 오른쪽으로 이동하고, 줄의 가장 오른쪽 문자들이 손실됩니다. 커서 위치는 변경되지 않습니다(지정되면, *y*, *x*로 이동한 후에).

window.**insstr**(*str*[, *attr*])

window.**insstr**(*y*, *x*, *str*[, *attr*])

커서 아래의 문자 앞에 문자열(줄에 맞는 최대 문자)을 삽입합니다. 커서 오른쪽의 모든 문자가 오른쪽으로 이동하고, 줄의 가장 오른쪽 문자들이 손실됩니다. 커서 위치는 변경되지 않습니다(지정되면, *y*, *x*로 이동한 후에).

window.**instr**([*n*])

window.**instr**(*y*, *x*[, *n*])

Return a bytes object of characters, extracted from the window starting at the current cursor position, or at *y*, *x* if specified. Attributes are stripped from the characters. If *n* is specified, *instr()* returns a string at most *n* characters long (exclusive of the trailing NUL). The maximum value for *n* is 2047.

버전 3.14에서 변경: The maximum value for *n* was increased from 1023 to 2047.

window.**is_linetouched**(*line*)

*refresh()*에 대한 마지막 호출 이후 지정된 줄이 수정되었으면 True를 반환합니다; 그렇지 않으면 False를 반환합니다. 주어진 창에서 *line*이 유효하지 않으면 *curses.error* 예외를 발생시킵니다.

window.**is_wintouched**()

*refresh()*에 대한 마지막 호출 이후 지정된 창이 수정되었으면 True를 반환합니다; 그렇지 않으면 False를 반환합니다.

window.**keypad**(*flag*)

*flag*가 True이면, 일부 키(키패드, 기능키)가 생성한 이스케이프 시퀀스를 *curses*가 해석합니다. *flag*가 False이면, 이스케이프 시퀀스는 입력 스트림에 그대로 남아 있습니다.

window.**leaveok**(*flag*)

*flag*가 True이면, 커서는 "커서 위치"가 아니라 갱신 중인 위치에 남아 있습니다. 가능하면 커서 이동이 줄어듭니다. 가능하면 커서가 보이지 않게 합니다.

*flag*가 False이면, 커서는 갱신 후 항상 "커서 위치"에 있게 됩니다.

window.**move**(*new_y*, *new_x*)

커서를 (new_y, new_x)로 이동합니다.

window.**mvderwin**(*y*, *x*)

창을 부모 창 내부로 이동합니다. 창의 화면에 상대적인 매개 변수는 변경되지 않습니다. 이 루틴은 부모 창의 다른 부분을 화면에서 같은 물리적 위치에 표시하는 데 사용됩니다.

window.**mvwin**(*new_y*, *new_x*)

왼쪽 위 모서리가 (new_y, new_x)가 되도록 창을 이동합니다.

window.**nodelay**(*flag*)

*flag*가 True이면, *getch()*가 비 블로킹이 됩니다.

window.**notimeout**(*flag*)

*flag*가 True이면, 이스케이프 시퀀스가 시간 초과하지 않습니다.

*flag*가 False이면, 몇 밀리초 후에, 이스케이프 시퀀스가 해석되지 않고, 그대로 입력 스트림에 남아 있습니다.

window.**noutrefresh**()

새로 고침을 표시하지만 기다립니다. 이 함수는 원하는 창의 상태를 나타내는 데이터 구조를 갱신하지만, 물리적 화면을 강제로 갱신하지는 않습니다. 이를 위해서는, *doupdate()*를 호출하십시오.

window.**overlay**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])

destwin 위에 창을 오버레이 합니다. 창의 크기가 같을 필요는 없으며, 겹치는 영역만 복사됩니다. 이 복사는 비 파괴적인데, 현재 배경 문자가 *destwin*의 이전 내용을 덮어쓰지 않습니다.

복사되는 영역을 세밀하게 제어하기 위해, *overlay()*의 두 번째 형식을 사용할 수 있습니다. *sminrow*와 *smincol*은 소스 창의 왼쪽 위 좌표이며, 다른 변수들은 대상 창의 사각형을 표시합니다.

window.**overwrite**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])

destwin 위에 창을 덮어씁니다. 창의 크기가 같을 필요는 없으며, 이 경우 겹치는 영역만 복사됩니다. 이 복사는 파괴적인데, 현재 배경 문자가 *destwin*의 이전 내용을 덮어씁니다.

복사된 영역을 세밀하게 제어하기 위해, *overwrite()*의 두 번째 형식을 사용할 수 있습니다. *sminrow*와 *smincol*은 소스 창의 왼쪽 위 좌표이며, 다른 변수들은 대상 창의 사각형을 표시합니다.

window.**putwin**(*file*)

창과 연관된 모든 데이터를 제공된 파일 객체에 기록합니다. 이 정보는 나중에 *getwin()* 함수를 사용하여 가져올 수 있습니다.

window.**redrawln**(*beg*, *num*)

beg 줄에서 시작하는 *num* 화면 줄이 손상되었으며 다음 *refresh()* 호출에서 완전히 다시 그려야 함을 나타냅니다.

window.**redrawwin**()

전체 창을 터치하여, 다음 *refresh()* 호출에서 완전히 다시 그려지도록 합니다.

window.**refresh**([*pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*])

화면을 즉시 갱신합니다 (이전 그리기/삭제 메서드와 실제 화면을 동기화합니다).

6개의 선택적 인자는 창이 *newpad()*로 만들어진 패드일 때만 지정할 수 있습니다. 추가 매개 변수는 패드와 화면의 어떤 부분이 관련되어 있는지를 나타내기 위해 필요합니다. *pminrow*와 *pmincol*은 패드에서 표시할 사각형의 왼쪽 위 모서리를 지정합니다. *sminrow*, *smincol*, *smaxrow* 및 *smaxcol*은 화면에 표시할 사각형의 변을 지정합니다. 사각형의 크기가 같아야 해서, 패드에서 표시할 사각형의 오른쪽 아래 모서리는 화면 좌표에서 계산됩니다. 두 사각형 모두 해당 구조 내에 완전히 포함되어야 합니다. *pminrow*, *pmincol*, *sminrow* 또는 *smincol*의 음수 값은 마치 0인 것처럼 처리됩니다.

window.**resize**(*nlines*, *ncols*)

curses 창의 스토리지를 재할당하여 지정된 값으로 크기를 조정합니다. 두 크기 중 하나가 현재 값보다 크면, 창의 데이터는 현재 배경 변환 (`bkgdset()`으로 설정한)을 갖는 공백으로 채워집니다.

window.**scroll**([*lines=1*])

화면이나 스크롤 영역을 *lines* 줄 위로 스크롤 합니다.

window.**scrollok**(*flag*)

맨 아래 줄에서의 줄 바꾸기나 마지막 줄의 마지막 문자 입력의 결과로, 창의 커서가 창이나 스크롤 영역의 경계를 벗어날 때 어떻게 할지를 제어합니다. *flag*가 `False`이면, 맨 아래 줄에 남습니다. *flag*가 `True`이면, 창은 한 줄 위로 스크롤 됩니다. 터미널에서 물리적 스크롤 효과를 얻으려면, `idlok()`도 호출해야 함에 유의하십시오.

window.**setscrreg**(*top*, *bottom*)

스크롤 영역을 *top* 줄에서 *bottom* 줄로 설정합니다. 모든 스크롤 작업은 이 영역에서 수행됩니다.

window.**standend**()

스탠드 아웃(standout) 속성을 끕니다. 일부 터미널에서는 모든 속성을 끄는 부작용이 있습니다.

window.**standout**()

속성 *A_STANDOUT*을 켭니다.

window.**subpad**(*begin_y*, *begin_x*)

window.**subpad**(*nlines*, *ncols*, *begin_y*, *begin_x*)

왼쪽 위 모서리가 `(begin_y, begin_x)`이고, 너비/높이가 *ncols*/*nlines*인 서브 창을 반환합니다.

window.**subwin**(*begin_y*, *begin_x*)

window.**subwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)

왼쪽 위 모서리가 `(begin_y, begin_x)`이고, 너비/높이가 *ncols*/*nlines*인 서브 창을 반환합니다.

기본적으로, 서브 창은 지정된 위치에서 창의 오른쪽 아래 모서리에 이릅니다.

window.**syncdown**()

조상 창에서 터치된 창의 각 위치를 터치합니다. 이 루틴은 `refresh()`에 의해 호출되므로, 수동으로 호출할 필요가 거의 없습니다.

window.**syncok**(*flag*)

*flag*가 `True`이면, 창에 변경 사항이 있을 때마다 `syncup()`이 자동으로 호출됩니다.

window.**syncup**()

창에서 변경된 창 조상의 모든 위치를 터치합니다.

window.**timeout**(*delay*)

창의 블로킹이나 비 블로킹 읽기 동작을 설정합니다. *delay*가 음수이면, 블로킹 읽기가 사용됩니다 (입력을 무한정 기다립니다). *delay*가 0이면, 비 블로킹 읽기가 사용되며, 대기 중인 입력이 없으면 `getch()`는 `-1`을 반환합니다. *delay*가 양수이면, `getch()`는 *delay* 밀리초 동안 블록 하고, 해당 시간이 지나도 여전히 입력이 없으면 `-1`을 반환합니다.

window.**touchline**(*start*, *count*[, *changed*])

줄 *start*로 시작하여 *count* 줄이 변경된 것으로 가정합니다. *changed*가 제공되면, 영향을 받는 줄이 변경되었다고 (*changed*=`True`) 또는 변경되지 않았다고 (*changed*=`False`) 표시할지를 지정합니다.

window.**touchwin**()

그리기 최적화를 위해, 전체 창이 변경된 것으로 가정합니다.

window.**untouchwin**()

`refresh()`를 마지막으로 호출한 후에 창의 모든 줄을 변경되지 않은 것으로 표시합니다.

window.**vline**(*ch*, *n*[, *attr*])

window.**vline**(*y*, *x*, *ch*, *n*[, *attr*])

attr 속성을 가진 문자 *ch*로 구성된 길이 *n*의 `(y, x)`에서 시작하는 세로 선을 표시합니다.

17.5.3 상수

curses 모듈은 다음 데이터 멤버를 정의합니다:

curses.**ERR**

정수를 반환하는 일부 curses 루틴은 (가령 *getch()*) 실패 시 *ERR*을 반환합니다.

curses.**OK**

정수를 반환하는 일부 curses 루틴은 (가령 *napms()*) 성공 시 *OK*를 반환합니다.

curses.**version**

curses.**__version__**

모듈의 현재 버전을 나타내는 바이트열 객체.

curses.**ncurses_version**

ncurses 라이브러리 버전의 세 가지 구성 요소를 포함하는 네임드 튜플: *major*, *minor* 및 *patch*. 모든 값은 정수입니다. 구성 요소는 이름으로도 액세스 할 수 있어서, `curses.ncurses_version[0]`은 `curses.ncurses_version.major`와 동등합니다.

가용성: ncurses 라이브러리가 사용된 경우.

Added in version 3.8.

curses.**COLORS**

The maximum number of colors the terminal can support. It is defined only after the call to *start_color()*.

curses.**COLOR_PAIRS**

The maximum number of color pairs the terminal can support. It is defined only after the call to *start_color()*.

curses.**COLS**

The width of the screen, i.e., the number of columns. It is defined only after the call to *initscr()*. Updated by *update_lines_cols()*, *resizeterm()* and *resize_term()*.

curses.**LINES**

The height of the screen, i.e., the number of lines. It is defined only after the call to *initscr()*. Updated by *update_lines_cols()*, *resizeterm()* and *resize_term()*.

문자 셀 속성을 지정하기 위해 일부 상수를 사용할 수 있습니다. 사용 가능한 정확한 상수는 시스템에 따라 다릅니다.

속성	의미
curses.**A_ALTCHARSET**	대체 문자 집합 모드
curses.**A_BLINK**	깜박임 모드
curses.**A_BOLD**	볼드 모드
curses.**A_DIM**	희미한 모드
curses.**A_INVIS**	보이지 않거나 공백 모드
curses.**A_ITALIC**	기울임 꼴 모드
curses.**A_NORMAL**	일반 속성
curses.**A_PROTECT**	보호 모드
curses.**A_REVERSE**	배경과 전경색 반전
curses.**A_STANDOUT**	눈에 띄는 모드
curses.**A_UNDERLINE**	밑줄 모드
curses.**A_HORIZONTAL**	수평 하이라이트
curses.**A_LEFT**	왼쪽 하이라이트
curses.**A_LOW**	낮은 하이라이트
curses.**A_RIGHT**	오른쪽 하이라이트
curses.**A_TOP**	상단 하이라이트
curses.**A_VERTICAL**	수직 하이라이트

Added in version 3.7: `A_ITALIC`이 추가되었습니다.

일부 메서드에서 반환한 해당 속성을 추출하기 위해 여러 상수를 사용할 수 있습니다.

비트 마스크	의미
curses.**A_ATTRIBUTES**	속성을 추출하는 비트 마스크
curses.**A_CHARTEXT**	문자를 추출하는 비트 마스크
curses.**A_COLOR**	색상 쌍 필드 정보를 추출하는 비트 마스크

키는 KEY_로 시작하는 이름을 가진 정수 상수로 참조됩니다. 사용 가능한 정확한 키 기능은 시스템에 따라 다릅니다.

키 상수	키
curses.**KEY_MIN**	최소 키값
curses.**KEY_BREAK**	브레이크 키 (신뢰할 수 없습니다)
curses.**KEY_DOWN**	아래쪽 화살표
curses.**KEY_UP**	위쪽 화살표
curses.**KEY_LEFT**	왼쪽 화살표
curses.**KEY_RIGHT**	오른쪽 화살표
curses.**KEY_HOME**	홈 키 (위쪽+왼쪽 화살표)
curses.**KEY_BACKSPACE**	백스페이스 (신뢰할 수 없습니다)
curses.**KEY_F0**	기능키. 최대 64개의 기능키가 지원됩니다.
curses.**KEY_Fn**	기능키 *n*의 값
curses.**KEY_DL**	줄 삭제
curses.**KEY_IL**	줄 삽입
curses.**KEY_DC**	문자 삭제

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

키 상수	키
curses.**KEY_IC**	문자 삽입이나 삽입 모드로 들어가기
curses.**KEY_EIC**	문자 삽입 모드 종료
curses.**KEY_CLEAR**	화면 지우기
curses.**KEY_EOS**	화면 끝까지 지우기
curses.**KEY_EOL**	줄 끝까지 지우기
curses.**KEY_SF**	한 줄 앞으로 스크롤
curses.**KEY_SR**	한 줄 뒤로 스크롤 (역)
curses.**KEY_NPAGE**	다음 페이지
curses.**KEY_PPAGE**	이전 페이지
curses.**KEY_STAB**	탭 설정
curses.**KEY_CTAB**	탭 지우기
curses.**KEY_CATAB**	모든 탭 지우기
curses.**KEY_ENTER**	엔터나 발송 (신뢰할 수 없습니다)
curses.**KEY_SRESET**	소프트(soft) (부분) 재설정 (신뢰할 수 없습니다)
curses.**KEY_RESET**	재설정이나 하드(hard) 재설정 (신뢰할 수 없습니다)
curses.**KEY_PRINT**	인쇄
curses.**KEY_LL**	홈 다운이나 바닥 (왼쪽 아래)
curses.**KEY_A1**	키패드의 왼쪽 위

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

키 상수	키
curses.**KEY_A3**	키패드의 오른쪽 위
curses.**KEY_B2**	키패드 가운데
curses.**KEY_C1**	키패드의 왼쪽 아래
curses.**KEY_C3**	키패드의 오른쪽 아래
curses.**KEY_BTAB**	백 탭
curses.**KEY_BEG**	Beg (시작)
curses.**KEY_CANCEL**	취소
curses.**KEY_CLOSE**	닫기
curses.**KEY_COMMAND**	Cmd (명령)
curses.**KEY_COPY**	복사
curses.**KEY_CREATE**	생성
curses.**KEY_END**	끝
curses.**KEY_EXIT**	종료
curses.**KEY_FIND**	찾기
curses.**KEY_HELP**	도움말
curses.**KEY_MARK**	표시
curses.**KEY_MESSAGE**	메시지
curses.**KEY_MOVE**	이동

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

키 상수	키
curses.**KEY_NEXT**	다음
curses.**KEY_OPEN**	열기
curses.**KEY_OPTIONS**	옵션
curses.**KEY_PREVIOUS**	Prev (이전)
curses.**KEY_REDO**	다시 하기
curses.**KEY_REFERENCE**	Ref (참조)
curses.**KEY_REFRESH**	새로 고침
curses.**KEY_REPLACE**	교체
curses.**KEY_RESTART**	재시작
curses.**KEY_RESUME**	재개
curses.**KEY_SAVE**	저장
curses.**KEY_SBEG**	시프트 Beg (시작)
curses.**KEY_SCANCEL**	시프트 취소
curses.**KEY_SCOMMAND**	시프트 명령
curses.**KEY_SCOPY**	시프트 복사
curses.**KEY_SCREATE**	시프트 생성
curses.**KEY_SDC**	시프트 문자 삭제
curses.**KEY_SDL**	시프트 줄 삭제

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

키 상수	키
curses.**KEY_SELECT**	선택
curses.**KEY_SEND**	시프트 끝
curses.**KEY_SEOL**	시프트 줄 지우기
curses.**KEY_SEXIT**	시프트 종료
curses.**KEY_SFIND**	시프트 찾기
curses.**KEY_SHELP**	시프트 도움말
curses.**KEY_SHOME**	시프트 홈
curses.**KEY_SIC**	시프트 입력
curses.**KEY_SLEFT**	시프트 왼쪽 화살표
curses.**KEY_SMESSAGE**	시프트 메시지
curses.**KEY_SMOVE**	시프트 이동
curses.**KEY_SNEXT**	시프트 다음
curses.**KEY_SOPTIONS**	시프트 옵션
curses.**KEY_SPREVIOUS**	시프트 Prev
curses.**KEY_SPRINT**	시프트 인쇄
curses.**KEY_SREDO**	시프트 다시 하기
curses.**KEY_SREPLACE**	시프트 교체
curses.**KEY_SRIGHT**	시프트 오른쪽 화살표

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

키 상수	키
curses.**KEY_SRSUME**	시프트 재개
curses.**KEY_SSAVE**	시프트 저장
curses.**KEY_SSUSPEND**	시프트 일시 중단
curses.**KEY_SUNDO**	시프트 실행 취소
curses.**KEY_SUSPEND**	일시 중단
curses.**KEY_UNDO**	실행 취소
curses.**KEY_MOUSE**	마우스 이벤트가 발생했습니다
curses.**KEY_RESIZE**	터미널 크기 조정 이벤트
curses.**KEY_MAX**	최대 키값

VT100과 이의 소프트웨어 에뮬레이션(가령 X 터미널 에뮬레이터)에는, 일반적으로 사용 가능한 기능키가 적어도 4개 (*KEY_F1*, *KEY_F2*, *KEY_F3*, *KEY_F4*) 있고, 화살표 키는 *KEY_UP*, *KEY_DOWN*, *KEY_LEFT* 및 *KEY_RIGHT*에 명백한 방식으로 매핑됩니다. 여러분의 기계에 PC 키보드가 있으면, 화살표 키와 12개의 기능키를 기대하는 것이 안전합니다 (이전 PC 키보드에는 10개의 기능키만 있을 수 있습니다); 또한, 다음과 같은 키패드 매핑이 표준입니다:

키 기능	상수
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

다음 표는 대체 문자 집합의 문자를 나열합니다. 이들은 VT100 터미널에서 상속되며, 일반적으로 X 터미널과 같은 소프트웨어 에뮬레이션에서 사용할 수 있습니다. 사용 가능한 그래픽이 없으면, curses는 조잡한 인쇄 가능한 ASCII 근사치로 폴 백 합니다.

참고

*initscr()*가 호출된 후에만 사용할 수 있습니다.

ACS 코드	의미
curses.**ACS_BBSS**	오른쪽 위 모서리의 대체 이름
curses.**ACS_BLOCK**	채워진 사각형 블록
curses.**ACS_BOARD**	사각형의 보드
curses.**ACS_BSBS**	수평선의 대체 이름
curses.**ACS_BSSB**	왼쪽 위 모서리의 대체 이름
curses.**ACS_BSSS**	상단 티(top tee)의 대체 이름
curses.**ACS_BTEE**	하단 티(bottom tee)
curses.**ACS_BULLET**	불릿
curses.**ACS_CKBOARD**	체커 보드 (점각)
curses.**ACS_DARROW**	아래쪽을 가리키는 화살표
curses.**ACS_DEGREE**	디그리 기호
curses.**ACS_DIAMOND**	다이아몬드
curses.**ACS_GEQUAL**	크거나 같음
curses.**ACS_HLINE**	수평선
curses.**ACS_LANTERN**	랜턴 기호
curses.**ACS_LARROW**	왼쪽 화살표
curses.**ACS_LEQUAL**	작거나 같음
curses.**ACS_LLCORNER**	왼쪽 아래 모서리

다음 페이지에 계속

표 2 – 이전 페이지에서 계속

ACS 코드	의미
curses.**ACS_LRCORNER**	오른쪽 아래 모서리
curses.**ACS_LTEE**	왼쪽 티(left tee)
curses.**ACS_NEQUAL**	같지 않음 기호
curses.**ACS_PI**	글자 파이(pi)
curses.**ACS_PLMINUS**	더하기-또는-빼기 기호
curses.**ACS_PLUS**	큰 더하기 기호
curses.**ACS_RARROW**	오른쪽 화살표
curses.**ACS_RTEE**	오른쪽 티(right tee)
curses.**ACS_S1**	스캔 줄 1
curses.**ACS_S3**	스캔 줄 3
curses.**ACS_S7**	스캔 줄 7
curses.**ACS_S9**	스캔 줄 9
curses.**ACS_SBBS**	오른쪽 아래 모서리의 대체 이름
curses.**ACS_SBSB**	세로줄의 대체 이름
curses.**ACS_SBSS**	오른쪽 티의 대체 이름
curses.**ACS_SSBB**	왼쪽 아래 모서리의 대체 이름
curses.**ACS_SSBS**	하단 티의 대체 이름
curses.**ACS_SSSB**	왼쪽 티의 대체 이름

다음 페이지에 계속

표 2 – 이전 페이지에서 계속

ACS 코드	의미
curses.**ACS_SSSS**	크로스 오버(crossover) 또는 큰 더하기의 대체 이름
curses.**ACS_STERLING**	파운드 스털링(pound sterling)
curses.**ACS_TTEE**	상단 티(top tee)
curses.**ACS_UARROW**	위쪽 화살표
curses.**ACS_ULCORNER**	왼쪽 위 모서리
curses.**ACS_URCORNER**	오른쪽 위 모서리
curses.**ACS_VLINE**	수직선

The following table lists mouse button constants used by `getmouse()`:

마우스 버튼 상수	의미
curses.**BUTTONn_PRESSED**	Mouse button *n* pressed
curses.**BUTTONn_RELEASED**	Mouse button *n* released
curses.**BUTTONn_CLICKED**	Mouse button *n* clicked
curses.**BUTTONn_DOUBLE_CLICKED**	Mouse button *n* double clicked
curses.**BUTTONn_TRIPLE_CLICKED**	Mouse button *n* triple clicked
curses.**BUTTON_SHIFT**	Shift was down during button state change
curses.**BUTTON_CTRL**	Control was down during button state change
curses.**BUTTON_ALT**	Control was down during button state change

버전 3.10에서 변경: The `BUTTON5_*` constants are now exposed if they are provided by the underlying curses library.

다음 표는 사전 정의된 색상을 나열합니다:

상수	색상
curses.**COLOR_BLACK**	검은색
curses.**COLOR_BLUE**	파랑
curses.**COLOR_CYAN**	시안 (청록색 - 밝은 초록이 섞인 파랑)
curses.**COLOR_GREEN**	초록
curses.**COLOR_MAGENTA**	마젠타 (자홍색)
curses.**COLOR_RED**	빨강
curses.**COLOR_WHITE**	흰색
curses.**COLOR_YELLOW**	노랑

17.6 curses.textpad — curses 프로그램을 위한 텍스트 입력 위젯

curses.textpad 모듈은 curses 창에서 기본 텍스트 편집을 처리하는 *Textbox* 클래스를 제공하며, Emacs 와 유사한 일련의 키 바인딩을 지원합니다 (따라서 Netscape Navigator, BBedit 6.x, FrameMaker 및 기타 여러 프로그램과도 유사한). 이 모듈은 텍스트 상자의 틀이나 다른 목적에 유용한 사각형 그리기 함수도 제공합니다.

curses.textpad 모듈은 다음 함수를 정의합니다:

curses.textpad.**rectangle**(*win, uly, ulx, lry, lrx*)

직사각형을 그립니다. 첫 번째 인자는 창 객체여야 합니다; 나머지 인자는 그 창에 상대적인 좌표입니다. 두 번째와 세 번째 인자는 그릴 사각형의 왼쪽 위 모서리의 y와 x 좌표입니다; 네 번째와 다섯 번째 인자는 오른쪽 아래 모서리의 y와 x 좌표입니다. 사각형은 이것이 가능한 터미널(xterm 과 대부분의 다른 소프트웨어 터미널 에뮬레이터를 포함합니다)에서 VT100/IBM PC 양식 문자를 사용하여 그려집니다. 그렇지 않으면 ASCII 대시, 세로 막대 및 더하기 기호로 그려집니다.

17.6.1 Textbox 객체

다음과 같이 *Textbox* 객체를 인스턴스 화 할 수 있습니다:

class curses.textpad.**Textbox**(*win*)

텍스트 상자 위젯 객체를 반환합니다. *win* 인자는 텍스트 상자가 포함될 curses 창 객체여야 합니다. 텍스트 상자의 편집 커서는 처음에 좌표 (0, 0)으로 포함하는 창의 왼쪽 위 모서리에 있습니다. 인스턴스의 *stripspaces* 플래그가 처음에 켜집니다.

Textbox 객체에는 다음과 같은 메서드가 있습니다:

edit([*validator*])

이것이 일반적으로 사용하는 진입점입니다. 종료 키 입력 중 하나를 입력할 때까지 편집 키 입력을 받아들입니다. *validator*가 제공되면, 반드시 함수여야 합니다. 입력한 각 키 입력에 대해 키 입력을 매개 변수로 호출됩니다; 명령 디스패치가 그 결과에 대해 수행됩니다. 이 메서드는

창 내용을 문자열로 반환합니다; 창의 공백이 포함되는지는 *stripspaces* 어트리뷰트의 영향을 받습니다.

do_command(*ch*)

단일 명령 키 입력을 처리합니다. 지원되는 특수키 입력은 다음과 같습니다:

키 입력	동작
Control-A	창의 왼쪽 가장자리로 이동합니다.
Control-B	커서를 왼쪽으로 옮깁니다, 필요하면 앞줄로 넘어갑니다.
Control-D	커서 아래의 문자를 삭제합니다.
Control-E	오른쪽 가장자리(stripspaces가 꺼졌을 때)나 줄 끝(stripspaces가 켜졌을 때)으로 이동합니다.
Control-F	커서를 오른쪽으로 옮깁니다, 필요하면 다음 줄로 넘어갑니다.
Control-G	종료하고, 창 내용을 반환합니다.
Control-H	문자를 뒤로 삭제합니다.
Control-J	창이 한 줄이면 종료하고, 그렇지 않으면 줄 바꿈을 삽입합니다.
Control-K	줄이 비어 있으면, 삭제하고, 그렇지 않으면 줄 끝까지 지웁니다.
Control-L	화면을 새로 고칩니다.
Control-N	커서를 아래로 옮깁니다; 한 줄 아래로 이동합니다.
Control-O	커서 위치에 빈 줄을 삽입합니다.
Control-P	커서를 위로 옮깁니다; 한 줄 위로 이동합니다.

이동이 불가능한 가장자리에 커서가 있으면 이동 연산이 수행되지 않습니다. 가능하면 다음 동의어가 지원됩니다:

상수	키 입력
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

다른 모든 키 입력은 주어진 문자를 삽입하고 (줄 넘김을 포함한) 오른쪽으로 이동하는 명령으로 처리됩니다.

gather()

창 내용을 문자열로 반환합니다; 창의 공백이 포함되는지는 *stripspaces* 멤버의 영향을 받습니다.

stripspaces

이 어트리뷰트는 창의 공백 해석을 제어하는 플래그입니다. 켜져 있으면, 각 줄의 후행 공백이 무시됩니다; 후행 공백에 커서를 놓는 커서 동작은 대신 해당 줄의 끝으로 이동하고, 창 내용이 수집될 때 후행 공백이 제거됩니다.

17.7 curses.ascii — ASCII 문자용 유틸리티

소스 코드: Lib/curses/ascii.py

---

curses.ascii 모듈은 ASCII 문자에 대한 이름 상수와 다양한 ASCII 문자 클래스에서 멤버십을 검사하는 함수를 제공합니다. 제공된 상수는 다음과 같이 제어 문자의 이름입니다:

이름	의미
curses.ascii.**NUL**	
curses.ascii.**SOH**	헤딩의 시작(Start of heading), 콘솔 인터럽트
curses.ascii.**STX**	텍스트의 시작(Start of text)
curses.ascii.**ETX**	텍스트 끝(End of text)
curses.ascii.**EOT**	전송 끝(End of transmission)
curses.ascii.**ENQ**	문의(Enquiry), *ACK* 흐름 제어와 함께 제공
curses.ascii.**ACK**	확인(Acknowledgement)
curses.ascii.**BEL**	벨(Bell)
curses.ascii.**BS**	백스페이스(Backspace)
curses.ascii.**TAB**	탭(Tab)
curses.ascii.**HT**	*TAB*의 별칭: "가로 탭(Horizontal tab)"
curses.ascii.**LF**	줄 바꿈(Line feed)
curses.ascii.**NL**	*LF*의 별칭: "새 줄(New line)"
curses.ascii.**VT**	세로 탭(Vertical tab)
curses.ascii.**FF**	용지 공급(Form feed)
curses.ascii.**CR**	캐리지 리턴(Carriage return)

다음 페이지에 계속

표 3 – 이전 페이지에서 계속

이름	의미
curses.ascii.**SO**	시프트 아웃(Shift-out), 대체 문자 집합 시작
curses.ascii.**SI**	시프트 인(Shift-in), 기본 문자 집합 재개
curses.ascii.**DLE**	데이터 링크 이스케이프(Data-link escape)
curses.ascii.**DC1**	XON, 흐름 제어용
curses.ascii.**DC2**	장치 제어(Device control) 2, 블록 모드 흐름 제어
curses.ascii.**DC3**	XOFF, 흐름 제어용
curses.ascii.**DC4**	장치 제어(Device control) 4
curses.ascii.**NAK**	부정적 확인(Negative acknowledgement)
curses.ascii.**SYN**	동기 대기(Synchronous idle)
curses.ascii.**ETB**	전송 블록 종료(End transmission block)
curses.ascii.**CAN**	취소(Cancel)
curses.ascii.**EM**	매체의 끝(End of medium)
curses.ascii.**SUB**	치환(Substitute)
curses.ascii.**ESC**	탈출(Escape)
curses.ascii.**FS**	파일 구분자(File separator)
curses.ascii.**GS**	그룹 구분자(Group separator)
curses.ascii.**RS**	레코드 구분자(Record separator), 블록 모드 종료자
curses.ascii.**US**	단위 구분자(Unit separator)

다음 페이지에 계속

표 3 – 이전 페이지에서 계속

이름	의미
curses.ascii.**SP**	스페이스(Space)
curses.ascii.**DEL**	삭제(Delete)

이들 중 많은 것들이 현대적인 사용법에서 실질적인 중요성을 거의 가지고 있지 않습니다. 니모닉은 디지털 컴퓨터 이전의 텔레 프린터 규칙에서 파생되었습니다.

이 모듈은 표준 C 라이브러리의 함수에 따라 다음 함수를 제공합니다:

curses.ascii.**isalnum**(*c*)
ASCII 영숫자를 확인합니다; `isalpha(c) or isdigit(c)`와 동등합니다.

curses.ascii.**isalpha**(*c*)
ASCII 알파벳 문자를 확인합니다. `isupper(c) or islower(c)`와 동등합니다.

curses.ascii.**isascii**(*c*)
7비트 ASCII 집합에 맞는 문자 값을 확인합니다.

curses.ascii.**isblank**(*c*)
ASCII 공백 문자를 확인합니다; 스페이스나 가로 탭.

curses.ascii.**iscntrl**(*c*)
ASCII 제어 문자를 확인합니다 (0x00에서 0x1f 범위에 있거나 0x7f).

curses.ascii.**isdigit**(*c*)
ASCII 십진 숫자, `'0'` 에서 `'9'`를 확인합니다. 이것은 `c in string.digits`와 동등합니다.

curses.ascii.**isgraph**(*c*)
스페이스를 제외한 인쇄 가능한 ASCII 문자를 확인합니다.

curses.ascii.**islower**(*c*)
ASCII 소문자를 확인합니다.

curses.ascii.**isprint**(*c*)
스페이스를 포함하여 인쇄 가능한 ASCII 문자를 확인합니다.

curses.ascii.**ispunct**(*c*)
스페이스나 영숫자가 아닌 인쇄 가능한 ASCII 문자를 확인합니다.

curses.ascii.**isspace**(*c*)
ASCII 공백 문자를 확인합니다; 스페이스, 줄 바꿈, 캐리지 리턴, 용지 공급, 가로 탭, 세로 탭.

curses.ascii.**isupper**(*c*)
ASCII 대문자를 확인합니다.

curses.ascii.**isxdigit**(*c*)
ASCII 16진수를 확인합니다. 이것은 `c in string.hexdigits`와 동등합니다.

curses.ascii.**isctrl**(*c*)
ASCII 제어 문자를 확인합니다 (정숫값 0에서 31).

curses.ascii.**ismeta**(*c*)
비 ASCII 문자를 확인합니다 (0x80 이상의 정수 값).

이 함수는 정수나 단일 문자 문자열을 받아들입니다; 인자가 문자열이면, 내장 함수 `ord()`를 사용하여 먼저 변환됩니다.

이 모든 함수는 전달한 문자열의 문자에서 파생된 서수 비트 값을 확인합니다; 호스트 기계의 문자 인코딩에 대해서는 실제로 아무것도 모릅니다.

다음 두 함수는 단일 문자 문자열이나 정수 바이트 값을 취합니다; 이들은 같은 유형의 값을 반환합니다.

curses.ascii.**ascii**(*c*)

*c*의 하위 7비트에 해당하는 ASCII 값을 반환합니다.

curses.ascii.**ctrl**(*c*)

주어진 문자에 해당하는 제어 문자를 반환합니다 (문자 비트 값은 0x1f와 비트별 and 됩니다).

curses.ascii.**alt**(*c*)

주어진 ASCII 문자에 해당하는 8비트 문자를 반환합니다 (문자 비트 값은 0x80과 비트별 or 됩니다).

다음 함수는 단일 문자 문자열이나 정숫값을 취합니다; 문자열을 반환합니다.

curses.ascii.**unctrl**(*c*)

ASCII 문자 *c*의 문자열 표현을 반환합니다. *c*가 인쇄 가능하면, 이 문자열은 문자 자체입니다. 문자가 제어 문자(0x00–0x1f) 이면 문자열은 캐럿('^')과 그 뒤에 오는 해당 대문자로 구성됩니다. 문자가 ASCII 삭제(0x7f) 이면 문자열은 '^?'입니다. 문자에 메타 비트(0x80) 가 설정되어 있으면, 메타 비트가 제거되고, 앞의 규칙을 적용한 후, '!'를 결과 앞에 붙입니다.

curses.ascii.**controlnames**

0(NUL) 에서 0x1f(US) 까지 32개의 ASCII 제어 문자에 대한 ASCII 니모닉과 (순서대로), 스페이스 문자를 위한 니모닉 SP를 포함하는 33요소 문자열 배열입니다.

17.8 curses.panel — curses 용 패널 스택 확장

패널은 깊이 기능이 추가된 창이라서, 서로의 위에 쌓을 수 있으며, 각 창의 보이는 부분만 표시됩니다. 패널을 추가하고, 스택에서 위나 아래로 옮기고, 제거할 수 있습니다.

17.8.1 함수

curses.panel 모듈은 다음 함수를 정의합니다:

curses.panel.**bottom_panel**()

패널 스택에서 최하단 패널을 반환합니다.

curses.panel.**new_panel**(*win*)

주어진 창 *win*과 연관 지어진 패널 객체를 반환합니다. 반환된 패널 객체가 명시적으로 참조되도록 유지해야 합니다. 그렇지 않으면 패널 객체는 가비지 수집되어 패널 스택에서 제거됩니다.

curses.panel.**top_panel**()

패널 스택의 최상단 패널을 반환합니다.

curses.panel.**update_panels**()

패널 스택이 변경된 후 가상 화면을 갱신합니다. 이것은 *curses.doupdate()*를 호출하지 않아서, 여러분이 직접 해야 합니다.

17.8.2 Panel 객체

위의 *new_panel()*에 의해 반환된 패널 객체는 쌓인 순서가 있는 창입니다. 패널과 연관된 창이 항상 있고, 창이 내용을 결정합니다. 패널 메서드는 패널 스택에서 창의 깊이를 담당합니다.

패널 객체에는 다음과 같은 메서드가 있습니다:

Panel.**above**()

현재 패널 위의 패널을 반환합니다.

Panel.**below**()

현재 패널 아래의 패널을 반환합니다.

Panel.**bottom**()

패널을 스택 맨 아래로 밉니다."

Panel.**hidden**()

패널이 숨겨져 있으면 (보이지 않으면) `True`를, 그렇지 않으면 `False`를 반환합니다.

Panel.**hide**()

패널을 숨깁니다. 이것은 객체를 삭제하지 않고, 화면의 창을 보이지 않게 합니다.

Panel.**move**(*y*, *x*)

패널을 화면 좌표 `(y, x)`로 이동합니다.

Panel.**replace**(*win*)

패널과 연관된 창을 창 *win*으로 변경합니다.

Panel.**set_userptr**(*obj*)

패널의 사용자 포인터를 *obj*로 설정합니다. 이것은 임의의 데이터를 패널과 연관시키는 데 사용되며, 임의의 파이썬 객체가 될 수 있습니다.

Panel.**show**()

(숨겼을 수도 있는) 패널을 표시합니다.

Panel.**top**()

패널을 스택 맨 위로 밉니다.

Panel.**userptr**()

패널의 사용자 포인터를 반환합니다. 이것은 임의의 파이썬 객체일 수 있습니다.

Panel.**window**()

패널과 연관된 창 객체를 반환합니다.

17.9 cmd — 줄 지향 명령 인터프리터 지원

소스 코드: Lib/cmd.py

Cmd 클래스는 줄 지향 명령 인터프리터를 작성하기 위한 간단한 프레임워크를 제공합니다. 이것들은 종종 테스트 하네스(test harnesses), 관리 도구 및 나중에 더 복잡한 인터페이스로 포장될 프로토타입에 유용합니다.

class cmd.**Cmd**(*completekey='tab', stdin=None, stdout=None*)

Cmd 인스턴스나 서브 클래스 인스턴스는 줄 지향 인터프리터 프레임워크입니다. *Cmd* 자체를 인스턴스로 만들 이유는 없습니다; 그보다는, *Cmd*의 메서드를 상속하고 액션 메서드를 캡슐화하기 위해 여러분 스스로 정의한 인터프리터 클래스의 슈퍼 클래스로 유용합니다.

선택적 인자 *completekey*는 완성 키(completion key)의 *readline* 이름입니다; 기본값은 `Tab`입니다. *completekey*가 *None*이 아니고 *readline*을 사용할 수 있으면, 명령 완성이 자동으로 수행됩니다.

The default, `'tab'`, is treated specially, so that it refers to the `Tab` key on every *readline.backend*. Specifically, if *readline.backend* is `editline`, `Cmd` will use `'^I'` instead of `'tab'`. Note that other values are not treated this way, and might only work with a specific backend.

선택적 인자 *stdin*과 *stdout*은 Cmd 인스턴스나 서브 클래스 인스턴스가 입출력에 사용할 입력과 출력 파일 객체를 지정합니다. 지정하지 않으면, 기본적으로 *sys.stdin*과 *sys.stdout*이 됩니다.

지정된 *stdin*을 사용하려면, 인스턴스의 *use_rawinput* 어트리뷰트를 `False`로 설정해야 합니다, 그렇지 않으면 *stdin*이 무시됩니다.

버전 3.13에서 변경: `completekey='tab'` is replaced by `'^I'` for `editline`.

17.9.1 Cmd 객체

Cmd 인스턴스에는 다음과 같은 메서드가 있습니다:

Cmd.**cmdloop**(*intro=None*)

반복해서 프롬프트를 제시하고, 입력을 받아들이고, 수신된 입력에서 초기 접두사를 구문 분석하고, 줄의 나머지 부분을 인자로 전달해서 액션 메서드를 호출합니다.

선택적 인자는 첫 번째 프롬프트 전에 제시할 배너나 소개 문자열입니다 (이것은 *intro* 클래스 어트리뷰트를 재정의합니다).

readline 모듈이 로드되면, 입력은 자동으로 **bash**와 유사한 히스토리 목록 편집을 상속합니다 (예를 들어, `Control-P`는 직전 명령으로 돌아가고 (scroll back), `Control-N`은 다음 명령으로 이동하고 (forward), `Control-F`는 커서를 비파괴적으로 오른쪽으로 이동하고, `Control-B`는 커서를 비파괴적으로 왼쪽으로 이동하고, 등등).

입력의 파일 끝 (end-of-file)은 문자열 `'EOF'`로 전달됩니다.

인터프리터 인스턴스는 메서드 `do_foo()`가 있을 때만 명령 이름 `foo`를 인식합니다. 특수한 경우로, 문자 `'?'`로 시작하는 줄은 메서드 *do_help()*를 호출합니다. 또 다른 특수한 경우로, 문자 `'!'`로 시작하는 줄은 메서드 `do_shell()`을 (해당 메서드가 정의되었다면) 호출합니다.

이 메서드는 *postcmd()* 메서드가 참값을 반환할 때 반환합니다. *postcmd()*에 대한 *stop* 인자는 명령의 해당 `do_*()` 메서드에서 반환되는 값입니다.

완성 (completion)이 활성화되면, 명령 완성이 자동으로 수행되고, 명령 인자의 완성은 인자 *text*, *line*, *begidx* 및 *endidx*로 `complete_foo()`를 호출하여 수행됩니다. *text*는 일치시키려는 문자열 접두사입니다: 반환된 모든 일치는 이 문자열로 시작해야 합니다. *line*은 선행 공백이 제거된 현재 입력 줄이며, *begidx*와 *endidx*는 접두사 텍스트의 시작과 끝 인덱스로, 인자의 위치에 따라 다른 완성을 제공하는 데 사용될 수 있습니다.

Cmd.**do_help**(*arg*)

*Cmd*의 모든 서브 클래스는 미리 정의된 `do_help()`를 상속합니다. 인자 `'bar'`로 호출되면, 이 메서드는 해당 메서드 `help_bar()`를 호출하고, 존재하지 않으면 `do_bar()`의 독스트링이 있다면 인쇄합니다. 인자가 없으면, `do_help()`는 사용 가능한 모든 도움말 주제 (즉, 해당 `help_*()` 메서드가 있거나 독스트링이 있는 모든 명령)을 나열하고, 설명이 없는 명령도 나열합니다.

Cmd.**onecmd**(*str*)

프롬프트에 대한 응답으로 입력된 것처럼 인자를 해석합니다. 재정의될 수도 있지만, 일반적으로 그럴 필요가 없어야 합니다; 유용한 실행 훅에 대해서는 *precmd()*와 *postcmd()* 메서드를 참조하십시오. 반환 값은 인터프리터의 명령 해석이 중지되어야 하는지를 나타내는 플래그입니다. 명령 *str*을 위한 `do_*()` 메서드가 있으면, 해당 메서드의 반환 값이 반환되고, 그렇지 않으면 *default()* 메서드의 반환 값이 반환됩니다.

Cmd.**emptyline**()

프롬프트에 응답하여 빈 줄을 입력할 때 호출되는 메서드. 이 메서드를 재정의하지 않으면, 입력된 마지막 비어 있지 않은 명령을 반복합니다.

Cmd.**default**(*line*)

명령 접두사가 인식되지 않을 때 입력 줄로 호출되는 메서드. 이 메서드를 재정의하지 않으면, 에러 메시지를 인쇄하고 반환합니다.

Cmd.**completedefault**(*text*, *line*, *begidx*, *endidx*)

명령 별 `complete_*()` 메서드가 없을 때 입력 줄을 완성하기 위해 호출되는 메서드. 기본적으로, 빈 리스트를 반환합니다.

Cmd.**columnize**(*list*, *displaywidth=80*)

Method called to display a list of strings as a compact set of columns. Each column is only as wide as necessary. Columns are separated by two spaces for readability.

Cmd.**precmd**(*line*)

명령 줄 *line*을 해석하기 직전에, 하지만 입력 프롬프트가 생성되고 제시된 후에 실행되는 훅 메서드. 이 메서드는 *Cmd*에서는 스텁 (stub) 입니다; 서브 클래스에 의해 재정의되기 위해 존재합니다. 반환

값은 *onecmd()* 메서드에 의해 실행될 명령으로 사용됩니다; *precmd()* 구현은 명령을 다시 쓰거나 단순히 *line*을 변경하지 않고 반환 할 수 있습니다.

Cmd.**postcmd**(*stop*, *line*)

명령 호출이 완료된 직후에 실행되는 훅 메서드. 이 메서는 *Cmd*에서는 스텁(stub) 입니다; 서브 클래스에 의해 재정의되기 위해 존재합니다. *line*은 실행된 명령 줄이고, *stop*은 *postcmd()*를 호출한 후 실행이 종료될지를 나타내는 플래그입니다; 이것은 *onecmd()* 메서드의 반환 값입니다. 이 메서드의 반환 값은 *stop*에 해당하는 내부 플래그의 새 값으로 사용됩니다; 거짓을 반환하면 해석이 계속됩니다.

Cmd.**preloop**()

*cmdloop()*가 호출될 때 한 번 실행되는 훅 메서드. 이 메서드는 *Cmd*에서는 스텁(stub) 입니다; 서브 클래스에 의해 재정의되기 위해 존재합니다.

Cmd.**postloop**()

*cmdloop()*가 반환하려고 할 때 한 번 실행되는 훅 메서드. 이 메서드는 *Cmd*에서는 스텁(stub) 입니다; 서브 클래스에 의해 재정의되기 위해 존재합니다.

Cmd 서브 클래스의 인스턴스에는 몇 가지 공용 인스턴스 변수가 있습니다:

Cmd.**prompt**

입력을 요청하는 프롬프트.

Cmd.**identchars**

명령 접두사에 허용되는 문자들의 문자열.

Cmd.**lastcmd**

비어 있지 않은 마지막 명령 접두사.

Cmd.**cmdqueue**

계류 중인 입력 줄의 리스트. cmdqueue 리스트는 새로운 입력이 필요할 때 *cmdloop()*에서 점검됩니다; 비어 있지 않으면, 프롬프트에서 입력한 것처럼 해당 요소가 순서대로 처리됩니다.

Cmd.**intro**

소개나 배너로 제시할 문자열. *cmdloop()* 메서드에 인자를 제공하여 재정의할 수 있습니다.

Cmd.**doc_header**

도움말 출력에 설명된 명령 섹션이 있을 때 제시할 헤더입니다.

Cmd.**misc_header**

도움말 출력에 기타 도움말 주제에 대한 섹션이 있을 때 제시할 헤더 (즉, 해당 `do_*()` 메서드가 없는 `help_*()` 메서드가 있을 때).

Cmd.**undoc_header**

도움말 출력에 설명되지 않은 명령에 대한 섹션이 있을 때 제시할 헤더 (즉, 해당 `help_*()` 메서드가 없는 `do_*()` 메서드가 있을 때).

Cmd.**ruler**

도움말 메시지 헤더 아래에 구분선을 그리는 데 사용되는 문자입니다. 비어 있으면, 눈금자 선이 그려지지 않습니다. 기본값은 `'='` 입니다.

Cmd.**use_rawinput**

기본값이 참인 플래그. 참이면, *cmdloop()*는 *input()*을 사용하여 프롬프트를 표시하고 다음 명령을 읽습니다; 거짓이면, *sys.stdout.write()*와 *sys.stdin.readline()*이 사용됩니다. (이는 지원하는 시스템에서 *readline*를 임포트 함으로써, 인터프리터가 **Emacs**와 유사한 줄 편집과 명령 히스토리 키 입력을 자동으로 지원한다는 의미입니다.)

17.9.2 Cmd 예

cmd 모듈은 주로 사용자가 대화식으로 프로그램을 사용할 수 있도록 하는 사용자 정의 셸을 만드는 데 유용합니다.

이 섹션에서는 *turtle* 모듈의 몇 가지 명령을 중심으로 셸을 작성하는 방법에 대한 간단한 예를 제공합니다.

*forward()*와 같은 기본 turtle 명령은 `do_forward()` 라는 메서드로 *Cmd* 서브 클래스에 추가됩니다. 인자는 숫자로 변환되어 turtle 모듈로 전달됩니다. 독스트링은 셸에서 제공하는 도움말 유틸리티에서 사용됩니다.

이 예제에는 *precmd()* 메서드로 구현된 기초적인 녹화와 재생 기능도 포함되는데, 이 메서드는 입력을 소문자로 변환하고 명령을 파일에 쓰는 역할을 합니다. `do_playback()` 메서드는 파일을 읽고 즉시 재생하기 위해 녹화된 명령을 *cmdqueue*에 추가합니다:

```
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
    intro = 'Welcome to the turtle shell.   Type help or ? to list commands.\n'
    prompt = '(turtle) '
    file = None

    # ----- basic turtle commands -----
    def do_forward(self, arg):
        'Move the turtle forward by the specified distance:  FORWARD 10'
        forward(*parse(arg))
    def do_right(self, arg):
        'Turn turtle right by given number of degrees:  RIGHT 20'
        right(*parse(arg))
    def do_left(self, arg):
        'Turn turtle left by given number of degrees:  LEFT 90'
        left(*parse(arg))
    def do_goto(self, arg):
        'Move turtle to an absolute position with changing orientation.  GOTO 100␣
↪200'
        goto(*parse(arg))
    def do_home(self, arg):
        'Return turtle to the home position:  HOME'
        home()
    def do_circle(self, arg):
        'Draw circle with given radius an options extent and steps:  CIRCLE 50'
        circle(*parse(arg))
    def do_position(self, arg):
        'Print the current turtle position:  POSITION'
        print('Current position is %d %d\n' % position())
    def do_heading(self, arg):
        'Print the current turtle heading in degrees:  HEADING'
        print('Current heading is %d\n' % (heading(),))
    def do_color(self, arg):
        'Set the color:  COLOR BLUE'
        color(arg.lower())
    def do_undo(self, arg):
        'Undo (repeatedly) the last turtle action(s):  UNDO'
    def do_reset(self, arg):
        'Clear the screen and return turtle to center:  RESET'
        reset()
    def do_bye(self, arg):
        'Stop recording, close the turtle window, and exit:  BYE'
        print('Thank you for using Turtle')
        self.close()
        bye()
        return True

    # ----- record and playback -----
    def do_record(self, arg):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        'Save future commands to filename:  RECORD rose.cmd'
        self.file = open(arg, 'w')
    def do_playback(self, arg):
        'Playback commands from a file:  PLAYBACK rose.cmd'
        self.close()
        with open(arg) as f:
            self.cmdqueue.extend(f.read().splitlines())
    def precmd(self, line):
        line = line.lower()
        if self.file and 'playback' not in line:
            print(line, file=self.file)
        return line
    def close(self):
        if self.file:
            self.file.close()
            self.file = None

def parse(arg):
    'Convert a series of zero or more numbers to an argument tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()
```

다음은 도움말 기능과, 명령을 반복하기 위해 빈 줄을 사용하는 방법과, 간단한 녹화와 재생기능을 보여주기 위해 turtle 셸을 사용한 예제 세션입니다:

```
Welcome to the turtle shell.   Type help or ? to list commands.

(turtle) ?

Documented commands (type help <topic>):
========================================
bye     color    goto     home  playback  record  right
circle  forward  heading  left  position  reset   undo

(turtle) help forward
Move the turtle forward by the specified distance:  FORWARD 10
(turtle) record spiral.cmd
(turtle) position
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
(turtle) circle 120
(turtle) right 30
(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0

Current heading is 0

Current heading is 180

(turtle) bye
Thank you for using Turtle
```

CHAPTER 18

동시 실행

이 장에서 설명하는 모듈은 코드의 동시 실행을 지원합니다. 적절한 도구 선택은 실행할 작업(CPU 병목 대 IO 병목)과 선호하는 개발 스타일(이벤트 구동 협력적 다중작업 대 선점적 다중작업)에 따라 달라집니다. 다음은 개요입니다:

18.1 threading — 스레드 기반 병렬 처리

소스 코드: Lib/threading.py

이 모듈은 저수준 *_thread* 모듈 위에 고수준 스레딩 인터페이스를 구축합니다.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

18.1.1 Introduction

The `threading` module provides a way to run multiple threads (smaller units of a process) concurrently within a single process. It allows for the creation and management of threads, making it possible to execute tasks in parallel, sharing memory space. Threads are particularly useful when tasks are I/O bound, such as file operations or making network requests, where much of the time is spent waiting for external resources.

A typical use case for `threading` includes managing a pool of worker threads that can process multiple tasks concurrently. Here' s a basic example of creating and starting threads using *Thread*:

```
import threading
import time

def crawl(link, delay=3):
    print(f"crawl started for {link}")
    time.sleep(delay)  # Blocking I/O (simulating a network request)
    print(f"crawl ended for {link}")

links = [
    "https://python.org",
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    "https://docs.python.org",
    "https://peps.python.org",
]

# Start threads for each link
threads = []
for link in links:
    # Using `args` to pass positional arguments and `kwargs` for keyword arguments
    t = threading.Thread(target=crawl, args=(link,), kwargs={"delay": 2})
    threads.append(t)

# Start each thread
for t in threads:
    t.start()

# Wait for all threads to finish
for t in threads:
    t.join()
```

버전 3.7에서 변경: 이 모듈은 선택 사양이었지만, 이제는 항상 사용 가능합니다.

더 보기

`concurrent.futures.ThreadPoolExecutor` offers a higher level interface to push tasks to a background thread without blocking execution of the calling thread, while still being able to retrieve their results when needed.

`queue` provides a thread-safe interface for exchanging data between running threads.

`asyncio` offers an alternative approach to achieving task level concurrency without requiring the use of multiple operating system threads.

참고

In the Python 2.x series, this module contained `camelCase` names for some methods and functions. These are deprecated as of Python 3.10, but they are still supported for compatibility with Python 2.5 and lower.

CPython에서는, 전역 인터프리터 록으로 인해 한 번에 하나의 스레드만 파이썬 코드를 실행할 수 있습니다 (설사 일부 성능 지향 라이브러리가 이 제한을 극복할 수 있을지라도). 응용 프로그램에서 멀티 코어 기계의 계산 자원을 더 잘 활용하려면 `multiprocessing`이나 `concurrent.futures.ProcessPoolExecutor`를 사용하는 것이 좋습니다. 그러나, 여러 I/O 병목 작업을 동시에 실행하고 싶을 때 threading은 여전히 적절한 모델입니다.

18.1.2 GIL and performance considerations

Unlike the `multiprocessing` module, which uses separate processes to bypass the *global interpreter lock* (GIL), the threading module operates within a single process, meaning that all threads share the same memory space. However, the GIL limits the performance gains of threading when it comes to CPU-bound tasks, as only one thread can execute Python bytecode at a time. Despite this, threads remain a useful tool for achieving concurrency in many scenarios.

As of Python 3.13, *free-threaded* builds can disable the GIL, enabling true parallel execution of threads, but this feature is not available by default (see **PEP 703**).

18.1.3 Reference

이 모듈은 다음 함수를 정의합니다:

threading.**active_count**()

현재 살아있는 *Thread* 객체 수를 반환합니다. 반환된 수는 *enumerate()* 가 반환한 리스트의 길이와 같습니다.

The function `activeCount` is a deprecated alias for this function.

threading.**current_thread**()

Return the current *Thread* object, corresponding to the caller' s thread of control. If the caller' s thread of control was not created through the `threading` module, a dummy thread object with limited functionality is returned.

The function `currentThread` is a deprecated alias for this function.

threading.**excepthook**(*args*, /)

Thread.run() 에 의해 발생한 포착되지 않은 예외를 처리합니다.

args 인자에는 다음과 같은 어트리뷰트가 있습니다:

- *exc_type*: 예외 형.
- *exc_value*: 예외 값, `None` 일 수 있습니다.
- *exc_traceback*: 예외 트레이스백, `None` 일 수 있습니다.
- *thread*: 예외를 발생시킨 스레드, `None` 일 수 있습니다.

exc_type 이 *SystemExit* 이면, 예외는 조용히 무시됩니다. 그렇지 않으면, *sys.stderr* 에 예외가 인쇄됩니다.

이 함수에서 예외가 발생하면, 이를 처리하기 위해 *sys.excepthook()* 이 호출됩니다.

Thread.run() 에 의해 발생한 포착되지 않은 예외를 처리하는 방법을 제어하기 위해 *threading.excepthook()* 을 재정의할 수 있습니다.

사용자 정의 훅을 사용하여 *exc_value* 를 저장하면 참조 순환을 만들 수 있습니다. 예외가 더는 필요하지 않을 때 참조 순환을 끊기 위해 명시적으로 지워야 합니다.

사용자 정의 훅을 사용하여 *thread* 를 저장하면 파이널라이즈 중인 객체로 설정되면 이를 되살릴 수 있습니다. 객체를 되살리는 것을 방지하려면 사용자 정의 훅이 완료된 후 *thread* 를 보관하지 마십시오.

> **더 보기**
>
> *sys.excepthook()* 은 포착되지 않은 예외를 처리합니다.

Added in version 3.8.

threading.**__excepthook__**

Holds the original value of *threading.excepthook()*. It is saved so that the original value can be restored in case they happen to get replaced with broken or alternative objects.

Added in version 3.10.

threading.**get_ident**()

현재 스레드의 '스레드 식별자' 를 반환합니다. 이것은 0이 아닌 정수입니다. 이 값은 직접적인 의미가 없습니다; 이것은 매직 쿠키로 사용하려는 것입니다, 예를 들어 스레드 특정 데이터의 딕셔너리를 인덱싱하는 데 사용됩니다. 스레드 식별자는 스레드가 종료되고 다른 스레드가 만들어질 때 재활용될 수 있습니다.

Added in version 3.3.

threading.**get_native_id**()

커널이 할당한 현재 스레드의 네이티브 정수 스레드 ID를 반환합니다. 음수가 아닌 정수입니다. 이 값은 시스템 전체에서 이 특정 스레드를 고유하게 식별하는 데 사용될 수 있습니다 (스레드가 종료될 때까지, 그 후에는 OS에서 값을 재활용할 수 있습니다).

가용성: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD, GNU/kFreeBSD.

Added in version 3.8.

버전 3.13에서 변경: Added support for GNU/kFreeBSD.

threading.**enumerate**()

현재 활성인 모든 *Thread* 객체의 리스트를 반환합니다. 이 리스트에는 데몬 스레드와 *current_thread()*에서 만든 더미 스레드 객체가 포함됩니다. 종료된 스레드와 아직 시작되지 않은 스레드는 제외합니다. 그러나, 메인 스레드는 종료되었을 때 조차 항상 결과의 일부입니다.

threading.**main_thread**()

메인 *Thread* 객체를 반환합니다. 정상적인 조건에서, 메인 스레드는 파이썬 인터프리터가 시작된 스레드입니다.

Added in version 3.4.

threading.**settrace**(*func*)

Set a trace function for all threads started from the `threading` module. The *func* will be passed to *sys.settrace()* for each thread, before its *run()* method is called.

threading.**settrace_all_threads**(*func*)

Set a trace function for all threads started from the `threading` module and all Python threads that are currently executing.

*func*는 *run()* 메서드가 호출되기 전에 각 스레드에 대해 *sys.settrace()*로 전달됩니다.

Added in version 3.12.

threading.**gettrace**()

Get the trace function as set by *settrace()*.

Added in version 3.10.

threading.**setprofile**(*func*)

Set a profile function for all threads started from the `threading` module. The *func* will be passed to *sys.setprofile()* for each thread, before its *run()* method is called.

threading.**setprofile_all_threads**(*func*)

Set a profile function for all threads started from the `threading` module and all Python threads that are currently executing.

*func*는 *run()* 메서드가 호출되기 전에 각 스레드에 대해 *sys.setprofile()*로 전달됩니다.

Added in version 3.12.

threading.**getprofile**()

Get the profiler function as set by *setprofile()*.

Added in version 3.10.

threading.**stack_size**([*size*])

새 스레드를 만들 때 사용된 스레드 스택 크기를 반환합니다. 선택적 *size* 인자는 이후에 만들어지는 스레드에 사용할 스택 크기를 지정하며, 0(플랫폼이나 구성된 기본값을 사용합니다)이거나 32,768 (32 KiB) 이상의 양의 정숫값이어야 합니다. *size*를 지정하지 않으면, 0이 사용됩니다. 스레드 스택 크기 변경이 지원되지 않으면, *RuntimeError*가 발생합니다. 지정된 스택 크기가 유효하지 않으면, *ValueError*가 발생하고 스택 크기는 수정되지 않습니다. 32 KiB는 현재 인터프리터 자체에 충분한 스택 공간을 보장하기 위해 지원되는 최소 스택 크기 값입니다. 최소 스택 크기가 32 KiB 보다 커야 한다거나 시스템 메모리 페이지 크기의 배수로 할당해야 하는 등 일부 플랫폼에는 스택 크기 값에 대한 특정 제한이 있을 수 있습니다 - 자세한 내용은 플랫폼 설명서를 참조하십시오 (4 KiB 페이지는

혼합니다; 스택 크기에 4096의 배수를 사용하는 것이 더 구체적인 정보가 없을 때 제안하는 방법입니다).

가용성: Windows, pthreads.

Unix platforms with POSIX threads support.

이 모듈은 또한 다음 상수를 정의합니다:

threading.**TIMEOUT_MAX**

블로킹 함수(`Lock.acquire()`, `RLock.acquire()`, `Condition.wait()` 등)의 *timeout* 매개 변수에 허용되는 최댓값. 이 값보다 큰 timeout을 지정하면 `OverflowError`가 발생합니다.

Added in version 3.2.

이 모듈은 아래 섹션에 자세히 설명되는 많은 클래스를 정의합니다.

이 모듈의 설계는 Java의 스레딩 모델에 약하게 기반합니다. 그러나, Java가 록(locks)과 조건 변수(condition variables)를 모든 객체의 기본 동작으로 만들지만, 파이썬에서는 별도의 객체입니다. 파이썬의 `Thread` 클래스는 Java Thread 클래스 동작의 부분 집합을 지원합니다; 현재, 우선순위가 없고, 스레드 그룹이 없으며 스레드를 파괴, 중지, 일시 중지, 재개 또는 인터럽트 할 수 없습니다. 구현될 때, Java 스레드 클래스의 정적 메서드는 모듈 수준 함수에 매핑됩니다.

아래에 설명된 모든 메서드는 원자 적으로 실행됩니다.

Thread-local data

Thread-local data is data whose values are thread specific. If you have data that you want to be local to a thread, create a `local` object and use its attributes:

```
>>> mydata = local()
>>> mydata.number = 42
>>> mydata.number
42
```

You can also access the `local`-object' s dictionary:

```
>>> mydata.__dict__
{'number': 42}
>>> mydata.__dict__.setdefault('widgets', [])
[]
>>> mydata.widgets
[]
```

If we access the data in a different thread:

```
>>> log = []
>>> def f():
...     items = sorted(mydata.__dict__.items())
...     log.append(items)
...     mydata.number = 11
...     log.append(mydata.number)

>>> import threading
>>> thread = threading.Thread(target=f)
>>> thread.start()
>>> thread.join()
>>> log
[[], 11]
```

we get different data. Furthermore, changes made in the other thread don' t affect data seen in this thread:

```
>>> mydata.number
42
```

Of course, values you get from a *local* object, including their `__dict__` attribute, are for whatever thread was current at the time the attribute was read. For that reason, you generally don' t want to save these values across threads, as they apply only to the thread they came from.

You can create custom *local* objects by subclassing the *local* class:

```
>>> class MyLocal(local):
...     number = 2
...     def __init__(self, /, **kw):
...         self.__dict__.update(kw)
...     def squared(self):
...         return self.number ** 2
```

This can be useful to support default values, methods and initialization. Note that if you define an `__init__()` method, it will be called each time the *local* object is used in a separate thread. This is necessary to initialize each thread' s dictionary.

Now if we create a *local* object:

```
>>> mydata = MyLocal(color='red')
```

we have a default number:

```
>>> mydata.number
2
```

an initial color:

```
>>> mydata.color
'red'
>>> del mydata.color
```

And a method that operates on the data:

```
>>> mydata.squared()
4
```

As before, we can access the data in a separate thread:

```
>>> log = []
>>> thread = threading.Thread(target=f)
>>> thread.start()
>>> thread.join()
>>> log
[[('color', 'red')], 11]
```

without affecting this thread' s data:

```
>>> mydata.number
2
>>> mydata.color
Traceback (most recent call last):
...
AttributeError: 'MyLocal' object has no attribute 'color'
```

Note that subclasses can define *__slots__*, but they are not thread local. They are shared across threads:

```
>>> class MyLocal(local):
...     __slots__ = 'number'

>>> mydata = MyLocal()
>>> mydata.number = 42
>>> mydata.color = 'red'
```

So, the separate thread:

```
>>> thread = threading.Thread(target=f)
>>> thread.start()
>>> thread.join()
```

affects what we see:

```
>>> mydata.number
11
```

class threading.**local**

스레드 로컬 데이터를 나타내는 클래스.

Thread objects

Thread 클래스는 별도의 제어 스레드에서 실행되는 활동을 나타냅니다. 활동을 지정하는 두 가지 방법이 있습니다: 콜러블 객체를 생성자에 전달하거나, 서브 클래스에서 *run()* 메서드를 재정의합니다. 서브 클래스에서는 다른 메서드(생성자를 제외하고)를 재정의해서는 안 됩니다. 즉, 오직 이 클래스의 `__init__()` 와 *run()* 메서드만 재정의합니다.

일단 스레드 객체가 만들어지면, 스레드의 *start()* 메서드를 호출하여 활동을 시작해야 합니다. 이것은 별도의 제어 스레드에서 *run()* 메서드를 호출합니다.

일단 스레드의 활동이 시작되면, 스레드는 '살아있는(alive)' 것으로 간주합니다. *run()* 메서드가 정상적으로 혹은 처리되지 않은 예외를 발생시켜서 종료할 때 살아있음을 멈춥니다. *is_alive()* 메서드는 스레드가 살아있는지 검사합니다.

다른 스레드는 스레드의 *join()* 메서드를 호출할 수 있습니다. 이것은 *join()* 메서드가 호출된 스레드가 종료될 때까지 호출하는 스레드를 블록 합니다.

스레드에는 이름이 있습니다. 이름은 생성자에 전달되고, *name* 어트리뷰트를 통해 읽거나 변경할 수 있습니다.

run() 메서드에서 예외가 발생하면, 이를 처리하기 위해 *threading.excepthook()* 이 호출됩니다. 기본적으로, *threading.excepthook()* 은 *SystemExit* 를 조용히 무시합니다.

스레드는 "데몬 스레드"로 플래그 할 수 있습니다. 이 플래그의 의미는 오직 데몬 스레드만 남았을 때 전체 파이썬 프로그램이 종료된다는 것입니다. 초깃값은 만드는 스레드에서 상속됩니다. 플래그는 *daemon* 프로퍼티나 *daemon* 생성자 인자를 통해 설정할 수 있습니다.

> **참고**
>
> 종료 시 데몬 스레드는 갑자기 중지됩니다. 그들의 자원(가령 열린 파일, 데이터베이스 트랜잭션 등)은 제대로 해제되지 않을 수 있습니다. 스레드가 우아하게 중지되도록 하려면, 스레드를 데몬이 아니도록 만들고 *Event* 와 같은 적절한 신호 메커니즘을 사용하십시오.

"메인 스레드" 객체가 있습니다; 이것은 파이썬 프로그램의 초기 제어 스레드에 해당합니다. 이것은 데몬 스레드가 아닙니다.

"더미 스레드 객체"가 만들어질 수 있습니다. 이들은 "외부 스레드"에 해당하는 스레드 객체로, C 코드에서 직접 만든 것처럼 threading 모듈 외부에서 시작된 제어 스레드입니다. 더미 스레드 객체는 기능이 제한적입니다; 항상 살아 있고 데몬으로 간주하며, 조인할 수 없습니다. 외부 스레드의 종료를 감지하는 것이 불가능하므로 삭제되지 않습니다.

class threading.**Thread**(*group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None, context=None*)

이 생성자는 항상 키워드 인자로 호출해야 합니다. 인자는 다음과 같습니다:

*group*은 None 이어야 합니다; ThreadGroup 클래스가 구현될 때 향후 확장을 위해 예약되어 있습니다.

*target*은 run() 메서드에 의해 호출될 콜러블 객체입니다. 기본값은 None 이며, 아무것도 호출되지 않습니다.

*name*은 스레드 이름입니다. 기본적으로, 고유한 이름이 "Thread-*N*" 형식으로 구성되는데, 여기서 *N*은 작은 십진수입니다. 혹은 *target* 인자가 지정되면, "Thread-*N* (target)" 형식인데, 여기서 "target" 은 target.__name__ 입니다.

*args*는 target 호출을 위한 인자의 리스트나 튜플입니다. 기본값은 () 입니다.

*kwargs*는 target 호출을 위한 키워드 인자의 딕셔너리입니다. 기본값은 {} 입니다.

None 이 아니면, *daemon*은 스레드가 데몬인지를 명시적으로 설정합니다. None(기본값)이면, 데몬 속성은 현재 스레드에서 상속됩니다.

context is the Context value to use when starting the thread. The default value is None which indicates that the sys.flags.thread_inherit_context flag controls the behaviour. If the flag is true, threads will start with a copy of the context of the caller of start(). If false, they will start with an empty context. To explicitly start with an empty context, pass a new instance of Context(). To explicitly start with a copy of the current context, pass the value from copy_context(). The flag defaults true on free-threaded builds and false otherwise.

서브 클래스가 생성자를 재정의하면, 스레드에 다른 작업을 수행하기 전에 베이스 클래스 생성자 (Thread.__init__())를 호출해야 합니다.

버전 3.3에서 변경: *daemon* 매개 변수를 추가했습니다.

버전 3.10에서 변경: Use the *target* name if *name* argument is omitted.

버전 3.14에서 변경: Added the *context* parameter.

start()

스레드 활동을 시작합니다.

스레드 객체 당 최대 한 번 호출되어야 합니다. 객체의 run() 메서드가 별도의 제어 스레드에서 호출되도록 배치합니다.

이 메서드는 같은 스레드 객체에서 두 번 이상 호출되면, RuntimeError를 발생시킵니다.

If supported, set the operating system thread name to threading.Thread.name. The name can be truncated depending on the operating system thread name limits.

버전 3.14에서 변경: Set the operating system thread name.

run()

스레드의 활동을 표현하는 메서드.

서브 클래스에서 이 메서드를 재정의할 수 있습니다. 표준 run() 메서드는 *target* 인자로 객체의 생성자에 전달된 콜러블 객체를 호출합니다, 있다면 *args*와 *kwargs* 인자에서 각각 취한 위치와 키워드 인자로 호출합니다.

Using list or tuple as the *args* argument which passed to the Thread could achieve the same effect.

예제:

```
>>> from threading import Thread
>>> t = Thread(target=print, args=[1])
>>> t.run()
1
>>> t = Thread(target=print, args=(1,))
>>> t.run()
1
```

join(*timeout=None*)

스레드가 종료할 때까지 기다립니다. `join()` 메서드가 호출된 스레드가 정상적으로 혹은 처리되지 않은 예외를 통해 종료하거나 선택적 시간제한 초과가 발생할 때까지 호출하는 스레드를 블록 합니다.

timeout 인자가 있고 `None`이 아니면, 작업의 시간제한을 초(또는 부분 초)로 지정하는 부동 소수점 숫자여야 합니다. `join()`은 항상 `None`을 반환하므로, `join()` 이후에 `is_alive()`를 호출하여 시간제한 초과가 발생했는지 판단해야 합니다 – 스레드가 아직 살아있다면, `join()` 호출이 시간제한을 초과한 것입니다.

timeout 인자가 없거나 `None`이면, 스레드가 종료될 때까지 작업이 블록 됩니다.

스레드는 여러 번 조인될 수 있습니다.

교착 상태를 유발할 수 있어서 현재 스레드를 조인하려고 시도하면 `join()`은 `RuntimeError`를 발생시킵니다. 스레드가 시작되기 전에 `join()` 하는 것도 에러이고 같은 예외가 발생합니다.

If an attempt is made to join a running daemonic thread in late stages of *Python finalization* `join()` raises a `PythonFinalizationError`.

버전 3.14에서 변경: May raise `PythonFinalizationError`.

name

식별 목적으로만 사용되는 문자열. 의미는 없습니다. 여러 스레드에 같은 이름을 지정할 수 있습니다. 초기 이름은 생성자가 설정합니다.

On some platforms, the thread name is set at the operating system level when the thread starts, so that it is visible in task managers. This name may be truncated to fit in a system-specific limit (for example, 15 bytes on Linux or 63 bytes on macOS).

Changes to *name* are only reflected at the OS level when the currently running thread is renamed. (Setting the *name* attribute of a different thread only updates the Python Thread object.)

getName()

setName()

`name`을 위한 폐지된 getter/setter API; 대신 프로퍼티로 직접 사용하십시오.

버전 3.10부터 폐지됨.

ident

이 스레드의 '스레드 식별자' 또는 스레드가 시작되지 않았으면 `None`. 이것은 0이 아닌 정수입니다. `get_ident()` 함수를 참조하십시오. 스레드 식별자는 스레드가 종료되고 다른 스레드가 만들어질 때 재활용될 수 있습니다. 스레드가 종료된 후에도 식별자를 사용할 수 있습니다.

native_id

OS(커널)에서 할당된, 이 스레드의 스레드 ID (`TID`). 음수가 아닌 정수, 또는 스레드가 시작되지 않았으면 `None`입니다. `get_native_id()` 함수를 참조하십시오. 이 값은 시스템 전체에서 이 특정 스레드를 고유하게 식별하는 데 사용될 수 있습니다 (스레드가 종료될 때까지, OS가 값을 재활용 할 수 있습니다).

> **참고**
>
> 프로세스 ID 와 유사하게, 스레드 ID는 스레드가 만들어진 시점부터 스레드가 종료될 때까지만 유효(시스템 전체에서 고유함이 보장)합니다.

가용성: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD.

Added in version 3.8.

is_alive()

스레드가 살아있는지를 반환합니다.

이 메서드는 `run()` 메서드가 시작되기 직전부터 `run()` 메서드가 종료된 직후까지 `True`를 반환합니다. 모듈 함수 `enumerate()`는 모든 살아있는 스레드 리스트를 반환합니다.

daemon

이 스레드가 데몬 스레드인지(True) 아닌지(False)를 나타내는 불리언 값. *start()*가 호출되기 전에 설정되어야 합니다, 그렇지 않으면 *RuntimeError*가 발생합니다. 초깃값은 만드는 스레드에서 상속됩니다; 메인 스레드는 데몬 스레드가 아니므로 메인 스레드에서 만드는 모든 스레드의 기본값은 *daemon* = False입니다.

살아있는 데몬이 아닌 스레드가 남아 있지 않으면 전체 파이썬 프로그램이 종료됩니다.

isDaemon()

setDaemon()

*daemon*을 위한 폐지된 getter/setter API; 대신 프로퍼티로 직접 사용하십시오.

버전 3.10부터 폐지됨.

Lock objects

프리미티브 록(primitive lock)은 잠겨있을 때 특정 스레드가 소유하지 않는 동기화 프리미티브입니다. 파이썬에서는 현재 *_thread* 확장 모듈에 의해 직접 구현되는 가장 낮은 수준의 동기화 프리미티브입니다.

프리미티브 록은 두 상태 중 하나입니다, "잠금(locked)" 이나 "잠금 해제(unlocked)". 잠금 해제 상태로 만들어집니다. 두 가지 기본 메서드가 있습니다, *acquire()*와 *release()*. 상태가 잠금 해제일 때, *acquire()*는 상태를 잠금으로 변경하고 즉시 반환합니다. 상태가 잠금일 때, *acquire()*는 다른 스레드에서의 *release()*에 호출이 잠금 해제로 변경할 때까지 블록 된 후, *acquire()* 호출이 이를 잠금으로 재설정하고 반환합니다. *release()* 메서드는 잠금 상태에서만 호출해야 합니다; 상태를 잠금 해제로 변경하고 즉시 반환합니다. 잠금 해제된 록을 해제하려고 하면, *RuntimeError*가 발생합니다.

록은 컨텍스트 관리자 프로토콜도 지원합니다.

둘 이상의 스레드가 *acquire()*에서 블록 되어 상태가 잠금 해제가 되기를 기다릴 때, *release()* 호출이 상태를 잠금 해제로 재설정하면 하나의 스레드만 진행됩니다; 대기 중인 스레드 중 어느 것이 진행하는지는 정의되지 않았으며, 구현에 따라 다를 수 있습니다.

모든 메서드는 원자 적으로 실행됩니다.

class threading.**Lock**

프리미티브 록 객체를 구현하는 클래스. 일단 스레드가 록을 획득하면, 이후에 해당 록을 확보하려고 시도하면 해제될 때까지 블록 합니다; 모든 스레드가 해제할 수 있습니다.

버전 3.13에서 변경: Lock is now a class. In earlier Pythons, Lock was a factory function which returned an instance of the underlying private lock type.

acquire(*blocking=True*, *timeout=-1*)

블로킹이거나 비 블로킹으로, 록을 획득합니다.

blocking 인자를 True(기본값)로 설정하여 호출하면, 록이 잠금 해제될 때까지 블록 한 다음, 잠금으로 설정하고 True를 반환합니다.

blocking 인자를 False로 설정하여 호출하면, 블록 하지 않습니다. *blocking*이 True로 설정된 호출이 블록 될 것이라면, 즉시 False를 반환합니다; 그렇지 않으면, 록을 잠금으로 설정하고 True를 반환합니다.

양수 값으로 설정된 부동 소수점 *timeout* 인자로 호출하면, 록을 획득할 수 없는 한 최대 *timeout*에 지정된 초 동안 블록 합니다. -1의 *timeout* 인자는 제한 없는 대기를 지정합니다. *blocking*이 False일 때 *timeout*을 지정하는 것은 금지되어 있습니다.

록이 성공적으로 획득되면 반환 값은 True이고, 그렇지 않으면 (예를 들어 *timeout*이 만료되면) False입니다.

버전 3.2에서 변경: *timeout* 매개 변수가 추가되었습니다.

버전 3.2에서 변경: 하부 스레딩 구현이 지원한다면 POSIX에서 시그널로 록 획득을 중단할 수 있습니다.

버전 3.14에서 변경: Lock acquisition can now be interrupted by signals on Windows.

release()

록을 해제합니다. 록을 획득한 스레드뿐만 아니라 모든 스레드에서 호출할 수 있습니다.

록이 잠금일 때, 잠금 해제로 재설정하고 반환합니다. 록이 잠금 해제될 때까지 다른 스레드가 블록 되어 기다리고 있으면, 그들 중 정확히 하나의 스레드가 진행되도록 합니다.

잠금 해제된 록에서 호출되면, `RuntimeError`가 발생합니다.

반환 값이 없습니다.

locked()

록이 획득되면 `True`를 반환합니다.

RLock objects

재진입 록(reentrant lock)은 같은 스레드에 의해 여러 번 획득될 수 있는 동기화 프리미티브입니다. 내부적으로, 프리미티브 록에서 사용하는 잠금/잠금 해제 상태에 더해 "소유하는 스레드(owning thread)"와 "재귀 수준(recursion level)" 개념을 사용합니다. 잠금 상태에서는, 어떤 스레드가 록을 소유합니다; 잠금 해제 상태에서는 아무런 스레드도 록을 소유하지 않습니다.

Threads call a lock' s `acquire()` method to lock it, and its `release()` method to unlock it.

> **참고**
>
> Reentrant locks support the *context management protocol*, so it is recommended to use `with` instead of manually calling `acquire()` and `release()` to handle acquiring and releasing the lock for a block of code.

RLock의 `acquire()`/`release()` 호출 쌍은 Lock의 `acquire()`/`release()`와 달리 중첩될 수 있습니다. 오직 마지막 `release()`(가장 바깥쪽 쌍의 `release()`)만 록을 잠금 해제 상태로 재설정하고 `acquire()`에서 블록 된 다른 스레드가 진행하도록 할 수 있습니다.

`acquire()`/`release()` must be used in pairs: each acquire must have a release in the thread that has acquired the lock. Failing to call release as many times the lock has been acquired can lead to deadlock.

class threading.**RLock**

이 클래스는 재진입 록 객체를 구현합니다. 재진입 록은 획득한 스레드에서 해제해야 합니다. 일단 스레드가 재진입 록을 획득하면, 같은 스레드는 블록 하지 않고 다시 스레드를 획득할 수 있습니다; 스레드는 획득할 때마다 한 번씩 해제해야 합니다.

`RLock`은 실제로는 플랫폼에서 지원하는 가장 효율적인 버전의 구상 RLock 클래스 인스턴스를 반환하는 팩토리 함수임에 유의하십시오.

acquire(*blocking=True, timeout=-1*)

블로킹이거나 비 블로킹으로, 록을 획득합니다.

> **더 보기**
>
> ***Using RLock as a context manager***
> Recommended over manual `acquire()` and `release()` calls whenever practical.

blocking 인자를 `True`(기본값)로 설정하여 호출할 때:

- If no thread owns the lock, acquire the lock and return immediately.
- If another thread owns the lock, block until we are able to acquire lock, or *timeout*, if set to a positive float value.
- If the same thread owns the lock, acquire the lock again, and return immediately. This is the difference between `Lock` and `RLock`; `Lock` handles this case the same as the previous, blocking until the lock can be acquired.

blocking 인자를 `False`로 설정하여 호출할 때:

- If no thread owns the lock, acquire the lock and return immediately.
- If another thread owns the lock, return immediately.
- If the same thread owns the lock, acquire the lock again and return immediately.

In all cases, if the thread was able to acquire the lock, return `True`. If the thread was unable to acquire the lock (i.e. if not blocking or the timeout was reached) return `False`.

If called multiple times, failing to call `release()` as many times may lead to deadlock. Consider using `RLock` as a context manager rather than calling acquire/release directly.

버전 3.2에서 변경: *timeout* 매개 변수가 추가되었습니다.

release()

재귀 수준을 낮추면서, 잠금을 해제합니다. 감소 후에 0이 되면, 록을 잠금 해제로 (아무런 스레드도 소유하지 않은) 재설정하고, 록이 잠금 해제되도록 기다리면서 블록 된 다른 스레드가 있으면, 그중 정확히 하나가 진행되도록 합니다. 감소 후에 재귀 수준이 여전히 0이 아니면, 록은 잠금이고, 호출하는 스레드에 의해 소유된 채로 유지됩니다.

호출하는 스레드가 록을 소유했을 때만 이 메서드를 호출하십시오. 록을 획득하지 않았을 때 이 메서드가 호출되면 `RuntimeError`가 발생합니다.

반환 값이 없습니다.

locked()

Return a boolean indicating whether this object is locked right now.

Added in version 3.14.

Condition objects

조건 변수(condition variable)는 항상 어떤 종류의 록과 연관됩니다; 이것은 전달되거나 기본적으로 만들어집니다. 전달하는 것은 여러 조건 변수가 같은 록을 공유해야 할 때 유용합니다. 록은 조건 객체의 일부입니다: 별도로 추적할 필요가 없습니다.

조건 변수는 컨텍스트 관리자 프로토콜을 준수합니다: `with` 문을 사용해서 감싼 블록의 지속 시간 동안 연관된 록을 획득합니다. `acquire()`와 `release()` 메서드도 연관된 록의 해당 메서드를 호출합니다.

다른 메서드들은 연관된 록을 잡은 상태에서 호출해야 합니다. `wait()` 메서드는 록을 해제한 다음, 다른 스레드가 `notify()`나 `notify_all()`을 호출하여 록을 해제할 때까지 블록 합니다. 일단 깨어나면, `wait()`는 록을 다시 획득하고 반환합니다. 시간제한을 지정할 수도 있습니다.

`notify()` 메서드는 있다면 조건 변수를 기다리는 스레드 중 하나를 깨웁니다. `notify_all()` 메서드는 조건 변수를 기다리는 모든 스레드를 깨웁니다.

참고: `notify()`와 `notify_all()` 메서드는 록을 해제하지 않습니다; 이것은 깨어난 스레드나 스레드들이 `wait()` 호출에서 즉시 반환되지 않지만, `notify()`나 `notify_all()`을 호출한 스레드가 최종적으로 록 소유권을 포기할 때만 반환됨을 의미합니다.

조건 변수를 사용하는 일반적인 프로그래밍 스타일은 록을 사용하여 어떤 공유 상태에 대한 액세스를 동기화합니다; 특정 상태 변경에 관심 있는 스레드는 원하는 상태를 볼 때까지 `wait()`를 반복적으로 호출하는 반면, 상태를 변경하는 스레드는 대기자 중 하나가 원하는 상태일 수 있도록 상태를 변경할 때 `notify()`나 `notify_all()`을 호출합니다. 예를 들어, 다음 코드는 무제한 버퍼 용량의 일반적인 생산자-소비자 상황입니다:

```
# 한 항목을 소비합니다
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# 한 항목을 생산합니다
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
with cv:
    make_an_item_available()
    cv.notify()
```

*wait()*가 임의의 긴 시간 후에 반환될 수 있고, *notify()* 호출을 유발한 조건이 더는 참이 아닐 수 있기 때문에, 응용 프로그램의 조건에 대한 `while` 루프 검사가 필요합니다. 이것은 다중 스레드 프로그래밍에 본질적으로 수반됩니다. *wait_for()* 메서드를 사용하면 조건 검사를 자동화하고 시간제한 계산을 쉽게 수행할 수 있습니다:

```
# 한 항목을 소비합니다
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

*notify()*와 *notify_all()* 중에서 선택하려면, 하나의 상태 변경에 흥미 있는 대기 중인 스레드가 하나일지 여러 개일지를 고려하십시오. 예를 들어 일반적인 생산자-소비자 상황에서, 하나의 항목을 버퍼에 추가하면 오직 하나의 소비자 스레드만 깨울 필요가 있습니다.

class threading.**Condition**(*lock=None*)

이 클래스는 조건 변수 객체를 구현합니다. 조건 변수를 사용하면 하나 이상의 스레드가 다른 스레드에 의해 통지될 때까지 기다릴 수 있습니다.

lock 인자가 제공되고 `None`이 아니면, *Lock*이나 *RLock* 객체여야 하며, 하부 록으로 사용됩니다. 그렇지 않으면, 새 *RLock* 객체가 만들어지고 하부 록으로 사용됩니다.

버전 3.3에서 변경: 팩토리 함수에서 클래스로 변경되었습니다.

acquire(**args*)

하부 록을 획득합니다. 이 메서드는 하부 록에서 해당 메서드를 호출합니다; 반환 값은 그 메서드가 반환하는 것입니다.

release()

하부 록을 해제합니다. 이 메서드는 하부 록에서 해당 메서드를 호출합니다; 반환 값이 없습니다.

locked()

Return a boolean indicating whether this object is locked right now.

Added in version 3.14.

wait(*timeout=None*)

통지되거나 시간제한이 만료될 때까지 기다립니다. 이 메서드가 호출될 때 호출하는 스레드가 록을 획득하지 않았으면, *RuntimeError*가 발생합니다.

이 메서드는 하부 록을 해제한 다음, 같은 조건 변수에 대한 다른 스레드에서의 *notify()*나 *notify_all()* 호출에 의해 깨어날 때까지 또는 선택적 시간제한 만료가 발생할 때까지 블록합니다. 일단 깨어나거나 시간제한이 만료되면, 록을 다시 획득하고 반환합니다.

timeout 인자가 있고 `None`이 아니면, 작업의 시간제한을 초(또는 부분 초)로 지정하는 부동 소수점 숫자여야 합니다.

하부 록이 *RLock*일 때, 록이 여러 번 재귀적으로 획득되었을 때 록을 실제로 잠금 해제하지 못할 수 있기 때문에, *release()* 메서드를 사용하여 록을 해제하지 않습니다. 대신, *RLock* 클래스의 내부 인터페이스가 사용되어, 재귀적으로 여러 번 획득한 경우에도 실제로 록을 잠금 해제합니다. 그런 다음 다른 내부 인터페이스를 사용하여 록을 다시 획득할 때 재귀 수준을 복원합니다.

주어진 *timeout*이 만료되지 않는 한 반환 값은 `True`이며, 만료되면 `False`입니다.

버전 3.2에서 변경: 이전에는, 메서드가 항상 `None`을 반환했습니다.

wait_for(*predicate, timeout=None*)

조건이 참으로 평가될 때까지 기다립니다. *predicate*는 불리언 값으로 해석될 결과를 반환하는 콜러블 이어야 합니다. 최대 대기 시간을 주는 *timeout*이 제공될 수 있습니다.

이 유틸리티 메서드는 술어(predicate)가 충족될 때까지, 또는 시간제한 만료가 발생할 때까지 *wait()*를 반복적으로 호출할 수 있습니다. 반환 값은 predicate의 마지막 반환 값이며 메서드가 시간제한 만료되면 `False`로 평가됩니다.

시간제한 기능을 무시할 때, 이 메서드를 호출하는 것은 대략 다음과 같이 작성하는 것과 동등합니다:

```
while not predicate():
    cv.wait()
```

따라서, *wait()*와 같은 규칙이 적용됩니다: 호출될 때 록을 잡고 있어야 하며 반환할 때 다시 확보됩니다. predicate는 록을 잡고 있는 상태로 평가됩니다.

Added in version 3.2.

notify(*n=1*)

기본적으로, (있다면) 이 조건에서 대기 중인 하나의 스레드를 깨웁니다. 이 메서드가 호출될 때 호출하는 스레드가 잠금을 획득하지 않았으면 *RuntimeError*가 발생합니다.

이 메서드는 조건 변수를 기다리는 스레드를 최대 *n* 개 깨웁니다; 기다리는 스레드가 없으면 아무런 일도 하지 않습니다.

적어도 *n* 스레드가 대기 중이면, 현재 구현은 정확히 *n* 스레드를 깨웁니다. 그러나, 이 동작에 의존하는 것은 안전하지 않습니다. 미래에는, 최적화된 구현이 때때로 *n* 스레드보다 많이 깨울 수 있습니다.

참고: 깨어난 스레드는 록을 다시 획득할 때까지 *wait()* 호출에서 실제로 반환하지 않습니다. *notify()*가 록을 해제하지 않기 때문에, 호출자가 해제해야 합니다.

notify_all()

이 조건에서 대기 중인 모든 스레드를 깨웁니다. 이 메서드는 *notify()*처럼 작동하지만, 하나 대신에 대기 중인 모든 스레드를 깨웁니다. 이 메서드가 호출될 때 호출하는 스레드가 잠금을 획득하지 않았으면 *RuntimeError*가 발생합니다.

The method `notifyAll` is a deprecated alias for this method.

Semaphore objects

이것은 일찌감치 네덜란드 컴퓨터 과학자 Edsger W. Dijkstra가 발명한, 컴퓨터 과학 역사상 가장 오래된 동기화 프리미티브 중 하나입니다 (그는 *acquire()*와 *release()* 대신 `P()`와 `V()`라는 이름을 사용했습니다).

세마포어는 각 *acquire()* 호출에 의해 감소하고 각 *release()* 호출에 의해 증가하는 내부 카운터를 관리합니다. 카운터는 절대 0 밑으로 떨어질 수 없습니다; *acquire()*가 0임을 발견하면, 다른 스레드가 *release()*를 호출할 때까지 대기하면서 블록 합니다.

세마포어도 컨텍스트 관리자 프로토콜을 지원합니다.

class threading.**Semaphore**(*value=1*)

이 클래스는 세마포어 객체를 구현합니다. 세마포어는 *release()* 호출 수에서 *acquire()* 호출 수를 빼고, 초깃값을 더한 원자 적 카운터를 관리합니다. *acquire()* 메서드는 카운터를 음수로 만들지 않고 반환할 수 있을 때까지 필요하면 블록 합니다. 지정하지 않으면, *value*의 기본값은 1입니다.

선택적 인자는 내부 카운터의 초깃 값(*value*)을 제공합니다; 기본값은 `1`입니다. 주어진 *value*가 0보다 작으면 *ValueError*가 발생합니다.

버전 3.3에서 변경: 팩토리 함수에서 클래스로 변경되었습니다.

acquire(*blocking=True, timeout=None*)

세마포어를 획득합니다.

인자 없이 호출될 때:

- 진입 시 내부 카운터가 0보다 크면, 1 감소시키고 즉시 `True`를 반환합니다.

- 진입 시 내부 카운터가 0이면, `release()`를 호출하여 깨울 때까지 블록 합니다. 일단 깨어나면 (그리고 카운터가 0보다 크면), 카운터를 1줄이고 `True`를 반환합니다. `release()`를 호출할 때마다 정확히 하나의 스레드가 깨어납니다. 스레드가 깨어나는 순서에 의존해서는 안 됩니다.

`False`로 설정한 *blocking*으로 호출하면 블록 하지 않습니다. 인자가 없는 호출이 블록 할 것이라면, 즉시 `False`를 반환합니다; 그렇지 않으면, 인자 없이 호출할 때와 같은 작업을 수행하고, `True`를 반환합니다.

`None` 이외의 *timeout*으로 호출하면, 최대 *timeout* 초 동안 블록 합니다. 그 기간 획득이 완료되지 않으면, `False`를 반환합니다. 그렇지 않으면 `True`를 반환합니다.

버전 3.2에서 변경: *timeout* 매개 변수가 추가되었습니다.

release(*n=1*)

내부 카운터를 *n* 증가시키면서 세마포어를 해제합니다. 진입 시 0이고 다른 스레드가 다시 0보다 커지기를 기다리고 있으면, 해당 스레드를 *n*개 깨웁니다.

버전 3.9에서 변경: 여러 대기 스레드를 한 번에 해제하기 위해 *n* 매개 변수를 추가했습니다.

class threading.**BoundedSemaphore**(*value=1*)

경계 세마포어 객체를 구현하는 클래스. 경계 세마포어는 현재 값이 초깃값을 초과하지 않는지 확인합니다. 그렇다면, `ValueError`가 발생합니다. 대부분은 세마포어는 제한된 용량의 자원을 보호하는 데 사용됩니다. 세마포어가 너무 여러 번 해제되면 버그의 징조입니다. 지정하지 않으면, *value*의 기본값은 1입니다.

버전 3.3에서 변경: 팩토리 함수에서 클래스로 변경되었습니다.

`Semaphore` example

세마포어는 예를 들어 데이터베이스 서버와 같이 제한된 용량의 자원을 보호하는 데 종종 사용됩니다. 자원의 크기가 고정된 상황에서는, 경계 세마포어를 사용해야 합니다. 작업자 스레드를 만들기 전에, 메인 스레드가 세마포어를 초기화합니다:

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

일단 만들어지면, 작업자 스레드는 서버에 연결해야 할 때 세마포어의 acquire 및 release 메서드를 호출합니다:

```
with pool_sema:
    conn = connectdb()
    try:
        # ... 연결을 사용합니다 ...
    finally:
        conn.close()
```

경계 세마포어를 사용하면 세마포어가 획득한 것보다 더 많이 해제되는 프로그래밍 에러가 감지되지 않을 가능성이 줄어듭니다.

Event objects

이것은 스레드 간 통신을 위한 가장 간단한 메커니즘 중 하나입니다: 하나의 스레드는 이벤트를 알리고 다른 스레드는 이를 기다립니다.

이벤트 객체는 `set()` 메서드를 사용하여 참으로 설정하고 `clear()` 메서드를 사용하여 거짓으로 재설정할 수 있는 내부 플래그를 관리합니다. `wait()` 메서드는 플래그가 참이 될 때까지 블록 합니다.

class threading.**Event**

이벤트 객체를 구현하는 클래스. 이벤트는 `set()` 메서드로 참으로 설정하고 `clear()` 메서드로 거짓으로 재설정 할 수 있는 플래그를 관리합니다. `wait()` 메서드는 플래그가 참이 될 때까지 블록 합니다. 플래그는 처음에는 거짓입니다.

버전 3.3에서 변경: 팩토리 함수에서 클래스로 변경되었습니다.

is_set()

내부 플래그가 참이면 그리고 오직 그때만 `True`를 반환합니다.

The method `isSet` is a deprecated alias for this method.

set()

내부 플래그를 참으로 설정합니다. 이것이 참이 되기를 기다리는 모든 스레드가 깨어납니다. 일단 플래그가 참이면 *wait()*를 호출하는 스레드는 전혀 블록 하지 않습니다.

clear()

내부 플래그를 거짓으로 재설정합니다. 이후 *wait()*를 호출하는 스레드는 내부 플래그를 다시 참으로 설정하기 위해 *set()*을 호출할 때까지 블록 합니다.

wait(*timeout=None*)

Block as long as the internal flag is false and the timeout, if given, has not expired. The return value represents the reason that this blocking method returned; `True` if returning because the internal flag is set to true, or `False` if a timeout is given and the internal flag did not become true within the given wait time.

timeout 인자가 있고 `None`이 아니면, 작업의 시간제한을 초 또는 부분 초로 지정하는 부동 소수점 숫자여야 합니다.

버전 3.1에서 변경: 이전에는, 메서드가 항상 `None`을 반환했습니다.

Timer objects

이 클래스는 특정 시간이 지난 후에만 실행되어야 하는 조치를 나타냅니다 – 타이머. *Timer*는 *Thread*의 서브 클래스이며 사용자 정의 스레드를 만드는 예제로도 기능합니다.

타이머는 스레드와 마찬가지로, *Timer.start* 메서드를 호출하여 시작됩니다. *cancel()* 메서드를 호출하여 (조치를 시작하기 전에) 타이머를 중지할 수 있습니다. 조치를 실행하기 전에 타이머가 대기하는 간격은 사용자가 지정한 간격과 정확히 같지 않을 수 있습니다.

예를 들면:

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # 30초 후에, "hello, world" 가 인쇄됩니다
```

class threading.**Timer**(*interval, function, args=None, kwargs=None*)

interval 초가 지난 후 *args* 인자와 *kwargs* 키워드 인자로 *function*을 실행하는 타이머를 만듭니다. *args*가 `None`(기본값)이면 빈 리스트가 사용됩니다. *kwargs*가 `None`(기본값)이면 빈 딕셔너리가 사용됩니다.

버전 3.3에서 변경: 팩토리 함수에서 클래스로 변경되었습니다.

cancel()

타이머를 중지하고, 타이머 조치의 실행을 취소합니다. 타이머가 아직 대기 상태에 있을 때만 작동합니다.

Barrier objects

Added in version 3.2.

이 클래스는 서로를 기다려야 하는 고정된 수의 스레드에서 사용할 수 있는 간단한 동기화 프리미티브를 제공합니다. 각 스레드는 *wait()* 메서드를 호출하여 장벽(barrier)을 통과하려고 시도하고 모든 스레드가 *wait()* 호출을 수행할 때까지 블록 합니다. 이 시점에서, 스레드가 동시에 해제됩니다.

장벽은 같은 수의 스레드에 대해 여러 번 재사용 할 수 있습니다.

예를 들어, 다음은 클라이언트와 서버 스레드를 동기화하는 간단한 방법입니다:

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class threading.**Barrier**(*parties*, *action=None*, *timeout=None*)

parties 수의 스레드에 대한 장벽 객체를 만듭니다. 제공되면, *action*은 해제될 때 스레드 중 하나가 호출할 콜러블입니다. *timeout*은 `wait()` 메서드에 대해 지정되지 않을 때 사용될 기본 시간제한 값입니다.

wait(*timeout=None*)

장벽을 통과합니다. 장벽에 속한 모든 스레드가 이 함수를 호출할 때, 모두 동시에 해제됩니다. *timeout*이 제공되면, 클래스 생성자에 제공된 것보다 우선하여 사용됩니다.

반환 값은 0에서 *parties* - 1 범위의 정수이며, 스레드마다 다릅니다. 이것은 특별한 하우스키핑을 수행할 스레드를 선택하는데 사용될 수 있습니다, 예를 들어:

```
i = barrier.wait()
if i == 0:
    # 오직 한 스레드만 이것을 인쇄하면 됩니다
    print("passed the barrier")
```

생성자에 *action*이 제공되면, 스레드 중 하나가 해제되기 전에 호출합니다. 이 호출로 에러가 발생하면, 장벽은 망가진 상태가 됩니다.

호출 시간제한이 만료되면, 장벽은 망가진 상태가 됩니다.

스레드가 기다리는 동안 장벽이 망가지거나 재설정되면 이 메서드는 `BrokenBarrierError` 예외를 발생시킬 수 있습니다.

reset()

장벽을 기본의 빈 상태로 되돌립니다. 대기 중인 모든 스레드는 `BrokenBarrierError` 예외를 수신합니다.

상태를 알 수 없는 다른 스레드가 있을 때 이 함수를 사용하려면 외부 동기화가 필요할 수 있습니다. 장벽이 망가지면 그냥 두고 새 장벽을 만드는 것이 좋습니다.

abort()

장벽을 망가진 상태로 보냅니다. 이로 인해 `wait()`에 대한 활성 또는 미래의 호출이 `BrokenBarrierError`로 실패합니다. 예를 들어 응용 프로그램의 교착 상태를 피하고자 스레드 중 하나를 중단해야 할 때 이를 사용하십시오.

스레드 중 하나가 잘못되는 것을 막기 위해 단순히 적절한 *timeout* 값으로 장벽을 만드는 것이 바람직 할 수 있습니다.

parties

장벽을 통과하는 데 필요한 스레드 수.

n_waiting

현재 장벽에서 대기 중인 스레드 수.

broken

장벽이 망가진 상태이면 `True`인 불리언.

exception threading.**BrokenBarrierError**

*RuntimeError*의 서브 클래스인, 이 예외는 *Barrier* 객체가 재설정되거나 망가질 때 발생합니다.

18.1.4 with 문으로 록, 조건 및 세마포어 사용하기

이 모듈에서 제공하는 `acquire`와 `release` 메서드가 있는 모든 객체는 `with` 문의 컨텍스트 관리자로 사용될 수 있습니다. 블록에 진입할 때 `acquire` 메서드가 호출되고, 블록을 벗어날 때 `release`가 호출됩니다. 따라서, 다음 코드 조각은:

```
with some_lock:
    # 뭔가 합니다...
```

다음과 동등합니다:

```
some_lock.acquire()
try:
    # 뭔가 합니다...
finally:
    some_lock.release()
```

현재 *Lock*, *RLock*, *Condition*, *Semaphore* 및 *BoundedSemaphore* 객체는 `with` 문 컨텍스트 관리자로 사용될 수 있습니다.

18.2 multiprocessing — Process-based parallelism

소스 코드: Lib/multiprocessing/

가용성: not Android, not iOS, not WASI.

이 모듈은 모바일 플랫폼이나 웹어셈블리 플랫폼에서 지원되지 않습니다.

18.2.1 소개

multiprocessing is a package that supports spawning processes using an API similar to the *threading* module. The *multiprocessing* package offers both local and remote concurrency, effectively side-stepping the *Global Interpreter Lock* by using subprocesses instead of threads. Due to this, the *multiprocessing* module allows the programmer to fully leverage multiple processors on a given machine. It runs on both POSIX and Windows.

multiprocessing 모듈은 *threading* 모듈에 대응 물이 없는 API도 제공합니다. 이것의 대표적인 예가 *Pool* 객체입니다. 이 객체는 여러 입력 값에 걸쳐 함수의 실행을 병렬 처리하고 입력 데이터를 프로세스에 분산시키는 편리한 방법을 제공합니다(데이터 병렬 처리). 다음 예제는 자식 프로세스가 해당 모듈을 성공적으로 임포트 할 수 있도록, 모듈에서 이러한 함수를 정의하는 일반적인 방법을 보여줍니다. 다음은 *Pool* 를 사용하는 데이터 병렬 처리의 기본 예제입니다:

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

표준 출력으로 다음과 같은 것을 인쇄합니다

```
[1, 4, 9]
```

더 보기

concurrent.futures.ProcessPoolExecutor offers a higher level interface to push tasks to a background process without blocking execution of the calling process. Compared to using the *Pool* interface directly, the *concurrent.futures* API more readily allows the submission of work to the underlying process pool to be separated from waiting for the results.

Process 클래스

*multiprocessing*에서, 프로세스는 *Process* 객체를 생성한 후 *start()* 메서드를 호출해서 스폰합니다. *Process* 는 *threading.Thread* 의 API를 따릅니다. 다중 프로세스 프로그램의 간단한 예는 다음과 같습니다

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

이 과정에 참여하는 개별 프로세스의 ID를 보기 위해, 이렇게 예제를 확장합니다:

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

`if __name__ == '__main__'` 부분이 필요한 이유에 대한 설명은 프로그래밍 지침을 보십시오.

The arguments to *Process* usually need to be unpickleable from within the child process. If you tried typing the above example directly into a REPL it could lead to an *AttributeError* in the child process trying to locate the *f* function in the `__main__` module.

컨텍스트 및 시작 방법

플랫폼에 따라, `multiprocessing`은 프로세스를 시작하는 세 가지 방법을 지원합니다. 이러한 시작 방법은

spawn

The parent process starts a fresh Python interpreter process. The child process will only inherit those resources necessary to run the process object' s `run()` method. In particular, unnecessary file descriptors and handles from the parent process will not be inherited. Starting a process using this method is rather slow compared to using *fork* or *forkserver*.

Available on POSIX and Windows platforms. The default on Windows and macOS.

fork

부모 프로세스는 `os.fork()` 를 사용하여 파이썬 인터프리터를 포크 합니다. 자식 프로세스는, 시작될 때, 부모 프로세스와 실질적으로 같습니다. 부모의 모든 자원이 자식 프로세스에 의해 상속됩니다. 다중 스레드 프로세스를 안전하게 포크 하기 어렵다는 점에 주의하십시오.

Available on POSIX systems.

버전 3.14에서 변경: This is no longer the default start method on any platform. Code that requires *fork* must explicitly specify that via `get_context()` or `set_start_method()`.

버전 3.12에서 변경: If Python is able to detect that your process has multiple threads, the `os.fork()` function that this start method calls internally will raise a `DeprecationWarning`. Use a different start method. See the `os.fork()` documentation for further explanation.

forkserver

When the program starts and selects the *forkserver* start method, a server process is spawned. From then on, whenever a new process is needed, the parent process connects to the server and requests that it fork a new process. The fork server process is single threaded unless system libraries or preloaded imports spawn threads as a side-effect so it is generally safe for it to use `os.fork()`. No unnecessary resources are inherited.

Available on POSIX platforms which support passing file descriptors over Unix pipes such as Linux. The default on those.

버전 3.14에서 변경: This became the default start method on POSIX platforms.

버전 3.4에서 변경: *spawn* added on all POSIX platforms, and *forkserver* added for some POSIX platforms. Child processes no longer inherit all of the parents inheritable handles on Windows.

버전 3.8에서 변경: On macOS, the *spawn* start method is now the default. The *fork* start method should be considered unsafe as it can lead to crashes of the subprocess as macOS system libraries may start threads. See bpo-33725.

버전 3.14에서 변경: On POSIX platforms the default start method was changed from *fork* to *forkserver* to retain the performance but avoid common multithreaded process incompatibilities. See gh-84559.

On POSIX using the *spawn* or *forkserver* start methods will also start a *resource tracker* process which tracks the unlinked named system resources (such as named semaphores or `SharedMemory` objects) created by processes of the program. When all processes have exited the resource tracker unlinks any remaining tracked object. Usually there should be none, but if a process was killed by a signal there may be some "leaked" resources. (Neither leaked semaphores nor shared memory segments will be automatically unlinked until the next reboot. This is problematic for both objects because the system allows only a limited number of named semaphores, and shared memory segments occupy some space in the main memory.)

시작 방법을 선택하려면 메인 모듈의 `if __name__ == '__main__'` 절에서 `set_start_method()`를 사용하십시오. 예를 들면:

```
import multiprocessing as mp

def foo(q):
    q.put('hello')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

set_start_method() 는 프로그램에서 한 번만 사용되어야 합니다.

또는, *get_context()*를 사용하여 컨텍스트 객체를 얻을 수 있습니다. 컨텍스트 객체는 multiprocessing 모듈과 같은 API를 제공하므로 한 프로그램에서 여러 시작 방법을 사용할 수 있습니다.

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

한 컨텍스트와 관련된 객체는 다른 컨텍스트의 프로세스와 호환되지 않을 수 있음에 주의하십시오. 특히 *fork* 컨텍스트를 사용하여 생성된 록은 *spawn* 또는 *forkserver* 시작 방법을 사용하여 시작된 프로세스로 전달될 수 없습니다.

Libraries using *multiprocessing* or *ProcessPoolExecutor* should be designed to allow their users to provide their own multiprocessing context. Using a specific context of your own within a library can lead to incompatibilities with the rest of the library user' s application. Always document if your library requires a specific start method.

경고

The `'spawn'` and `'forkserver'` start methods generally cannot be used with "frozen" executables (i.e., binaries produced by packages like **PyInstaller** and **cx_Freeze**) on POSIX systems. The `'fork'` start method may work if code does not use threads.

프로세스 간 객체 교환

multiprocessing 은 두 가지 유형의 프로세스 간 통신 채널을 지원합니다:

큐

Queue 클래스는 *queue.Queue* 의 클론에 가깝습니다. 예를 들면:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    print(q.get())     # prints "[42, None, 'hello']"
    p.join()
```

Queues are thread and process safe. Any object put into a *multiprocessing* queue will be serialized.

파이프

Pipe() 함수는 파이프로 연결된 한 쌍의 연결 객체를 돌려주는데 기본적으로 양방향(duplex) 입니다. 예를 들면:

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None, 'hello']"
    p.join()
```

Pipe() 가 반환하는 두 개의 연결 객체는 파이프의 두 끝을 나타냅니다. 각 연결 객체에는 (다른 것도 있지만) `send()` 및 `recv()` 메서드가 있습니다. 두 프로세스 (또는 스레드) 가 파이프의 같은 끝에서 동시에 읽거나 쓰려고 하면 파이프의 데이터가 손상될 수 있습니다. 물론 파이프의 다른 끝을 동시에 사용하는 프로세스로 인해 손상될 위험은 없습니다.

The `send()` method serializes the object and `recv()` re-creates the object.

프로세스 간 동기화

multiprocessing 은 *threading* 에 있는 모든 동기화 프리미티브의 등가물을 포함합니다. 예를 들어 한 번에 하나의 프로세스만 표준 출력으로 인쇄하도록 록을 사용할 수 있습니다:

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

록을 사용하지 않으면 다른 프로세스의 출력들이 모두 섞일 수 있습니다.

프로세스 간 상태 공유

위에서 언급했듯이, 동시성 프로그래밍을 할 때 보통 가능한 한 공유된 상태를 사용하지 않는 것이 최선입니다. 여러 프로세스를 사용할 때 특히 그렇습니다.

그러나, 정말로 공유 데이터를 사용해야 한다면 *multiprocessing* 이 몇 가지 방법을 제공합니다.

공유 메모리

데이터는 *Value* 또는 *Array*를 사용하여 공유 메모리 맵에 저장 될 수 있습니다. 예를 들어, 다음 코드는

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

를 인쇄할 것입니다

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

`num` 과 `arr` 을 만들 때 사용되는 `'d'` 와 `'i'` 인자는 *array* 모듈에서 사용되는 종류의 타입 코드입니다: `'d'` 는 배정밀도 부동 소수점을 나타내고, `'i'` 는 부호 있는 정수를 나타냅니다. 이러한 공유 객체는 프로세스 및 스레드에 안전합니다.

공유 메모리를 더 유연하게 사용하려면, 공유 메모리에 할당된 임의의 ctypes 객체 생성을 지원하는 *multiprocessing.sharedctypes* 모듈을 사용할 수 있습니다.

서버 프로세스

Manager() 가 반환한 관리자 객체는 파이썬 객체를 유지하고 다른 프로세스가 프락시를 사용하여 이 객체를 조작할 수 있게 하는 서버 프로세스를 제어합니다.

A manager returned by *Manager()* will support types *list*, *dict*, *set*, *Namespace*, *Lock*, *RLock*, *Semaphore*, *BoundedSemaphore*, *Condition*, *Event*, *Barrier*, *Queue*, *Value* and *Array*. For example,

```
from multiprocessing import Process, Manager

def f(d, l, s):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()
    s.add('a')
    s.add('b')

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))
        s = manager.set()

        p = Process(target=f, args=(d, l, s))
        p.start()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        p.join()

        print(d)
        print(l)
        print(s)
```

를 인쇄할 것입니다

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
{'a', 'b'}
```

서버 프로세스 관리자는 임의의 객체 형을 지원하도록 만들 수 있으므로 공유 메모리 객체를 사용하는 것보다 융통성이 있습니다. 또한, 단일 관리자를 네트워크를 통해 서로 다른 컴퓨터의 프로세스에서 공유 할 수 있습니다. 그러나 공유 메모리를 사용할 때보다 느립니다.

작업자 풀 사용

Pool 클래스는 작업자 프로세스 풀을 나타냅니다. 여기에는 몇 가지 다른 방법으로 작업을 작업자 프로세스로 넘길 수 있는 메서드가 있습니다.

예를 들면:

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))

        # print same numbers in arbitrary order
        for i in pool.imap_unordered(f, range(10)):
            print(i)

        # evaluate "f(20)" asynchronously
        res = pool.apply_async(f, (20,))      # runs in *only* one process
        print(res.get(timeout=1))             # prints "400"

        # evaluate "os.getpid()" asynchronously
        res = pool.apply_async(os.getpid, ()) # runs in *only* one process
        print(res.get(timeout=1))             # prints the PID of that process

        # launching multiple evaluations asynchronously *may* use more processes
        multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
        print([res.get(timeout=1) for res in multiple_results])

        # make a single worker sleep for 10 seconds
        res = pool.apply_async(time.sleep, (10,))
        try:
            print(res.get(timeout=1))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        except TimeoutError:
            print("We lacked patience and got a multiprocessing.TimeoutError")

        print("For the moment, the pool remains available for more work")

    # exiting the 'with'-block has stopped the pool
    print("Now the pool is closed and no longer available")
```

풀의 메서드는 풀을 만든 프로세스에서만 사용되어야 함에 유의하세요.

참고

이 패키지 내의 기능을 사용하려면 `__main__` 모듈을 자식이 임포트 할 수 있어야 합니다. 이것은 프로그래밍 지침에서 다루지만, 여기에서 지적할 가치가 있습니다. 이것은 몇몇 예제, 가령 *multiprocessing.pool.Pool* 예제가 대화형 인터프리터에서 동작하지 않음을 의미합니다. 예를 들면:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> with p:
...     p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
Traceback (most recent call last):
Traceback (most recent call last):
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_
↪importlib.BuiltinImporter'>)>
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_
↪importlib.BuiltinImporter'>)>
AttributeError: Can't get attribute 'f' on <module '__main__' (<class '_frozen_
↪importlib.BuiltinImporter'>)>
```

(이것을 시도해 보면 실제로 세 개의 전체 트레이스백이 어느 정도 임의로 번갈아 출력됩니다. 그런 다음 부모 프로세스를 중지시켜야 할 수도 있습니다.)

18.2.2 레퍼런스

multiprocessing 패키지는 대부분 *threading* 모듈의 API를 복제합니다.

Process와 예외

class multiprocessing.**Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, **, daemon=None*)

프로세스 객체는 별도의 프로세스에서 실행되는 작업을 나타냅니다. *Process* 클래스는 *threading.Thread* 의 모든 메서드와 같은 메서드를 갖습니다.

The constructor should always be called with keyword arguments. *group* should always be `None`; it exists solely for compatibility with *threading.Thread*. *target* is the callable object to be invoked by the *run()* method. It defaults to `None`, meaning nothing is called. *name* is the process name (see *name* for more details). *args* is the argument tuple for the target invocation. *kwargs* is a dictionary of keyword arguments for the target invocation. If provided, the keyword-only *daemon* argument sets the process *daemon* flag to `True` or `False`. If `None` (the default), this flag will be inherited from the creating process.

By default, no arguments are passed to *target*. The *args* argument, which defaults to `()`, can be used to specify a list or tuple of the arguments to pass to *target*.

If a subclass overrides the constructor, it must make sure it invokes the base class constructor (`super().__init__()`) before doing anything else to the process.

참고

In general, all arguments to `Process` must be picklable. This is frequently observed when trying to create a `Process` or use a `concurrent.futures.ProcessPoolExecutor` from a REPL with a locally defined *target* function.

Passing a callable object defined in the current REPL session causes the child process to die via an uncaught `AttributeError` exception when starting as *target* must have been defined within an importable module in order to be loaded during unpickling.

Example of this uncatchable error from the child:

```
>>> import multiprocessing as mp
>>> def knigit():
...     print("Ni!")
...
>>> process = mp.Process(target=knigit)
>>> process.start()
>>> Traceback (most recent call last):
  File ".../multiprocessing/spawn.py", line ..., in spawn_main
  File ".../multiprocessing/spawn.py", line ..., in _main
AttributeError: module '__main__' has no attribute 'knigit'
>>> process
<SpawnProcess name='SpawnProcess-1' pid=379473 parent=378707 stopped␣
↪exitcode=1>
```

See *spawn 과 forkserver 시작 방법*. While this restriction is not true if using the `"fork"` start method, as of Python `3.14` that is no longer the default on any platform. See 컨텍스트 및 시작 방법. See also gh-132898.

버전 3.3에서 변경: Added the *daemon* parameter.

run()

프로세스의 활동을 나타내는 메서드.

서브 클래스에서 이 메서드를 재정의할 수 있습니다. 표준 `run()` 메서드는 객체의 생성자에 target 인자로 전달된 콜러블 객체를 호출하는데 (있다면) *args* 와 *kwargs* 인자를 각각 위치 인자와 키워드 인자로 사용합니다.

Using a list or tuple as the *args* argument passed to `Process` achieves the same effect.

Example:

```
>>> from multiprocessing import Process
>>> p = Process(target=print, args=[1])
>>> p.run()
1
>>> p = Process(target=print, args=(1,))
>>> p.run()
1
```

start()

프로세스의 활동을 시작합니다.

이것은 프로세스 객체 당 최대 한 번 호출되어야 합니다. 객체의 `run()` 메서드가 별도의 프로세스에서 호출되도록 합니다.

join([*timeout*])

선택적 인자 *timeout* 이 None (기본값) 인 경우, 메서드는 *join()* 메서드가 호출된 프로세스가 종료될 때까지 블록 됩니다. *timeout* 이 양수면 최대 *timeout* 초 동안 블록 됩니다. 이 메서드는 프로세스가 종료되거나 메서드가 시간 초과 되면 None 을 돌려줌에 주의해야 합니다. 프로세스의 *exitcode* 를 검사하여 종료되었는지 확인하십시오.

프로세스는 여러 번 조인할 수 있습니다.

교착 상태를 유발할 수 있으므로 프로세스는 자신을 조인할 수 없습니다. 프로세스가 시작되기 전에 프로세스에 조인하려고 하면 에러가 발생합니다.

name

프로세스의 이름. 이름은 식별 목적으로만 사용되는 문자열입니다. 다른 의미는 없습니다. 여러 프로세스에 같은 이름이 주어질 수 있습니다.

초기 이름은 생성자에 의해 설정됩니다. 명시적 이름이 생성자에 제공되지 않으면, 'Process-N_1:N_2:…:N_k' 형식의 이름이 만들어지는데, 각각의 N_k 는 부모의 N 번째 자식입니다.

is_alive()

프로세스가 살아있는지 아닌지를 반환합니다.

대략, 프로세스 객체는 *start()* 메서드가 반환하는 순간부터 자식 프로세스가 종료될 때까지 살아있습니다.

daemon

프로세스의 데몬 플래그, 논리값. *start()* 가 호출되기 전에 설정되어야 합니다.

초깃값은 생성 프로세스에서 상속됩니다.

프로세스가 종료할 때, 모든 데몬 자식 프로세스를 강제 종료시키려고(terminate) 시도합니다.

데몬 프로세스는 하위 프로세스를 만들 수 없음에 유의하십시오. 그렇지 않으면 부모 프로세스가 종료될 때 데몬 프로세스가 강제 종료되어, 데몬 프로세스가 자식 프로세스를 고아로 남기게 됩니다. 또한, 이들은 유닉스 데몬이나 서비스가 **아닙니다**, 데몬이 아닌 프로세스들이 종료되면 강제 종료되는 (그리고 조인되지 않는) 일반 프로세스입니다.

threading.Thread API 외에도 *Process* 객체는 다음 어트리뷰트와 메서드도 지원합니다:

pid

프로세스 ID를 돌려줍니다. 프로세스가 스폰 되기 전에는 None 입니다.

exitcode

The child' s exit code. This will be None if the process has not yet terminated.

If the child' s *run()* method returned normally, the exit code will be 0. If it terminated via *sys.exit()* with an integer argument *N*, the exit code will be *N*.

If the child terminated due to an exception not caught within *run()*, the exit code will be 1. If it was terminated by signal *N*, the exit code will be the negative value -*N*.

authkey

프로세스의 인증 키 (바이트열) 입니다.

multiprocessing 이 초기화될 때, 메인 프로세스는 *os.urandom()* 을 사용하여 임의의 문자열을 할당받습니다.

Process 객체가 생성될 때, 부모 프로세스의 인증 키를 상속받습니다. *authkey* 를 다른 바이트열로 설정하여 변경할 수 있습니다.

인증 키를 참조하세요.

sentinel

프로세스가 끝나면 "준비(ready)" 될 시스템 객체의 숫자 핸들.

You can use this value if you want to wait on several events at once using *multiprocessing.connection.wait()*. Otherwise calling *join()* is simpler.

On Windows, this is an OS handle usable with the `WaitForSingleObject` and `WaitForMultipleObjects` family of API calls. On POSIX, this is a file descriptor usable with primitives from the *select* module.

Added in version 3.3.

interrupt()

Terminate the process. Works on POSIX using the *SIGINT* signal. Behavior on Windows is undefined.

By default, this terminates the child process by raising *KeyboardInterrupt*. This behavior can be altered by setting the respective signal handler in the child process *signal.signal()* for *SIGINT*.

Note: if the child process catches and discards *KeyboardInterrupt*, the process will not be terminated.

Note: the default behavior will also set *exitcode* to `1` as if an uncaught exception was raised in the child process. To have a different *exitcode* you may simply catch *KeyboardInterrupt* and call `exit(your_code)`.

Added in version 3.14.

terminate()

Terminate the process. On POSIX this is done using the *SIGTERM* signal; on Windows `TerminateProcess()` is used. Note that exit handlers and finally clauses, etc., will not be executed.

프로세스의 자손 프로세스들은 강제 종료되지 않을 것입니다 – 단순히 고아가 될 것입니다.

> **경고**
>
> 연결된 프로세스가 파이프 또는 큐를 사용할 때 이 메서드를 사용하면, 파이프 또는 큐가 손상되어 다른 프로세스에서 사용할 수 없게 될 수 있습니다. 마찬가지로, 프로세스가 록이나 세마포어 등을 획득한 경우 강제 종료하면 다른 프로세스가 교착 상태가 될 수 있습니다.

kill()

Same as *terminate()* but using the `SIGKILL` signal on POSIX.

Added in version 3.7.

close()

Process 객체를 닫아, 그것과 관련된 모든 자원을 해제합니다. 하부 프로세스가 여전히 실행 중이면 *ValueError* 가 발생합니다. 일단 *close()* 가 성공적으로 반환되면, *Process* 객체의 다른 대부분의 메서드와 어트리뷰트는 *ValueError* 를 발생시킵니다.

Added in version 3.7.

start(), *join()*, *is_alive()*, *terminate()* 및 *exitcode* 메서드는 프로세스 객체를 생성한 프로세스에 의해서만 호출되어야 합니다.

Process 의 몇몇 메서드를 사용하는 예제:

```
>>> import multiprocessing, time, signal
>>> mp_context = multiprocessing.get_context('spawn')
>>> p = mp_context.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<...Process ... initial> False
>>> p.start()
>>> print(p, p.is_alive())
<...Process ... started> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<...Process ... stopped exitcode=-SIGTERM> False
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.**ProcessError**

모든 *multiprocessing* 예외의 베이스 클래스입니다.

exception multiprocessing.**BufferTooShort**

Exception raised by `Connection.recv_bytes_into()` when the supplied buffer object is too small for the message read.

`e` 가 *BufferTooShort* 의 인스턴스라면, `e.args[0]` 는 메시지를 바이트열로 줍니다.

exception multiprocessing.**AuthenticationError**

인증 에러가 일어날 때 발생합니다.

exception multiprocessing.**TimeoutError**

시간제한이 초과하였을 때 시간제한을 건 메서드에 의해 발생합니다.

파이프와 큐

여러 프로세스를 사용할 때, 일반적으로 프로세스 간 통신을 위해 메시지 전달을 사용하고 록과 같은 동기화 프리미티브 사용을 피합니다.

메시지를 전달하기 위해 *Pipe()* (두 프로세스 간의 연결) 또는 큐(여러 생산자와 소비자를 허용합니다)를 사용할 수 있습니다.

Queue, *SimpleQueue* 그리고 *JoinableQueue* 형은, 표준 라이브러리의 *queue.Queue* 클래스에 따라 모델링 된, 다중 생산자, 다중 소비자 FIFO 큐입니다. 이것들은 파이썬 2.5의 *queue.Queue* 클래스에서 도입된 *task_done()*과 *join()* 메서드가 *Queue* 에 없다는 점에서 다릅니다.

*JoinableQueue*를 사용하면, 큐에서 제거된 작업마다 *JoinableQueue.task_done()*을 호출해야 합니다. 그렇지 않으면 완료되지 않은 작업의 수를 세는 데 사용되는 세마포어가 결국 오버플로 되어 예외를 일으킵니다.

One difference from other Python queue implementations, is that *multiprocessing* queues serializes all objects that are put into them using *pickle*. The object return by the get method is a re-created object that does not share memory with the original object.

관리자 객체를 사용하여 공유 큐를 생성할 수도 있습니다 – 관리자를 보세요.

> **참고**
>
> *multiprocessing* 은 제한 시간 초과 신호를 보내기 위해 보통 *queue.Empty* 와 *queue.Full* 예외를 사용합니다. *multiprocessing* 이름 공간에는 없으므로 *queue*에서 임포트 해야 합니다.

> **참고**
>
> 객체를 큐에 넣으면, 객체는 피클 되고 배경 스레드가 나중에 피클 된 데이터를 하부 파이프로 플러시 합니다. 이것은 다소 의외의 결과로 이어지지만, 실제적인 어려움을 일으키지는 않아야 합니다 – 이것이 여러분을 정말로 신경 쓰이게 한다면, 대신 관리자 로 만든 큐를 사용할 수 있습니다.
>
> (1) 빈 큐에 객체를 넣은 후에, *empty()* 메서드가 *False*를 반환하고 *get_nowait()*가 *queue.Empty*를 일으키지 않고 반환할 수 있기 전까지 극히 작은 지연이 있을 수 있습니다.
>
> (2) 여러 프로세스가 객체를 큐에 넣는 경우, 반대편에서 객체가 다른 순서로 수신될 수 있습니다. 그러나, 같은 프로세스에 의해 큐에 들어간 객체들은 항상 상대적인 순서가 유지됩니다.

> **경고**
>
> *Queue*를 사용하려고 하는 동안 *Process.terminate()* 또는 *os.kill()* 을 사용하여 프로세스를 죽이면, 큐의 데이터가 손상될 수 있습니다. 이로 인해 나중에 다른 프로세스가 큐를 사용하려고 할 때 예외가 발생할 수 있습니다.

> **경고**
>
> 위에서 언급했듯이, 자식 프로세스가 항목을 큐에 넣었을 때 (그리고 *JoinableQueue.cancel_join_thread*를 사용하지 않았다면), 버퍼링 된 모든 항목이 파이프로 플러시 될 때까지 해당 프로세스가 종료되지 않습니다.
>
> 이것은, 여러분이 그 자식 프로세스를 조인하려고 하면, 큐에 넣은 모든 항목을 소진하지 않는 한 교착 상태가 발생할 수 있다는 뜻입니다. 마찬가지로, 그 자식 프로세스가 데몬이 아니면 부모 프로세스가 종료 시점에 데몬이 아닌 모든 자식을 조인하려고 할 때 정지될 수 있습니다.
>
> 관리자를 사용하여 생성된 큐에는 이 문제가 없습니다. 프로그래밍 지침을 참조하세요.

프로세스 간 통신을 위해 큐를 사용하는 예는 예제을 참조하십시오.

multiprocessing.**Pipe**([*duplex*])

파이프의 끝을 나타내는 *Connection* 객체 쌍 (conn1, conn2) 를 반환합니다.

duplex 가 True (기본값) 면 파이프는 양방향입니다. *duplex* 가 False 인 경우 파이프는 단방향입니다: conn1 은 메시지를 받는 데에만 사용할 수 있고, conn2 는 메시지를 보낼 때만 사용할 수 있습니다.

The send() method serializes the object using *pickle* and the recv() re-creates the object.

class multiprocessing.**Queue**([*maxsize*])

파이프와 몇 개의 록/세마포어를 사용하여 구현된 프로세스 공유 큐를 반환합니다. 프로세스가 처음으로 항목을 큐에 넣으면 버퍼에서 파이프로 객체를 전송하는 피더 스레드가 시작됩니다.

제한 시간 초과를 알리기 위해 표준 라이브러리의 *queue* 모듈에서 정의되는 *queue.Empty* 와 *queue.Full* 예외를 일으킵니다.

Queue 는 *task_done()* 과 *join()* 을 제외한 *queue.Queue* 의 모든 메서드를 구현합니다.

qsize()

큐의 대략의 크기를 돌려줍니다. 다중 스레딩/다중 프로세싱 특성을 타기 때문에 이 숫자는 신뢰할 수 없습니다.

Note that this may raise *NotImplementedError* on platforms like macOS where sem_getvalue() is not implemented.

empty()

큐가 비어 있다면 True 를, 그렇지 않으면 False 를 반환합니다. 다중 스레딩/다중 프로세싱 특성을 타기 때문에 신뢰할 수 없습니다.

May raise an *OSError* on closed queues. (not guaranteed)

full()

큐가 가득 차면 True 를, 그렇지 않으면 False 를 반환합니다. 다중 스레딩/다중 프로세싱 특성을 타기 때문에 신뢰할 수 없습니다.

put(*obj*[, *block*[, *timeout*]])

obj를 큐에 넣습니다. 선택적 인자 *block* 이 True (기본값) 이고 *timeout* 이 None (기본값) 이면, 빈 슬롯이 생길 때까지 필요한 경우 블록합니다. *timeout* 이 양수인 경우, 최대 *timeout* 초만큼 블록하고 그 시간 내에 사용 가능 슬롯이 생기지 않으면 *queue.Full* 예외를 발생시킵니다. 그렇지 않으면 (*block* 이 False) 빈 슬롯을 즉시 사용할 수 있으면 큐에 항목을 넣고, 그렇지 않으면 *queue.Full* 예외를 발생시킵니다 (이 경우 *timeout* 은 무시됩니다).

버전 3.8에서 변경: 큐가 닫혔으면, *AssertionError* 대신 *ValueError*가 발생합니다.

put_nowait(*obj*)

put(obj, False) 와 같습니다.

get([*block*[, *timeout*]])

큐에서 항목을 제거하고 반환합니다. 선택적 인자 *block* 이 True (기본값) 이고 *timeout* 이 None (기본값) 이면, 항목이 들어올 때까지 필요한 경우 블록합니다. *timeout* 이 양수인 경우, 최대 *timeout* 초만큼 블록하고 그 시간 내에 항목이 들어오지 않으면 *queue.Empty* 예외를 발생시킵니다. 그렇지 않으면 (block 이 False) 즉시 사용할 수 있는 항목이 있으면 반환하고, 그렇지 않으면 *queue.Empty* 예외를 발생시킵니다 (이 경우 *timeout* 은 무시됩니다).

버전 3.8에서 변경: 큐가 닫혔으면, *OSError* 대신 *ValueError*가 발생합니다.

get_nowait()

get(False) 와 같습니다.

multiprocessing.Queue 에는 *queue.Queue* 에서 찾을 수 없는 몇 가지 추가 메서드가 있습니다. 일반적으로 이러한 메서드는 대부분 코드에서 필요하지 않습니다:

close()

큐를 닫습니다: 내부 자원을 해제합니다.

A queue must not be used anymore after it is closed. For example, *get()*, *put()* and *empty()* methods must no longer be called.

The background thread will quit once it has flushed all buffered data to the pipe. This is called automatically when the queue is garbage collected.

join_thread()

배경 스레드에 조인합니다. *close()* 가 호출된 후에만 사용할 수 있습니다. 배경 스레드가 종료될 때까지 블록해서 버퍼의 모든 데이터가 파이프로 플러시 되었음을 보증합니다.

기본적으로 프로세스가 큐를 만든 주체가 아니면 종료할 때 큐의 배경 스레드를 조인하려고 합니다. 프로세스는 *cancel_join_thread()*를 호출하여 *join_thread()* 가 아무것도 하지 않게 할 수 있습니다.

cancel_join_thread()

join_thread() 의 블록을 방지합니다. 특히, 프로세스가 종료할 때 배경 스레드를 자동으로 조인하는 것을 막습니다 – *join_thread()*를 보십시오.

A better name for this method might be allow_exit_without_flush(). It is likely to cause enqueued data to be lost, and you almost certainly will not need to use it. It is really only there if you need the current process to exit immediately without waiting to flush enqueued data to the underlying pipe, and you don' t care about lost data.

> **참고**
>
> 이 클래스의 기능은 호스트 운영 체제의 작동하는 공유 세마포어 구현을 요구합니다. 그런 것이 없으면, 클래스의 기능이 비활성화되고, *Queue* 의 인스턴스를 만들려고 하면 *ImportError* 를 일으킵니다. 자세한 내용은 bpo-3770을 참조하십시오. 아래에 나열된 특수 큐 형들도 마찬가지입니다.

class multiprocessing.**SimpleQueue**

이것은 단순화된 *Queue* 형으로, 록이 걸린 *Pipe* 에 매우 가깝습니다.

close()

큐를 닫습니다: 내부 자원을 해제합니다.

큐를 닫은 후에는 더는 사용해서는 안 됩니다. 예를 들어, *get()*, *put()* 및 *empty()* 메서드가 더는 호출되지 않아야 합니다.

Added in version 3.9.

empty()

큐가 비어 있다면 `True` 를, 그렇지 않으면 `False` 를 반환합니다.

Always raises an `OSError` if the SimpleQueue is closed.

get()

큐에서 항목을 제거하고 반환합니다.

put(*item*)

item 을 큐에 넣습니다.

class multiprocessing.**JoinableQueue**([*maxsize*])

`Queue` 서브 클래스 `JoinableQueue` 는 추가로 `task_done()` 과 `join()` 메서드를 가진 큐입니다.

task_done()

앞서 큐에 넣은 작업이 완료되었음을 나타냅니다. 큐 소비자가 사용합니다. 작업을 가져오는데 사용된 각 `get()` 마다, 뒤따르는 `task_done()` 호출은 작업에 대한 처리가 완료되었음을 큐에 알립니다.

만약 `join()` 이 현재 블록하고 있다면, 모든 항목이 처리될 때 재개될 것입니다 (`put()` 으로 큐에 넣은 모든 항목에 대해 `task_done()` 호출을 수신했다는 뜻입니다).

큐에 있는 항목보다 많이 호출되면 `ValueError` 를 발생시킵니다.

join()

큐의 모든 항목을 가져가서 처리할 때까지 블록합니다.

항목이 큐에 추가될 때마다 완료되지 않은 작업의 수는 올라갑니다. 소비자가 그 항목을 꺼냈고 그에 대한 모든 작업을 완료했음을 알리기 위해 `task_done()` 을 호출할 때마다 숫자는 줄어듭니다. 완료되지 않은 작업의 수가 0으로 떨어지면 `join()` 이 블록으로부터 풀려납니다.

잡동사니

multiprocessing.**active_children**()

현재 프로세스의 모든 살아있는 자식 리스트를 반환합니다.

이것을 호출하면 이미 완료된 프로세스에 "조인" 하는 부작용이 있습니다.

multiprocessing.**cpu_count**()

시스템의 CPU 수를 반환합니다.

This number is not equivalent to the number of CPUs the current process can use. The number of usable CPUs can be obtained with `os.process_cpu_count()` (or `len(os.sched_getaffinity(0))`).

When the number of CPUs cannot be determined a `NotImplementedError` is raised.

> **더 보기**
>
> `os.cpu_count()` `os.process_cpu_count()`

버전 3.13 에서 변경: The return value can also be overridden using the `-X cpu_count` flag or `PYTHON_CPU_COUNT` as this is merely a wrapper around the `os` cpu count APIs.

multiprocessing.**current_process**()

현재 프로세스에 해당하는 `Process` 객체를 반환합니다.

`threading.current_thread()` 와 유사한 기능을 제공합니다.

multiprocessing.**parent_process**()

`current_process()` 의 부모 프로세스에 해당하는 `Process` 객체를 반환합니다. 메인 프로세스에서, `parent_process` 는 `None` 입니다.

Added in version 3.8.

multiprocessing.**freeze_support**()

Add support for when a program which uses *multiprocessing* has been frozen to produce an executable. (Has been tested with **py2exe**, **PyInstaller** and **cx_Freeze**.)

메인 모듈의 `if __name__ == '__main__'` 줄 바로 뒤에서 이 함수를 호출해야 합니다. 예를 들면:

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

`freeze_support()` 줄이 생략된 경우 고정된 실행 파일을 실행하려고 하면 *RuntimeError* 가 발생합니다.

Calling `freeze_support()` has no effect when the start method is not *spawn*. In addition, if the module is being run normally by the Python interpreter (the program has not been frozen), then `freeze_support()` has no effect.

multiprocessing.**get_all_start_methods**()

Returns a list of the supported start methods, the first of which is the default. The possible start methods are `'fork'`, `'spawn'` and `'forkserver'`. Not all platforms support all methods. See 컨텍스트 및 시작 방법.

Added in version 3.4.

multiprocessing.**get_context**(*method=None*)

multiprocessing 모듈과 같은 어트리뷰트를 가진 컨텍스트 객체를 반환합니다.

If *method* is `None` then the default context is returned. Note that if the global start method has not been set, this will set it to the default method. Otherwise *method* should be `'fork'`, `'spawn'`, `'forkserver'`. *ValueError* is raised if the specified start method is not available. See 컨텍스트 및 시작 방법.

Added in version 3.4.

multiprocessing.**get_start_method**(*allow_none=False*)

프로세스를 기동하기 위해서 사용되는 시작 방법의 이름을 돌려줍니다.

If the global start method has not been set and *allow_none* is `False`, then the start method is set to the default and the name is returned. If the start method has not been set and *allow_none* is `True` then `None` is returned.

The return value can be `'fork'`, `'spawn'`, `'forkserver'` or `None`. See 컨텍스트 및 시작 방법.

Added in version 3.4.

버전 3.8에서 변경: macOS에서, *spawn* 시작 방법이 이제 기본값입니다. *fork* 시작 방법은 서브 프로세스의 충돌로 이어질 수 있기 때문에, 안전하지 않은 것으로 간주해야 합니다. bpo-33725를 참조하십시오.

multiprocessing.**set_executable**(*executable*)

Set the path of the Python interpreter to use when starting a child process. (By default *sys.executable* is used). Embedders will probably need to do some thing like

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

자식 프로세스를 만들기 전에 해야 합니다.

버전 3.4에서 변경: Now supported on POSIX when the `'spawn'` start method is used.

버전 3.11에서 변경: Accepts a *path-like object*.

multiprocessing.**set_forkserver_preload**(*module_names*)

Set a list of module names for the forkserver main process to attempt to import so that their already imported state is inherited by forked processes. Any `ImportError` when doing so is silently ignored. This can be used as a performance enhancement to avoid repeated work in every process.

For this to work, it must be called before the forkserver process has been launched (before creating a `Pool` or starting a `Process`).

Only meaningful when using the `'forkserver'` start method. See 컨텍스트 및 시작 방법.

Added in version 3.4.

multiprocessing.**set_start_method**(*method, force=False*)

Set the method which should be used to start child processes. The *method* argument can be `'fork'`, `'spawn'` or `'forkserver'`. Raises `RuntimeError` if the start method has already been set and *force* is not `True`. If *method* is `None` and *force* is `True` then the start method is set to `None`. If *method* is `None` and *force* is `False` then the context is set to the default context.

이것은 한 번만 호출해야 하며, 메인 모듈의 `if __name__ == '__main__'` 절 내에서 보호되어야 합니다.

See 컨텍스트 및 시작 방법.

Added in version 3.4.

> **참고**
>
> `multiprocessing` 에는 `threading.active_count()`, `threading.enumerate()`, `threading.settrace()`, `threading.setprofile()`, `threading.Timer` 또는 `threading.local` 의 대응 물이 없습니다.

Connection 객체

연결 객체를 사용하면 피클 가능한 객체나 문자열을 보내고 받을 수 있습니다. 메시지 지향 연결된 소켓으로 생각할 수 있습니다.

연결 객체는 보통 `Pipe` 를 사용해서 만들어집니다 – 리스너와 클라이언트 도 참고하세요.

class multiprocessing.connection.**Connection**

send(*obj*)

연결의 반대편 끝에서 `recv()`를 사용하여 읽을 객체를 보냅니다.

객체는 피클 가능해야 합니다. 매우 큰 피클(약 32 MiB+, OS에 따라 다릅니다)은 `ValueError` 예외를 발생시킬 수 있습니다.

recv()

연결의 반대편 끝에서 `send()`로 보낸 객체를 반환합니다. 뭔가 수신할 때까지 블록합니다. 수신할 내용이 없고 반대편 끝이 닫혔으면 `EOFError`를 발생시킵니다.

fileno()

연결이 사용하는 파일 기술자나 핸들을 돌려줍니다.

close()

연결을 닫습니다.

연결이 가비지 수집될 때 자동으로 호출됩니다.

poll([*timeout*])

읽어 들일 데이터가 있는지를 돌려줍니다.

timeout 을 지정하지 않으면 즉시 반환됩니다. *timeout* 이 숫자면 블록할 최대 시간(초)을 지정합니다. *timeout* 이 `None` 이면 시간제한이 없습니다.

여러 개의 연결 객체를 `multiprocessing.connection.wait()` 을 사용하여 한 번에 폴링 할 수 있습니다.

send_bytes(*buffer*[, *offset*[, *size*]])

바이트열류 객체 의 바이트 데이터를 하나의 완전한 메시지로 보냅니다.

offset 이 주어지면 *buffer* 의 해당 위치부터 데이터를 읽습니다. *size* 가 주어지면 그만큼의 바이트를 버퍼에서 읽습니다. 매우 큰 버퍼(약 32 MiB+, OS에 따라 다릅니다)는 `ValueError` 예외를 발생시킬 수 있습니다

recv_bytes([*maxlength*])

접속의 반대편 끝에서 송신된 바이트 데이터의 완전한 메시지를 문자열로 돌려줍니다. 뭔가 수신할 때까지 블록합니다. 수신할 내용이 없고 반대편 끝이 닫혔으면 `EOFError`를 발생시킵니다.

maxlength 가 지정되고 메시지가 *maxlength* 보다 길면 `OSError` 가 발생하고 연결은 더는 읽을 수 없게 됩니다.

버전 3.3에서 변경: 이 함수는 `IOError` 를 발생시켜왔는데, 이제는 `OSError` 의 별칭입니다.

recv_bytes_into(*buffer*[, *offset*])

연결의 반대편 끝에서 보낸 바이트 데이터의 전체 메시지를 *buffer* 로 읽어 들이고, 메시지의 바이트 수를 반환합니다. 뭔가 수신할 때까지 블록합니다. 수신할 내용이 없고 반대편 끝이 닫혔으면 `EOFError`를 발생시킵니다.

buffer 는 쓰기 가능한 바이트열류 객체 여야 합니다. *offset* 이 지정되면, 버퍼의 그 위치로부터 메시지를 씁니다. offset은 *buffer* 길이보다 작은 음수가 아닌 정수여야 합니다 (바이트 단위).

버퍼가 너무 작으면 `BufferTooShort` 예외가 발생하고, 완전한 메시지는 `e.args[0]` 으로 제공되는데, 여기서 `e` 는 예외 인스턴스입니다.

버전 3.3에서 변경: 이제 연결 객체 자체를 `Connection.send()` 와 `Connection.recv()` 를 사용하여 프로세스 간에 전송할 수 있습니다.

Connection objects also now support the context management protocol – see 컨텍스트 관리자 형. `__enter__()` returns the connection object, and `__exit__()` calls `close()`.

예를 들어:

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

경고

`Connection.recv()` 메서드는 수신한 데이터를 자동으로 언 피클 합니다. 메시지를 보낸 프로세스를 신뢰할 수 없다면 보안상 위험 할 수 있습니다.

따라서, 연결 객체가 `Pipe()` 를 사용하여 생성되지 않았다면, 일종의 인증을 수행한 후에만 `recv()` 및 `send()` 메서드를 사용해야 합니다. 인증 키를 참조하세요.

경고

프로세스가 파이프에 읽거나 쓰려고 할 때 죽으면, 파이프의 데이터가 손상될 가능성이 있습니다. 메시지 경계가 어디에 있는지 확신할 수 없는 상태가 될 가능성이 있기 때문입니다.

동기화 프리미티브

일반적으로 다중 프로세스 프로그램에서는 동기화 프리미티브가 다중 스레드 프로그램에서만큼 필요하지는 않습니다. `threading` 모듈에 대한 설명서를 참조하십시오.

관리자 객체를 사용하여 동기화 프리미티브를 생성할 수도 있습니다 – 관리자를 참조하세요.

class multiprocessing.**Barrier**(*parties*[, *action*[, *timeout*]])

배리어(barrier) 객체: `threading.Barrier` 의 복제본.

Added in version 3.3.

class multiprocessing.**BoundedSemaphore**([*value*])

제한된 세마포어 객체: `threading.BoundedSemaphore` 과 유사한 대응 물.

대응 물과 한 가지 차이가 있습니다: `acquire` 메서드의 첫 번째 인자에 *block* 이라는 이름을 사용해서 `Lock.acquire()` 와의 일관성을 유지합니다.

참고

On macOS, this is indistinguishable from `Semaphore` because `sem_getvalue()` is not implemented on that platform.

class multiprocessing.**Condition**([*lock*])

조건 변수: `threading.Condition` 의 별칭.

lock 을 지정할 때는 `multiprocessing` 의 `Lock` 이나 `RLock` 객체여야 합니다.

버전 3.3에서 변경: `wait_for()` 메서드가 추가되었습니다.

class multiprocessing.**Event**

`threading.Event` 의 복제본.

class multiprocessing.**Lock**

비 재귀적 록 객체: `threading.Lock` 과 유사한 대응 물. 일단 프로세스 또는 스레드가 록을 획득하면, 프로세스 또는 스레드에서 록을 획득하려는 후속 시도는 록이 해제될 때까지 블록 됩니다; 모든 프로세스 또는 스레드가 이를 해제할 수 있습니다. 스레드에 적용되는 `threading.Lock` 의 개념과 동작은, 명시된 경우를 제외하고, `multiprocessing.Lock` 를 통해 프로세스나 스레드에 그대로 적용됩니다.

`Lock` 은 실제로 기본 컨텍스트로 초기화된 `multiprocessing.synchronize.Lock` 의 인스턴스를 반환하는 팩토리 함수입니다.

`Lock` 은 컨텍스트 관리자 프로토콜을 지원하므로 `with` 문에서 사용될 수 있습니다.

acquire(*block=True*, *timeout=None*)

블록하거나 블록하지 않는 방식으로 록을 획득합니다.

block 인자가 `True` (기본값) 로 설정되면, 메서드 호출은 록이 해제 상태가 될 때까지 블록 한 다음, 잠금 상태로 만들고 `True` 를 반환합니다. 이 첫 번째 인자의 이름은 `threading.Lock.acquire()` 와 다르다는 것에 유의하세요.

block 인자가 `False` 로 설정되면, 메서드 호출은 블록 되지 않습니다. 록이 현재 잠금 상태면 `False` 를 반환합니다. 그렇지 않으면 록을 잠금 상태로 설정하고 `True` 를 반환합니다.

timeout 에 대해 양의 부동 소수점 값을 사용하여 호출하는 경우, 록을 얻을 수 없는 한 최대 *timeout* 으로 지정된 시간(초) 동안 블록합니다. *timeout* 을 음수 값으로 호출하는 것은 *timeout* 에 0을 주는 것과 같습니다. *timeout* 값이 `None` (기본값) 인 호출은 제한 시간을 무한대로 설정합니다. *timeout* 에 대한 음수와 `None` 값의 처리는 *threading.Lock.acquire()* 에서 구현된 동작과 다르다는 것에 주의하십시오. *timeout* 인자는 *block* 인자가 `False` 로 설정되면 실제적인 의미는 없고 무시됩니다. 록이 획득되면 `True` 를 돌려주고, 제한 시간 초과가 발생하면 `False` 를 돌려줍니다.

release()

록을 해제합니다. 이것은 원래 록을 획득한 프로세스나 스레드뿐만 아니라 모든 프로세스나 스레드에서 호출 할 수 있습니다.

동작은 *threading.Lock.release()* 와 같지만, 해제된 록에서 호출될 때 *ValueError* 가 발생한다는 점만 다릅니다.

locked()

Return a boolean indicating whether this object is locked right now.

Added in version 3.14.

class multiprocessing.**RLock**

재귀적 록 객체: *threading.RLock* 과 유사한 대응 물. 재귀적 록은 획득한 프로세스 또는 스레드에 의해 해제되어야 합니다. 일단 프로세스나 스레드가 재귀적 록을 획득하면, 같은 프로세스나 스레드가 블록 없이 다시 획득할 수 있습니다; 해당 프로세스나 스레드는 획득할 때마다 한 번 해제해야 합니다.

RLock 은 실제로 기본 컨텍스트로 초기화된 `multiprocessing.synchronize.RLock` 의 인스턴스를 반환하는 팩토리 함수입니다.

RLock 은 컨텍스트 관리자 프로토콜을 지원하므로 `with` 문에서 사용될 수 있습니다.

acquire(*block=True*, *timeout=None*)

블록하거나 블록하지 않는 방식으로 록을 획득합니다.

block 인자를 `True` 로 설정해서 호출하면, 록이 현재 프로세스나 스레드가 이미 획득한 상태가 아니면 록이 (어떤 프로세스나 스레드도 획득하지 않은) 록 해제 상태가 될 때까지 블록합니다. 이후에 현재 프로세스나 스레드가 (소유권이 아직 없는 경우) 록 소유권을 얻게 되며 록 내 재귀 수준이 1 증가하고 `True` 를 반환합니다. 이 첫 번째 인자의 동작에는, 인자의 이름부터 시작해서 *threading.RLock.acquire()* 구현과 비교되는 몇 가지 차이점이 있습니다.

block 인자를 `False` 로 설정해서 호출하면 블록하지 않습니다. 록이 이미 다른 프로세스나 스레드에 의해 획득되었으면 (그래서 소유하고 있으면), 현재 프로세스나 스레드는 소유권을 갖지 않으며 록 내 재귀 수준은 변경되지 않고 `False` 를 반환합니다. 록이 해제 상태에 있으면, 현재 프로세스 또는 스레드가 소유권을 가져오며 재귀 수준이 증가하고 `True` 를 반환합니다.

timeout 인자의 사용법과 동작은 *Lock.acquire()* 와 같습니다. *timeout* 의 이러한 동작 중 일부는 *threading.RLock.acquire()* 에서 구현된 동작과 다르다는 것에 주의하십시오.

release()

재귀 수준을 감소시키면서 록을 해제합니다. 감소 후에 재귀 수준이 0이면, 록을 해제 상태 (어떤 프로세스나 스레드에도 소유되지 않음)로 재설정하고, 다른 프로세스나 스레드가 록이 해제될 때까지 기다리며 블록하고 있는 경우 해당 프로세스나 스레드 중 정확히 하나가 계속 진행하도록 허용합니다. 감소 후에 재귀 수준이 여전히 0이 아닌 경우, 록은 획득된 상태로 남고 호출한 프로세스나 스레드에 의해 소유됩니다.

호출한 프로세스나 스레드가 록을 소유하고 있을 때만 이 메서드를 호출하십시오. 이 메서드가 소유자가 아닌 프로세스나 스레드에 의해 호출되거나, 록이 해제 (소유되지 않은) 상태면 *AssertionError* 가 발생합니다. 이 상황에서 발생하는 예외 형은 *threading.RLock.release()* 에서 구현된 동작과 다릅니다.

locked()

Return a boolean indicating whether this object is locked right now.

Added in version 3.14.

class multiprocessing.**Semaphore**([*value*])

세마포어 객체: `threading.Semaphore` 와 유사한 대응 물.

대응 물과 한 가지 차이가 있습니다: `acquire` 메서드의 첫 번째 인자에 *block* 이라는 이름을 사용해서 `Lock.acquire()` 와의 일관성을 유지합니다.

> **참고**
>
> On macOS, `sem_timedwait` is unsupported, so calling `acquire()` with a timeout will emulate that function's behavior using a sleeping loop.

> **참고**
>
> 이 패키지의 기능 중 일부는 호스트 운영 체제의 작동하는 공유 세마포어 구현을 요구합니다. 그런 것이 없으면, `multiprocessing.synchronize` 모듈이 비활성화되고, 임포트하려고 하면 `ImportError` 를 일으킵니다. 자세한 내용은 bpo-3770을 참조하십시오.

공유 ctypes 객체

자식 프로세스가 상속할 수 있는 공유 메모리를 사용하여 공유 객체를 만들 수 있습니다.

multiprocessing.**Value**(*typecode_or_type*, **args*, *lock=True*)

공유 메모리에 할당된 `ctypes` 객체를 반환합니다. 기본적으로 반환 값은, 사실 객체에 대한 동기화된 래퍼입니다. 객체 자체는 `Value` 의 *value* 어트리뷰트를 통해 접근 할 수 있습니다.

typecode_or_type 은 반환된 객체의 형을 결정합니다: ctypes 형이거나 `array` 모듈에 의해 사용되는 종류의 한 문자 typecode 입니다. **args* 는 형의 생성자로 전달됩니다.

lock 이 `True` (기본값) 면 값에 대한 액세스를 동기화하기 위해 새 재귀적 록 객체가 생성됩니다. *lock* 이 `Lock` 또는 `RLock` 객체인 경우, 이 값이 값에 대한 액세스를 동기화하는 데 사용됩니다. *lock* 이 `False` 면, 반환된 객체에 대한 액세스는 록에 의해 자동으로 보호되지 않으므로 "프로세스 안전" 하지 않습니다.

읽기와 쓰기를 포함하는 `+=` 와 같은 연산은 원자 적(atomic) 이지 않습니다. 따라서, 예를 들어, 공유 값을 원자 적으로 증가시키려면, 다음과 같이 하는 것으로는 충분하지 않습니다:

```
counter.value += 1
```

연관된 록이 재귀적이라고 가정하면 (기본적으로 그렇습니다), 대신 다음과 같이 할 수 있습니다

```
with counter.get_lock():
    counter.value += 1
```

lock 은 키워드 전용 인자입니다.

multiprocessing.**Array**(*typecode_or_type*, *size_or_initializer*, ***, *lock=True*)

공유 메모리에서 할당된 ctypes 배열을 반환합니다. 기본적으로 반환 값은, 사실 배열에 대한 동기화된 래퍼입니다.

typecode_or_type 은 반환된 배열의 요소의 형을 결정합니다: ctypes 형이거나 `array` 모듈에 의해 사용되는 종류의 한 문자 typecode 입니다. *size_or_initializer* 가 정수면, 배열의 길이를 결정하고 배열은 0으로 초기화됩니다. 그렇지 않으면, *size_or_initializer* 는 배열을 초기화하는 데 사용되는 시퀀스고, 길이는 배열의 길이를 결정합니다.

lock 이 `True` (기본값) 면 값에 대한 액세스를 동기화하기 위해 새 록 객체가 생성됩니다. *lock* 이 `Lock` 또는 `RLock` 객체인 경우, 이 값이 값에 대한 액세스를 동기화하는 데 사용됩니다. *lock* 이 `False` 면, 반환된 객체에 대한 액세스는 록에 의해 자동으로 보호되지 않으므로 "프로세스 안전" 하지 않습니다.

lock 은 키워드 전용 인자입니다.

`ctypes.c_char` 의 배열은 *value* 와 *raw* 어트리뷰트를 가지고 있습니다. 이 어트리뷰트를 사용하여 문자열을 저장하고 꺼낼 수 있습니다.

`multiprocessing.sharedctypes` 모듈

`multiprocessing.sharedctypes` 모듈은 자식 프로세스에 의해 상속될 수 있는 공유 메모리에 `ctypes` 객체를 할당하는 기능을 제공합니다.

> **참고**
>
> 공유 메모리에 포인터를 저장할 수는 있지만, 특정 프로세스의 주소 공간에 있는 위치를 참조하게 됩니다. 그러나 포인터는 두 번째 프로세스의 컨텍스트에서는 유효하지 않을 가능성이 커서, 두 번째 프로세스에서 포인터를 역 참조하려고 하면 충돌이 일어날 수 있습니다.

`multiprocessing.sharedctypes.`**`RawArray`**(*typecode_or_type*, *size_or_initializer*)

공유 메모리에 할당된 ctypes 배열을 반환합니다.

typecode_or_type 은 반환된 배열의 요소의 형을 결정합니다: ctypes 형이거나 `array` 모듈에 의해 사용되는 종류의 한 문자 typecode 입니다. *size_or_initializer* 가 정수면, 배열의 길이를 결정하고 배열은 0으로 초기화됩니다. 그렇지 않으면, *size_or_initializer* 는 배열을 초기화하는 데 사용되는 시퀀스고, 길이는 배열의 길이를 결정합니다.

요소를 쓰고 읽는 것은 잠재적으로 원자 적이지 않습니다 – 액세스가 록을 사용하여 자동으로 동기화되기 원하면 `Array()` 를 대신 사용하세요.

`multiprocessing.sharedctypes.`**`RawValue`**(*typecode_or_type*, **args*)

공유 메모리에 할당된 ctypes 객체를 반환합니다.

typecode_or_type 은 반환된 객체의 형을 결정합니다: ctypes 형이거나 `array` 모듈에 의해 사용되는 종류의 한 문자 typecode 입니다. **args* 는 형의 생성자로 전달됩니다.

값을 쓰고 읽는 것은 잠재적으로 원자 적이지 않습니다 – 액세스가 록을 사용하여 자동으로 동기화되기 원하면 `Value()` 를 대신 사용하세요.

`ctypes.c_char` 의 배열은 *value* 와 *raw* 어트리뷰트를 가지고 있습니다. 이 어트리뷰트를 사용하여 문자열을 저장하고 꺼낼 수 있습니다 – `ctypes` 설명서를 보십시오.

`multiprocessing.sharedctypes.`**`Array`**(*typecode_or_type*, *size_or_initializer*, **, *lock=True*)

lock 값에 따라, 날 ctypes 배열 대신 프로세스 안전한 동기화 래퍼가 반환될 수 있다는 것을 제외하고는 `RawArray()` 와 같습니다.

lock 이 `True` (기본값) 면 값에 대한 액세스를 동기화하기 위해 새 록 객체가 생성됩니다. *lock* 이 `Lock` 또는 `RLock` 객체인 경우, 이 값이 값에 대한 액세스를 동기화하는 데 사용됩니다. *lock* 이 `False` 면, 반환된 객체에 대한 액세스는 록에 의해 자동으로 보호되지 않으므로 "프로세스 안전" 하지 않습니다.

lock 은 키워드 전용 인자입니다.

`multiprocessing.sharedctypes.`**`Value`**(*typecode_or_type*, **args*, *lock=True*)

lock 값에 따라, 날 ctypes 객체 대신 프로세스 안전한 동기화 래퍼가 반환될 수 있다는 것을 제외하고는 `RawValue()` 와 같습니다.

lock 이 `True` (기본값) 면 값에 대한 액세스를 동기화하기 위해 새 록 객체가 생성됩니다. *lock* 이 `Lock` 또는 `RLock` 객체인 경우, 이 값이 값에 대한 액세스를 동기화하는 데 사용됩니다. *lock* 이 `False` 면, 반환된 객체에 대한 액세스는 록에 의해 자동으로 보호되지 않으므로 "프로세스 안전" 하지 않습니다.

lock 은 키워드 전용 인자입니다.

multiprocessing.sharedctypes.**copy**(*obj*)

공유 메모리에서 할당된 ctypes 객체를 반환합니다. 이 객체는 ctypes 객체 *obj* 의 복사본입니다.

multiprocessing.sharedctypes.**synchronized**(*obj*[, *lock*])

lock 을 사용하여 액세스를 동기화하는 ctypes 객체에 대한 프로세스 안전한 래퍼 객체를 반환합니다. *lock* 이 None (기본값) 이면 *multiprocessing.RLock* 객체가 자동으로 생성됩니다.

동기화 래퍼는 래핑 된 객체의 메서드 외에도 두 개의 메서드를 더 갖습니다: get_obj() 는 래핑 된 객체를 반환하고, get_lock() 은 동기화에 사용되는 록 객체를 반환합니다.

래퍼를 통해 ctypes 객체에 액세스하는 것은, 날 ctypes 객체에 액세스하는 것보다 훨씬 느릴 수 있습니다.

버전 3.5에서 변경: 동기화된 객체는 컨텍스트 관리자 프로토콜을 지원합니다.

아래 표는 공유 메모리에 공유 ctypes 객체를 만드는 문법과 일반적인 ctypes 문법을 비교합니다. (표에서 MyStruct 는 *ctypes.Structure* 의 서브 클래스입니다.)

ctypes	type을 사용하는 공유 ctypes	typecode를 사용하는 공유 ctypes
c_double(2.4)	RawValue(c_double, 2.4)	RawValue('d' , 2.4)
MyStruct(4, 6)	RawValue(MyStruct, 4, 6)	
(c_short * 7)()	RawArray(c_short, 7)	RawArray('h' , 7)
(c_int * 3)(9, 2, 8)	RawArray(c_int, (9, 2, 8))	RawArray('i' , (9, 2, 8))

다음은 자식 프로세스가 여러 ctypes 객체를 수정하는 예입니다:

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print(n.value)
    print(x.value)
    print(s.value)
    print([(a.x, a.y) for a in A])
```

인쇄되는 결과는 이렇습니다

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

관리자

관리자는 서로 다른 컴퓨터에서 실행되는 프로세스 간에 네트워크를 통해 공유하는 것을 포함하여 서로 다른 프로세스 간에 공유할 수 있는 데이터를 만드는 방법을 제공합니다. 관리자 객체는 공유 객체 를 관리하는 서버 프로세스를 제어합니다. 다른 프로세스는 프락시를 사용하여 공유 객체에 액세스 할 수 있습니다.

multiprocessing.**Manager**()

프로세스 간에 객체를 공유하는 데 사용할 수 있는 시작된 *SyncManager* 객체를 반환합니다. 반환된 관리자 객체는 생성된 자식 프로세스에 해당하며 공유 객체를 만들고 해당 프락시를 반환하는 메서드가 있습니다.

관리자 프로세스는 가비지 수집되거나 상위 프로세스가 종료되자마자 종료됩니다. 관리자 클래스는 *multiprocessing.managers* 모듈에 정의되어 있습니다:

class multiprocessing.managers.**BaseManager**(*address=None, authkey=None, serializer='pickle', ctx=None, *, shutdown_timeout=1.0*)

BaseManager 객체를 만듭니다.

일단 생성되면 관리자 객체가 시작된 관리자 프로세스를 참조하게 하려고 *start()* 또는 `get_server().serve_forever()` 를 호출해야 합니다.

address 는 관리자 프로세스가 새 연결을 리슨하는 주소입니다. *address* 가 `None` 이면 임의의 것이 선택됩니다.

authkey 는 서버 프로세스로 들어오는 연결의 유효성을 검사하는 데 사용되는 인증 키입니다. *authkey* 가 `None` 이면 `current_process().authkey` 가 사용됩니다. 그렇지 않으면 *authkey* 가 사용되며 바이트열이어야 합니다.

serializer must be `'pickle'` (use *pickle* serialization) or `'xmlrpclib'` (use *xmlrpc.client* serialization).

ctx is a context object, or `None` (use the current context). See the `get_context()` function.

shutdown_timeout is a timeout in seconds used to wait until the process used by the manager completes in the *shutdown()* method. If the shutdown times out, the process is terminated. If terminating the process also times out, the process is killed.

버전 3.11 에서 변경: Added the *shutdown_timeout* parameter.

start([*initializer*[, *initargs*]])

관리자를 시작시키기 위해 서브 프로세스를 시작합니다. *initializer* 가 `None` 이 아닌 경우, 서브 프로세스는 시작할 때 `initializer(*initargs)` 를 호출합니다.

get_server()

Manager 의 제어를 받는 실제 서버를 나타내는 `Server` 객체를 반환합니다. `Server` 객체는 `serve_forever()` 메서드를 지원합니다:

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey=b'abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

`Server` 는 추가로 *address* 어트리뷰트를 가지고 있습니다.

connect()

지역 관리자 객체를 원격 관리자 프로세스에 연결합니다:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 50000), authkey=b'abc')
>>> m.connect()
```

shutdown()

관리자가 사용하는 프로세스를 중지합니다. *start()*를 사용하여 서버 프로세스를 시작한 경우에만 사용할 수 있습니다.

여러 번 호출 될 수 있습니다.

register(*typeid*[, *callable*[, *proxytype*[, *exposed*[, *method_to_typeid*[, *create_method*]]]]])

관리자 클래스에 형이나 콜러블을 등록하는데 사용할 수 있는 클래스 메서드.

typeid 는 특정 형의 공유 객체를 식별하는 데 사용되는 "형 식별자" 입니다. 문자열이어야 합니다.

callable 은 이 형 식별자에 대한 객체를 만드는 데 사용되는 콜러블 객체입니다. 관리자 인스턴스가 *connect()* 메서드를 사용하여 서버에 연결되거나, *create_method* 인자가 `False` 면 `None` 으로 남겨 둘 수 있습니다.

proxytype 은, 이 *typeid* 의 공유 객체의 프락시를 만드는 데 사용되는 *BaseProxy* 의 서브 클래스입니다. `None` 이면 프락시 클래스가 자동으로 생성됩니다.

exposed 는 이 typeid에 대한 프락시가 *BaseProxy._callmethod()* 를 사용하여 액세스 할 수 있도록 허용해야 하는 메서드 이름의 시퀀스를 지정하는 데 사용됩니다. (만약 *exposed* 가 `None` 이면, 존재하는 경우, `proxytype._exposed_` 가 대신 사용됩니다.) exposed 리스트가 지정되지 않은 경우, 공유 객체의 모든 "공용 메서드" 에 액세스 할 수 있습니다. (여기서 "공용 메서드" 는 `__call__()` 메서드가 있고 그 이름이 `'_'` 로 시작하지 않는 어트리뷰트를 의미합니다.)

method_to_typeid 는 프락시를 반환해야 하는 노출된 메서드의 반환형을 지정하는 데 사용되는 매핑입니다. 메서드 이름을 typeid 문자열로 매핑합니다. (만일 *method_to_typeid* 가 `None` 이면, 존재한다면, `proxytype._method_to_typeid_` 가 대신 사용됩니다.) 메서드의 이름이 이 매핑의 키가 아니거나 매핑이 `None` 이면, 메서드에 의해 반환된 객체는 값으로 복사됩니다.

create_method 는 이름이 *typeid* 인 메서드를 만들어야 하는지를 결정합니다. 이 메서드는 서버 프로세스에 새 공유 객체를 만들고 프락시를 반환하도록 지시하는 데 사용될 수 있습니다. 기본적으로 `True` 입니다.

BaseManager 인스턴스는 읽기 전용 프로퍼티를 하나 가지고 있습니다:

address

관리자가 사용하는 주소.

버전 3.3에서 변경: 관리자 객체는 컨텍스트 관리 프로토콜을 지원합니다 – 컨텍스트 관리자 형를 보세요. *__enter__()* 는 서버 프로세스를 시작하고 (아직 시작하지 않았다면), 관리자 객체를 반환합니다. *__exit__()* 는 *shutdown()*을 호출합니다.

이전 버전에서 *__enter__()* 는 관리자의 서버 프로세스가 아직 시작되지 않았을 때 시작시키지 않았습니다.

class multiprocessing.managers.**SyncManager**

프로세스의 동기화에 사용할 수 있는 *BaseManager* 의 서브 클래스입니다. 이 형의 객체는 *multiprocessing.Manager()* 에 의해 반환됩니다.

이 클래스의 메서드는 여러 프로세스에서 동기화 할 수 있도록 일반적으로 사용되는 많은 데이터형을 생성하고 프락시 객체를 반환합니다. 특히 공유 리스트와 딕셔너리가 포함됩니다.

Barrier(*parties*[, *action*[, *timeout*]])

공유 *threading.Barrier* 객체를 생성하고 프락시를 반환합니다.

Added in version 3.3.

BoundedSemaphore([*value*])

공유 *threading.BoundedSemaphore* 객체를 생성하고 프락시를 반환합니다.

Condition([*lock*])

공유 `threading.Condition` 객체를 생성하고 프락시를 반환합니다.

lock 이 제공되면 `threading.Lock` 또는 `threading.RLock` 객체에 대한 프락시여야 합니다.

버전 3.3에서 변경: `wait_for()` 메서드가 추가되었습니다.

Event()

공유 `threading.Event` 객체를 생성하고 프락시를 반환합니다.

Lock()

공유 `threading.Lock` 객체를 생성하고 프락시를 반환합니다.

Namespace()

공유 `Namespace` 객체를 생성하고 프락시를 반환합니다.

Queue([*maxsize*])

공유 `queue.Queue` 객체를 생성하고 프락시를 반환합니다.

RLock()

공유 `threading.RLock` 객체를 생성하고 프락시를 반환합니다.

Semaphore([*value*])

공유 `threading.Semaphore` 객체를 생성하고 프락시를 반환합니다.

Array(*typecode*, *sequence*)

배열을 만들고 프락시를 반환합니다.

Value(*typecode*, *value*)

쓰기 가능한 `value` 어트리뷰트를 가진 객체를 생성하고 프락시를 반환합니다.

dict()

dict(*mapping*)

dict(*sequence*)

공유 `dict` 객체를 생성하고 프락시를 반환합니다.

list()

list(*sequence*)

공유 `list` 객체를 생성하고 프락시를 반환합니다.

set()

set(*sequence*)

set(*mapping*)

Create a shared `set` object and return a proxy for it.

Added in version 3.14: `set` support was added.

버전 3.6에서 변경: 공유 객체는 중첩될 수 있습니다. 예를 들어, 공유 리스트와 같은 공유 컨테이너 객체는, `SyncManager` 에 의해 모두 관리되고 동기화되는 다른 공유 객체를 포함 할 수 있습니다.

class multiprocessing.managers.**Namespace**

`SyncManager` 로 등록 할 수 있는 형입니다.

이름 공간 객체에는 공용 메서드가 없지만, 쓰기 가능한 어트리뷰트가 있습니다. repr 은 그것의 어트리뷰트 값을 보여줍니다.

그러나, 이름 공간 객체의 프락시를 사용할 때, `'_'` 로 시작하는 어트리뷰트는 프락시의 어트리뷰트가 되며 참조 대상의 어트리뷰트가 아닙니다:

```
>>> mp_context = multiprocessing.get_context('spawn')
>>> manager = mp_context.Manager()
>>> Global = manager.Namespace()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3    # this is an attribute of the proxy
>>> print(Global)
Namespace(x=10, y='hello')
```

사용자 정의 관리자

자신만의 관리자를 만들려면, *BaseManager* 의 서브 클래스를 만들고 *register()* 클래스 메서드를 사용하여 새로운 형이나 콜러블을 관리자 클래스에 등록합니다. 예를 들면:

```
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))         # prints 7
        print(maths.mul(7, 8))         # prints 56
```

원격 관리자 사용하기

한 기계에서 관리자 서버를 실행하고 다른 기계의 클라이언트가 관리자 서버를 사용하도록 할 수 있습니다 (관련된 방화벽이 허용한다고 가정합니다).

다음 명령을 실행하면 원격 클라이언트가 액세스 할 수 있는 단일 공유 큐를 위한 서버가 만들어집니다:

```
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

한 클라이언트는 다음과 같이 서버에 액세스 할 수 있습니다:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

또 다른 클라이언트도 사용할 수 있습니다:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

지역 프로세스 역시, 위의 클라이언트가 원격으로 액세스하는 코드를 사용하여 같은 큐에 액세스 할 수 있습니다:

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super().__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

프락시 객체

프락시는 (아마도) 다른 프로세스에 있는 공유 객체를 가리키는 객체입니다. 공유 객체는 프락시의 지시 대상 이라고 합니다. 여러 프락시 객체는 같은 지시 대상을 가질 수 있습니다.

프락시 객체에는 지시 대상의 해당 메서드를 호출하는 메서드가 있습니다 (그러나 지시 대상의 모든 메서드가 반드시 프락시를 통해 사용할 수 있는 것은 아닙니다). 이런 식으로, 프락시는 지시 대상처럼 사용될 수 있습니다:

```
>>> mp_context = multiprocessing.get_context('spawn')
>>> manager = mp_context.Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print(repr(l))
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

프락시에 *str()* 을 적용하면 지시 대상의 표현이 반환되는 반면, *repr()* 을 적용하면 프락시의 표현이 반환됩니다.

프락시 객체의 중요한 특징은, 피클 가능해서 프로세스 간에 전달될 수 있다는 것입니다. 지시 대상은 프락시 객체를 포함 할 수 있습니다. 이것은 관리된 리스트, 딕셔너리 및 다른 프락시 객체 의 중첩을 허용합니다:

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)         # referent of a now contains referent of b
>>> print(a, b)
[<ListProxy object, typeid 'list' at ...>] []
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

비슷하게, 딕셔너리와 리스트 프락시는 서로 중첩될 수 있습니다:

```
>>> l_outer = manager.list([ manager.dict() for i in range(2) ])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
>>> l_outer[1]['z'] = 26
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

(프락시가 아닌) 표준 *list* 또는 *dict* 객체가 지시 대상에 포함되어있는 경우, 이 가변 값들에 대한 수정은 관리자를 통해 전파되지 않습니다. 포함된 값이 언제 수정되는지 프락시가 알 방법이 없기 때문입니다. 그러나 컨테이너 프락시에 값을 저장하는 것(프락시 객체의 `__setitem__` 을 호출합니다)은 관리자를 통해 전파되므로, 그 항목을 효과적으로 수정하기 위해, 수정된 값을 컨테이너 프락시에 다시 대입할 수 있습니다:

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# updating the dictionary, the proxy is notified of the change
lproxy[0] = d
```

이 접근법은 아마도 대부분의 사용 사례에서 중첩된 프락시 객체를 사용하는 것보다 불편하지만, 동기화에 대한 제어 수준을 보여줍니다.

참고

multiprocessing 의 프락시 형은 값으로 비교하는 것을 지원하지 않습니다. 그래서, 예를 들어, 이런 결과를 얻습니다:

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

비교할 때는 지시 대상의 사본을 대신 사용해야 합니다.

class multiprocessing.managers.**BaseProxy**

프락시 객체는 *BaseProxy* 의 서브 클래스의 인스턴스입니다.

_callmethod(*methodname*[, *args*[, *kwds*]])

프락시의 지시 대상 메서드를 호출하고 결과를 반환합니다.

proxy 가 프락시이고, 그 지시 대상이 obj 면, 표현식

```
proxy._callmethod(methodname, args, kwds)
```

은 표현식

```
getattr(obj, methodname)(*args, **kwds)
```

을 관리자 프로세스에서 평가합니다.

반환된 값은 호출 결과의 복사본이거나 새 공유 객체에 대한 프락시입니다 – *BaseManager.register()* 의 *method_to_typeid* 인자에 대한 설명서를 보십시오.

호출 때문에 예외가 발생하면, *_callmethod()* 가 다시 일으킵니다. 관리자 프로세스에서 다른 예외가 발생하면 RemoteError 예외로 변환되어 *_callmethod()* 가 일으킵니다.

특히, *methodname* 이 노출되지 않았으면 예외가 발생합니다.

_callmethod() 사용법의 예:

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),)) # equivalent to l[2:7]
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))          # equivalent to l[20]
Traceback (most recent call last):
...
IndexError: list index out of range
```

_getvalue()

지시 대상의 복사본을 반환합니다.

지시 대상이 피클 가능하지 않으면 예외가 발생합니다.

__repr__()

프락시 객체의 표현을 반환합니다.

__str__()

지시 대상의 표현을 반환합니다.

정리

프락시 객체는 weakref 콜백을 사용해서 가비지 수집 시 자신의 지시 대상을 소유한 관리자에서 자신을 등록 취소합니다.

더는 참조하는 프락시가 없는 경우 공유 객체는 관리자 프로세스에서 삭제됩니다.

프로세스 풀

Pool 클래스를 사용하여, 제출된 작업을 수행할 프로세스 풀을 만들 수 있습니다.

class multiprocessing.pool.**Pool**([*processes*[, *initializer*[, *initargs*[, *maxtasksperchild*[, *context*]]]]])

작업을 제출할 수 있는 작업자 프로세스 풀을 제어하는 프로세스 풀 객체. 제한 시간과 콜백을 사용하는 비동기 결과를 지원하고 병렬 map 구현을 제공합니다.

processes is the number of worker processes to use. If *processes* is None then the number returned by *os.process_cpu_count()* is used.

initializer 가 None 이 아니면, 각 작업자 프로세스는 시작할 때 initializer(*initargs) 를 호출합니다.

maxtasksperchild 는, 사용되지 않는 자원을 해제할 수 있도록, 작업 프로세스가 종료되고 새 작업 프로세스로 교체되기 전에 완료할 수 있는 작업 수입니다. 기본 *maxtasksperchild* 는 `None` 입니다. 이는 작업자 프로세스가 풀만큼 오래감을 의미합니다.

context 는 작업자 프로세스를 시작하는 데 사용되는 컨텍스트를 지정하는 데 사용할 수 있습니다. 보통 풀은 `multiprocessing.Pool()` 또는 컨텍스트 객체의 `Pool()` 메서드를 사용하여 생성됩니다. 두 경우 모두 *context* 가 적절하게 설정됩니다.

풀 객체의 메서드는 풀을 생성한 프로세스에 의해서만 호출되어야 합니다.

경고

`multiprocessing.pool` 객체에는 풀을 컨텍스트 관리자로 사용하거나 `close()`와 `terminate()`를 수동으로 호출하여 (다른 자원과 마찬가지로) 올바르게 관리해야 하는 내부 자원이 있습니다. 이를 수행하지 않으면 파이널리제이션 때 프로세스가 멈출 수 있습니다.

Note that it is **not correct** to rely on the garbage collector to destroy the pool as CPython does not assure that the finalizer of the pool will be called (see `object.__del__()` for more information).

버전 3.2에서 변경: Added the *maxtasksperchild* parameter.

버전 3.4에서 변경: Added the *context* parameter.

버전 3.13에서 변경: *processes* uses `os.process_cpu_count()` by default, instead of `os.cpu_count()`.

참고

`Pool` 내의 작업자 프로세스는 일반적으로 Pool의 작업 큐의 전체 지속 기간 지속합니다. 작업자가 잡은 자원을 해제하기 위해 다른 시스템 (가령 Apache, mod_wsgi 등)에서 흔히 사용되는 패턴은, 풀 내에 있는 작업자가 종료되고 새 프로세스가 스폰되어 예전 것을 교체하기 전에 일정한 분량의 작업만 완료하도록 하는 것입니다. `Pool` 의 *maxtasksperchild* 인자는 이 기능을 일반 사용자에게 노출합니다.

apply(*func*[, *args*[, *kwds*]])

인자 *args* 및 키워드 인자 *kwds* 를 사용하여 *func* 를 호출합니다. 결과가 준비될 때까지 블록 됩니다. 이 블록 때문에, `apply_async()` 가 병렬로 작업을 수행하는 데 더 적합합니다. 또한 *func* 는 풀의 작업자 중 하나에서만 실행됩니다.

apply_async(*func*[, *args*[, *kwds*[, *callback*[, *error_callback*]]]])

`AsyncResult` 객체를 반환하는 `apply()` 메서드의 변형입니다.

callback 이 지정되면 단일 인자를 받아들이는 콜러블이어야 합니다. 결과가 준비되면 *callback* 을 이 결과를 인자로 호출합니다. 실패한 결과면 *error_callback* 이 대신 적용됩니다.

error_callback 이 지정되면 단일 인자를 허용하는 콜러블이어야 합니다. 대상 함수가 실패하면, *error_callback* 이 예외 인스턴스를 인자로 호출됩니다.

콜백은 즉시 완료되어야 합니다. 그렇지 않으면 결과를 처리하는 스레드가 블록 됩니다.

map(*func*, *iterable*[, *chunksize*])

`map()` 내장 함수의 병렬 버전입니다 (하지만 하나의 *iterable* 인자만 지원합니다, 여러 이터러블에 대해서는 `starmap()`을 참조하십시오). 결과가 준비될 때까지 블록 됩니다.

이 메서드는 iterable을 여러 묶음으로 잘라서 별도의 작업으로 프로세스 풀에 제출합니다. 이러한 묶음의 (대략적인) 크기는 *chunksize* 를 양의 정수로 설정하여 지정할 수 있습니다.

매우 긴 이터러블은 높은 메모리 사용을 유발할 수 있습니다. 더 나은 효율성을 위해, 명시적인 *chunksize* 옵션으로 `imap()` 이나 `imap_unordered()` 를 사용하는 것을 고려하십시오.

map_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

`AsyncResult` 객체를 반환하는 `map()` 메서드의 변형입니다.

callback 이 지정되면 단일 인자를 받아들이는 콜러블이어야 합니다. 결과가 준비되면 *callback* 을 이 결과를 인자로 호출합니다. 실패한 결과면 *error_callback* 이 대신 적용됩니다.

error_callback 이 지정되면 단일 인자를 허용하는 콜러블이어야 합니다. 대상 함수가 실패하면, *error_callback* 이 예외 인스턴스를 인자로 호출됩니다.

콜백은 즉시 완료되어야 합니다. 그렇지 않으면 결과를 처리하는 스레드가 블록 됩니다.

imap(*func*, *iterable*[, *chunksize*])

`map()` 의 느긋한 버전.

chunksize 인자는 `map()` 메서드에서 사용된 인자와 같습니다. 매우 긴 iterable의 경우 *chunksize* 에 큰 값을 사용하면 기본값 `1` 을 사용하는 것보다 작업을 **많이** 빠르게 완료 할 수 있습니다.

또한 *chunksize* 가 `1` 이면 `imap()` 메서드에 의해 반환된 이터레이터의 `next()` 메서드는 선택적 *timeout* 매개 변수를 가집니다: `next(timeout)` 은 결과가 *timeout* 초 내에 반환될 수 없는 경우 `multiprocessing.TimeoutError` 를 발생시킵니다.

imap_unordered(*func*, *iterable*[, *chunksize*])

`imap()`과 같지만, 반환된 이터레이터가 제공하는 결과의 순서가 임의적인 것으로 간주하여야 합니다. (단 하나의 작업자 프로세스가 있는 경우에만 순서가 "올바름" 이 보장됩니다.

starmap(*func*, *iterable*[, *chunksize*])

Like `map()` except that the elements of the *iterable* are expected to be iterables that are unpacked as arguments.

따라서 *iterable* 이 `[(1,2), (3, 4)]` 미면 결과는 `[func(1,2), func(3,4)]` 가 됩니다.

Added in version 3.3.

starmap_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])

`starmap()` 과 `map_async()` 의 조합으로 이터러블의 *iterable* 을 이터레이트하고 이터러블을 언팩해서 *func* 를 호출합니다. 결과 객체를 반환합니다.

Added in version 3.3.

close()

더는 작업이 풀에 제출되지 않도록 합니다. 모든 작업이 완료되면 작업자 프로세스가 종료됩니다.

terminate()

계류 중인 작업을 완료하지 않고 즉시 작업자 프로세스를 중지합니다. 풀 객체가 가비지 수집될 때 `terminate()` 가 즉시 호출됩니다.

join()

작업자 프로세스가 종료될 때까지 기다립니다. `join()` 호출 전에 반드시 `close()` 나 `terminate()`를 호출해야합니다.

버전 3.3에서 변경: 풀 객체는 이제 컨텍스트 관리 프로토콜을 지원합니다 – 컨텍스트 관리자 형를 보십시오. `__enter__()` 는 풀 객체를 반환하고, `__exit__()` 는 `terminate()`를 호출합니다.

class multiprocessing.pool.**AsyncResult**

`Pool.apply_async()`와 `Pool.map_async()` 에 의해 반환되는 결과의 클래스.

get([*timeout*])

결과가 도착할 때 반환합니다. *timeout* 이 `None` 이 아니고 결과가 *timeout* 초 내에 도착하지 않으면 `multiprocessing.TimeoutError` 가 발생합니다. 원격 호출이 예외를 발생시키는 경우 해당 예외는 `get()` 에 의해 다시 발생합니다.

wait([*timeout*])

결과가 사용 가능할 때까지 또는 *timeout* 초가 지날 때까지 기다립니다.

ready()

호출이 완료했는지를 돌려줍니다.

successful()

예외를 발생시키지 않고 호출이 완료되었는지를 돌려줍니다. 결과가 준비되지 않았으면 *ValueError* 를 발생시킵니다.

버전 3.7에서 변경: 결과가 준비되지 않았으면, *AssertionError* 대신 *ValueError*가 발생합니다.

다음 예제는 풀 사용 방법을 보여줍니다.:

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:          # start 4 worker processes
        result = pool.apply_async(f, (10,)) # evaluate "f(10)" asynchronously in a␣
↪single process
        print(result.get(timeout=1))         # prints "100" unless your computer is␣
↪*very* slow

        print(pool.map(f, range(10)))        # prints "[0, 1, 4,..., 81]"

        it = pool.imap(f, range(10))
        print(next(it))                      # prints "0"
        print(next(it))                      # prints "1"
        print(it.next(timeout=1))            # prints "4" unless your computer is␣
↪*very* slow

        result = pool.apply_async(time.sleep, (10,))
        print(result.get(timeout=1))         # raises multiprocessing.TimeoutError
```

리스너와 클라이언트

보통 프로세스 간 메시지 전달은 큐를 사용하거나 *Pipe()* 가 반환하는 *Connection* 객체를 사용하여 수행됩니다.

그러나, *multiprocessing.connection* 모듈은 약간의 추가적인 유연성을 허용합니다. 기본적으로 소켓이나 윈도우의 이름있는 파이프를 다루는 높은 수준의 메시지 지향 API를 제공합니다. 또한 *hmac* 모듈을 사용한 다이제스트 인증 과 다중 연결을 동시에 폴링하는 방법을 지원합니다.

multiprocessing.connection.**deliver_challenge**(*connection*, *authkey*)

무작위로 생성된 메시지를 연결의 다른 쪽 끝으로 보내고 응답을 기다립니다.

응답이 *authkey* 를 키로 사용하는 메시지의 다이제스트와 일치하면 환영 메시지가 연결의 다른 끝으로 전송됩니다. 그렇지 않으면 *AuthenticationError* 가 발생합니다.

multiprocessing.connection.**answer_challenge**(*connection*, *authkey*)

메시지를 수신하고, *authkey* 를 키로 사용하여 메시지의 다이제스트를 계산한 다음, 다이제스트를 다시 보냅니다.

환영 메시지가 수신되지 않으면, *AuthenticationError* 가 발생합니다.

multiprocessing.connection.**Client**(*address*[, *family*[, *authkey*]])

주소 *address* 를 사용하는 리스너에 대한 연결을 설정하려고 시도하고, *Connection*을 반환합니다.

연결 유형은 *family* 인자에 의해 결정되지만, 일반적으로 *address* 형식에서 유추 할 수 있으므로 일반적으로 생략 할 수 있습니다. (주소 형식를 참조하세요)

If *authkey* is given and not None, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is None. AuthenticationError is raised if authentication fails. See 인증 키.

class multiprocessing.connection.**Listener**([*address*[, *family*[, *backlog*[, *authkey*]]]])

연결을 '리스닝' 하는 바인드된 소켓이나 윈도우의 이름있는 파이프에 대한 래퍼입니다.

address 는 리스너 객체의 바인드된 소켓이나 이름있는 파이프가 사용할 주소입니다.

> **참고**
>
> 주소가 '0.0.0.0' 인 경우, 주소는 윈도우에서 연결 가능한 끝점이 아닙니다. 연결할 수 있는 끝점이 필요한 경우, '127.0.0.1' 을 사용해야 합니다.

family 는 사용할 소켓(또는 이름있는 파이프)의 유형입니다. 문자열 'AF_INET' (TCP 소켓), 'AF_UNIX' (유닉스 도메인 소켓), 'AF_PIPE' (윈도우 이름있는 파이프) 중 하나일 수 있습니다. 이 중 오직 첫 번째 것만 항상 사용할 수 있음이 보장됩니다. *family* 가 None 이면, *address* 의 형식으로부터 유추됩니다. *address* 역시 None 이면, 기본값이 선택됩니다. 이 기본값은 사용 가능한 것 중 가장 빠른 것으로 기대되는 것입니다. 주소 형식를 참조하세요. *family* 가 'AF_UNIX' 이고 주소가 None 이면, 소켓은 tempfile.mkstemp() 를 사용하여 만들어진 비공개 임시 디렉터리에 생성됩니다.

리스너 객체가 소켓을 사용하면, *backlog* (기본적으로 1) 는 소켓이 바인드되면 소켓의 listen() 메서드에 전달됩니다.

If *authkey* is given and not None, it should be a byte string and will be used as the secret key for an HMAC-based authentication challenge. No authentication is done if *authkey* is None. AuthenticationError is raised if authentication fails. See 인증 키.

accept()

리스너 객체의 바인드된 소켓 또는 이름있는 파이프에 대한 연결을 수락하고 Connection 객체를 반환합니다. 인증이 시도되고 실패하면 AuthenticationError 가 발생합니다.

close()

리스너 객체의 바운드된 소켓 또는 이름있는 파이프를 닫습니다. 리스너가 가비지 수집될 때 자동으로 호출됩니다. 그러나 명시적으로 호출하는 것이 좋습니다.

리스너 객체는 다음과 같은 읽기 전용 프로퍼티를 가집니다:

address

리스너 객체에서 사용 중인 주소.

last_accepted

마지막으로 수락한 연결이 온 주소. 없으면 None 입니다.

버전 3.3에서 변경: 리스너 객체는 컨텍스트 관리 프로토콜을 지원합니다 – 컨텍스트 관리자 형를 보세요. __enter__() 는 리스너 객체를 반환하고, __exit__() 는 close()를 호출합니다.

multiprocessing.connection.**wait**(*object_list*, *timeout=None*)

object_list 에 있는 객체가 준비될 때까지 기다립니다. *object_list* 에 있는 객체 중 준비된 것들의 리스트를 반환합니다. *timeout* 이 float 면, 호출이 최대 지정된 초만큼 블록 됩니다. *timeout* 이 None 이면, 시간제한 없이 블록 됩니다. 음수 timeout은 0과 같습니다.

For both POSIX and Windows, an object can appear in *object_list* if it is

- 읽기 가능한 Connection 객체;
- 연결되고 읽기 가능한 socket.socket 객체; 또는
- Process 객체의 sentinel 어트리뷰트.

연결이나 소켓 객체는 읽을 수 있는 데이터가 있거나 반대편 끝이 닫히면 준비가 됩니다.

POSIX: `wait(object_list, timeout)` almost equivalent `select.select(object_list, [], [], timeout)`. The difference is that, if *select.select()* is interrupted by a signal, it can raise *OSError* with an error number of `EINTR`, whereas *wait()* will not.

Windows: An item in *object_list* must either be an integer handle which is waitable (according to the definition used by the documentation of the Win32 function `WaitForMultipleObjects()`) or it can be an object with a *fileno()* method which returns a socket handle or pipe handle. (Note that pipe handles and socket handles are **not** waitable handles.)

Added in version 3.3.

예제

다음 서버 코드는 인증 키로 `'secret password'` 를 사용하는 리스너를 만듭니다. 그런 다음 연결을 기다리고 어떤 데이터를 클라이언트로 보냅니다.:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)     # family is deduced to be 'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)

        conn.send([2.25, None, 'junk', float])

        conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

다음 코드는 서버에 연결하고 서버로부터 어떤 데이터를 받습니다:

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())                  # => [2.25, None, 'junk', float]

    print(conn.recv_bytes())            # => 'hello'

    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr))    # => 8
    print(arr)                          # => array('i', [42, 1729, 0, 0, 0])
```

다음 코드는 *wait()* 을 사용하여 여러 프로세스로부터 오는 메시지를 한 번에 기다립니다:

```
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure that
        # p is the only process which owns a handle for it.  This
        # ensures that when p closes its handle for the writable end,
        # wait() will promptly report the readable end as being ready.
        w.close()

    while readers:
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)
```

주소 형식

- `'AF_INET'` 주소는 `(hostname, port)` 형식의 튜플입니다. *hostname* 은 문자열이고, *port* 는 정수입니다.
- `'AF_UNIX'` 주소는 파일 시스템의 파일 이름을 나타내는 문자열입니다.
- An `'AF_PIPE'` address is a string of the form `r'\\.\pipe\PipeName'`. To use `Client()` to connect to a named pipe on a remote computer called *ServerName* one should use an address of the form `r'\\ServerName\pipe\PipeName'` instead.

두 개의 역 슬래시로 시작하는 문자열은 기본적으로 `'AF_UNIX'` 주소가 아니라 `'AF_PIPE'` 주소로 간주합니다.

인증 키

`Connection.recv` 를 사용할 때, 수신된 데이터는 자동으로 언 피클 됩니다. 안타깝게도, 신뢰할 수 없는 출처의 데이터를 언 피클 하는 것은 보안상의 위험입니다. 때문에 `Listener`와 `Client()` 는 `hmac` 모듈을 사용하여 다이제스트 인증을 제공합니다.

인증 키는 암호로 여겨질 수 있는 바이트열입니다: 일단 연결이 이루어지면 양 끝은 다른 쪽이 인증 키를 알고 있음을 증명하도록 요구합니다. (양쪽 끝이 같은 키를 사용하고 있음을 증명하는 데는 연결을 통해 키를 보내는 것을 수반하지 **않습니다**.)

인증이 요청되었지만 인증 키가 지정되지 않으면, `current_process().authkey` 의 반환 값이 사용됩니다 (`Process` 를 보세요). 이 값은 현재 프로세스가 생성하는 `Process` 객체에 의해 자동으로 상속됩니다. 이것은 다중 프로세스 프로그램의 모든 프로세스는 (기본적으로) 자신들 간의 연결을 설정할 때 사용할 수 있는 하나의 인증 키를 공유한다는 것을 뜻합니다.

적절한 인증 키는 `os.urandom()` 을 사용하여 생성할 수도 있습니다.

로깅

로깅에 대한 일부 지원이 제공됩니다. 그러나, `logging` 패키지는 프로세스 공유 록을 사용하지 않으므로 (처리기형에 따라) 다른 프로세스의 메시지가 뒤섞일 가능성이 있습니다.

`multiprocessing.get_logger()`

`multiprocessing`에서 사용되는 로거를 반환합니다. 필요하다면, 새로운 것이 만들어집니다.

When first created the logger has level `logging.NOTSET` and no default handler. Messages sent to this logger will not by default propagate to the root logger.

윈도우에서 자식 프로세스는 부모 프로세스의 로거의 수준만 상속받습니다 – 그 밖의 다른 로거 사용자 지정은 상속되지 않습니다.

multiprocessing.**log_to_stderr**(*level=None*)

This function performs a call to `get_logger()` but in addition to returning the logger created by get_logger, it adds a handler which sends output to `sys.stderr` using format `'[%(levelname)s/%(processName)s] %(message)s'`. You can modify `levelname` of the logger by passing a `level` argument.

다음은 로깅이 켜져 있는 예제 세션입니다:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

로깅 수준의 전체 표는 `logging` 모듈을 참조하십시오.

multiprocessing.dummy 모듈

`multiprocessing.dummy` 는 `multiprocessing` 의 API를 복제하지만 `threading` 모듈에 대한 래퍼일 뿐입니다.

특히, `multiprocessing.dummy`에서 제공하는 `Pool` 함수는 같은 메서드 호출을 모두 지원하지만, 작업자 프로세스가 아닌 작업자 스레드 풀을 사용하는 `Pool`의 서브 클래스인 `ThreadPool`의 인스턴스를 반환합니다.

class multiprocessing.pool.**ThreadPool**([*processes*[, *initializer*[, *initargs*]]])

작업을 제출할 수 있는 작업자 스레드 풀을 제어하는 스레드 풀 객체. `ThreadPool` 인스턴스는 `Pool` 인스턴스와 완전히 호환되며, 해당 리소스는 컨텍스트 관리자로 풀을 사용하거나 `close()`와 `terminate()`를 수동으로 호출하여 적절하게 관리해야 합니다.

processes is the number of worker threads to use. If *processes* is `None` then the number returned by `os.process_cpu_count()` is used.

initializer 가 `None` 이 아니면, 각 작업자 프로세스는 시작할 때 `initializer(*initargs)` 를 호출합니다.

`Pool`과 달리, *maxtasksperchild*와 *context*는 제공할 수 없습니다.

> **참고**
>
> `ThreadPool`은 프로세스 풀을 중심으로 설계되고 `concurrent.futures` 모듈 도입 이전에 설계된 `Pool`과 같은 인터페이스를 공유합니다. 따라서, 스레드가 지원하는 풀에 적합하지 않은 일부 연산을 상속하고, 비동기 작업의 상태를 나타내는 자체 형 `AsyncResult`를 가지고 있는데 다른 라이브러리에서는 이해하지 못합니다.
>
> 사용자는 일반적으로 처음부터 스레드를 중심으로 설계되고 `asyncio`를 포함한 다른 많은 라이브러리와 호환되는 `concurrent.futures.Future` 인스턴스를 반환하는 더 간단한 인터페이스를 가진 `concurrent.futures.ThreadPoolExecutor`를 사용하는 것을 선호해야 합니다.

18.2.3 프로그래밍 지침

*multiprocessing*를 사용할 때 준수해야 할 지침과 관용구가 있습니다.

모든 시작 방법

다음은 모든 시작 방법에 적용됩니다.

공유 상태를 피하세요

가능한 한 프로세스 간에 많은 양의 데이터가 이동하지 않도록 해야 합니다.

저수준 동기화 프리미티브를 사용하기보다, 프로세스 간 통신을 위해 큐나 파이프를 사용하는 것이 아마도 최선입니다.

피클 가능성

프락시 메서드에 대한 인자가 피클 가능한지 확인하십시오.

프락시의 스레드 안전성

록으로 보호하지 않는 한 둘 이상의 스레드에서 프락시 객체를 사용하지 마십시오.

(여러 프로세스가 같은 프락시를 사용하는 문제는 존재하지 않습니다.)

좀비 프로세스 조인하기

On POSIX when a process finishes but has not been joined it becomes a zombie. There should never be very many because each time a new process starts (or *active_children()* is called) all completed processes which have not yet been joined will be joined. Also calling a finished process' s *Process.is_alive* will join the process. Even so it is probably good practice to explicitly join all the processes that you start.

피클/언 피클보다 상속하는 것이 더 좋습니다.

spawn 이나 *forkserver* 시작 방법을 사용할 때, *multiprocessing* 의 여러 형은 자식 프로세스가 사용할 수 있도록 피클 가능할 필요가 있습니다. 그러나, 일반적으로 파이프나 큐를 사용하여 공유 객체를 다른 프로세스로 보내는 것을 피해야 합니다. 대신 다른 곳에 만들어진 공유 자원에 접근해야 하는 프로세스가 조상 프로세스에서 그것들을 상속받을 수 있도록 프로그램을 배치해야 합니다.

프로세스 강제 종료를 피하세요

Process.terminate 메서드를 사용해서 프로세스를 정지시키는 것은, 그 프로세스가 현재 사용하고 있는 공유 자원(가령 록, 세마포어, 파이프, 큐)을 손상하거나 다른 프로세스에서 사용할 수 없게 만들 수 있습니다.

따라서, 아마도 어떤 공유 자원도 사용하지 않는 프로세스에만 *Process.terminate* 사용을 고려하는 것이 최선일 겁니다.

큐를 사용하는 프로세스 조인하기

큐에 항목을 넣은 프로세스는 종료되기 전에 버퍼링 된 모든 항목이 "피더" 스레드에 의해 하부 파이프로 공급될 때까지 대기합니다. (자식 프로세스는 *Queue.cancel_join_thread* 메서드를 호출해서 이 동작을 회피할 수 있습니다.)

이것은, 큐를 사용할 때마다 큐에 넣은 모든 항목이 결국 프로세스가 조인되기 전에 제거되도록 해야 함을 의미합니다. 그렇지 않으면 큐에 항목을 넣은 프로세스가 종료되리라고 보장할 수 없습니다. 데몬이 아닌 프로세스가 자동으로 조인된다는 것도 기억하세요.

교착 상태에 빠지는 예는 다음과 같습니다:

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                    # this deadlocks
    obj = queue.get()
```

이 문제를 고치는 방법은 마지막 두 줄의 순서를 바꾸는 것입니다 (또는 간단히 `p.join()` 줄을 지우는 것입니다).

자식 프로세스에 자원을 명시적으로 전달하세요.

On POSIX using the *fork* start method, a child process can make use of a shared resource created in a parent process using a global resource. However, it is better to pass the object as an argument to the constructor for the child process.

윈도우 및 다른 시작 방법과 (잠재적으로) 호환될 수 있는 코드를 만드는 것 외에도, 이것은 자식 프로세스가 아직 살아있는 동안 객체가 부모 프로세스에서 가비지 수집되지 않음을 보장합니다. 부모 프로세스에서 그 객체가 가비지 수집될 때 일부 자원이 해제되면 이것이 중요 할 수 있습니다.

그래서 예를 들면

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

는 다음과 같이 다시 써야 합니다

```
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

sys.stdin 을 "파일류 객체" 로 교체할 때 조심하세요

*multiprocessing*은 원래 무조건 다음과 같이 호출했습니다

```
os.close(sys.stdin.fileno())
```

`multiprocessing.Process._bootstrap()` 메서드에서 하는 작업입니다 — 이것은 손자 프로세스와 관련된 문제로 이어졌습니다. 이것은 다음과 같이 변경되었습니다:

```
sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY), closefd=False)
```

Which solves the fundamental issue of processes colliding with each other resulting in a bad file descriptor error, but introduces a potential danger to applications which replace *sys.stdin()* with a "file-like object" with output buffering. This danger is that if multiple processes call *close()* on this file-like object, it could result in the same data being flushed to the object multiple times, resulting in corruption.

파일류 객체를 작성하고 여러분 자신의 캐싱을 구현하면, 캐시에 추가할 때마다 pid를 저장하고, pid가 변경되면 캐시를 버려서 포크에 안전하게 만들 수 있습니다. 예를 들면:

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

자세한 내용은 bpo-5155, bpo-5313 및 bpo-5331을 참조하십시오.

spawn 과 *forkserver* 시작 방법

There are a few extra restrictions which don' t apply to the *fork* start method.

더 높은 피클 가능성

Ensure that all arguments to `Process` are picklable. Also, if you subclass `Process.__init__`, you must make sure that instances will be picklable when the `Process.start` method is called.

전역 변수

자식 프로세스에서 실행되는 코드가 전역 변수에 접근하려고 시도하면, 그 값은 (있는 경우) `Process.start` 가 호출되는 시점의 부모 프로세스의 값과 같지 않을 수 있습니다.

하지만, 모듈 수준의 상수인 전역 변수는 문제가 되지 않습니다.

메인 모듈의 안전한 임포트

Make sure that the main module can be safely imported by a new Python interpreter without causing unintended side effects (such as starting a new process).

예를 들어, *spawn* 또는 *forkserver* 시작 방법을 사용해서 다음 모듈을 실행하면 `RuntimeError` 로 실패합니다:

```
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

대신 다음과 같이 `if __name__ == '__main__':` 을 사용하여 프로그램의 "진입 지점"을 보호해야 합니다:

```
from multiprocessing import Process, freeze_support, set_start_method

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    set_start_method('spawn')
    p = Process(target=foo)
    p.start()
```

(`freeze_support()` 줄은 프로그램이 프로즌 되지 않고 정상적으로 실행될 경우 생략될 수 있습니다.)

이것은 새로 스폰 된 파이썬 인터프리터가 모듈을 안전하게 임포트 한 다음 모듈의 `foo()` 함수를 실행할 수 있게 해줍니다.

메인 모듈에서 풀이나 관리자를 만들면 비슷한 제한이 적용됩니다.

18.2.4 예제

사용자 정의된 관리자와 프락시를 만들고 사용하는 방법에 대한 시연:

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

# A simple generator function
def baz():
    for i in range(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)

##

def test():
    manager = MyManager()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print('-' * 20)

    it = manager.baz()
    for i in it:
        print('<%d>' % i, end=' ')
    print()

    print('-' * 20)

    op = manager.operator()
    print('op.add(23, 45) =', op.add(23, 45))
    print('op.pow(2, 94) =', op.pow(2, 94))
    print('op._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
    test()
```

Pool 사용하기:

```
import multiprocessing
import time
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
        )
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# Test code
#

def test():
    PROCESSES = 4
    print('Creating pool with %d processes\n' % PROCESSES)

    with multiprocessing.Pool(PROCESSES) as pool:
        #
        # Tests
        #

        TASKS = [(mul, (i, 7)) for i in range(10)] + \
                [(plus, (i, 8)) for i in range(10)]

        results = [pool.apply_async(calculate, t) for t in TASKS]
        imap_it = pool.imap(calculatestar, TASKS)
        imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

        print('Ordered results using pool.apply_async():')
        for r in results:
            print('\t', r.get())
        print()

        print('Ordered results using pool.imap():')
        for x in imap_it:
            print('\t', x)
        print()

        print('Unordered results using pool.imap_unordered():')
        for x in imap_unordered_it:
            print('\t', x)
        print()

        print('Ordered results using pool.map() --- will block till complete:')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        for x in pool.map(calculatestar, TASKS):
            print('\t', x)
        print()

        #
        # Test error handling
        #

        print('Testing error handling:')

        try:
            print(pool.apply(f, (5,)))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.apply()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(pool.map(f, list(range(10))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.map()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(list(pool.imap(f, list(range(10)))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from list(pool.imap())')
        else:
            raise AssertionError('expected ZeroDivisionError')

        it = pool.imap(f, list(range(10)))
        for i in range(10):
            try:
                x = next(it)
            except ZeroDivisionError:
                if i == 5:
                    pass
            except StopIteration:
                break
            else:
                if i == 5:
                    raise AssertionError('expected ZeroDivisionError')

        assert i == 9
        print('\tGot ZeroDivisionError as expected from IMapIterator.next()')
        print()

        #
        # Testing timeouts
        #

        print('Testing ApplyResult.get() with timeout:', end=' ')
        res = pool.apply_async(calculate, TASKS[0])
        while 1:
            sys.stdout.flush()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
            try:
                sys.stdout.write('\n\t%s' % res.get(0.02))
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()

        print('Testing IMapIterator.next() with timeout:', end=' ')
        it = pool.imap(calculatestar, TASKS)
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % it.next(0.02))
            except StopIteration:
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()


if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
```

큐를 사용하여 작업을 작업자 프로세스 집단에 제공하고 결과를 수집하는 방법을 보여주는 예:

```
import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)

#
# Functions referenced by tasks
#

def mul(a, b):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print('Unordered results:')
    for i in range(len(TASKS1)):
        print('\t', done_queue.get())

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print('\t', done_queue.get())

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')


if __name__ == '__main__':
    freeze_support()
    test()
```

18.3 multiprocessing.shared_memory — 프로세스 간 직접 액세스를 위한 공유 메모리

소스 코드: Lib/multiprocessing/shared_memory.py

Added in version 3.8.

이 모듈은 멀티 코어나 대칭 멀티 프로세서 (SMP) 기계에서 하나 이상의 프로세스가 액세스할 공유 메모리의 할당과 관리를 위한 클래스 *SharedMemory*를 제공합니다. 특히 별개의 프로세스에 걸친 공유 메모리의 수명 주기 관리를 지원하기 위해, *BaseManager* 서브 클래스인 *SharedMemoryManager*도 *multiprocessing.managers* 모듈에서 제공됩니다.

이 모듈에서, 공유 메모리는 "POSIX 스타일" 공유 메모리 블록을 가리키며 (꼭 그런 식으로 구현돼야 할 필요는 없습니다), "분산 공유 메모리"를 가리키지는 않습니다. 이 스타일의 공유 메모리는 개별 프로세스가 잠재적으로 휘발성 메모리의 공통 (또는 공유) 영역을 읽고 쓸 수 있게 합니다. 프로세스는 일반적으로 자체 프로세스 메모리 공간에만 액세스 할 수 있도록 제한되지만, 공유 메모리는 프로세스 간에 데이터를 공유 할 수 있도록 해서, 프로세스 간에 대신 해당 데이터가 포함된 메시지를 보낼 필요가 없도록 합니다. 메모리를 통해 직접 데이터를 공유하면 디스크나 소켓 또는 직렬화/역 직렬화와 데이터의 복사를 요구하는 다른 통신과 비교하여 상당한 성능상의 이점을 얻을 수 있습니다.

class multiprocessing.shared_memory.**SharedMemory**(*name=None, create=False, size=0, *, track=True*)

새 공유 메모리 블록을 만들거나 기존 공유 메모리 블록에 연결하기 위해 SharedMemory 클래스의 인스턴스를 만듭니다. 각 공유 메모리 블록에는 고유한 이름이 지정됩니다. 이런 식으로, 하나의 프로세스가 특정 이름을 가진 공유 메모리 블록을 생성 할 수 있으며, 다른 프로세스가 같은 이름을 사용하여 같은 공유 메모리 블록에 연결할 수 있습니다.

프로세스 간에 데이터를 공유하기 위한 자원으로서, 공유 메모리 블록은 생성한 원래 프로세스보다 오래갈 수 있습니다. 한 프로세스가 더는 다른 프로세스가 필요로 할 수도 있는 공유 메모리 블록에 대한 액세스를 필요로하지 않으면 *close()* 메서드를 호출해야 합니다. 어떤 프로세스에서도 공유 메모리 블록이 더는 필요하지 않으면, 적절한 정리를 위해 *unlink()* 메서드를 호출해야 합니다.

매개변수

- **name** (*str | None*) – 문자열로 지정된, 요청된 공유 메모리의 고유한 이름. 새 공유 메모리 블록을 만들 때, 이름에 None(기본값)이 제공되면, 새로운 이름이 생성됩니다.
- **create** (bool) – 새 공유 메모리 블록을 만들지(True), 또는 기존 공유 메모리 블록을 연결할지(False)를 제어합니다.
- **size** (int) – 새 공유 메모리 블록을 만들 때 요청된 바이트 수. 일부 플랫폼은 해당 플랫폼의 메모리 페이지 크기를 기반으로 메모리 덩어리를 할당하기 때문에, 공유 메모리 블록의 정확한 크기는 요청한 크기보다 크거나 같을 수 있습니다. 기존 공유 메모리 블록에 연결할 때는, *size* 매개 변수가 무시됩니다.
- **track** (bool) – When True, register the shared memory block with a resource tracker process on platforms where the OS does not do this automatically. The resource tracker ensures proper cleanup of the shared memory even if all other processes with access to the memory exit without doing so. Python processes created from a common ancestor using *multiprocessing* facilities share a single resource tracker process, and the lifetime of shared memory segments is handled automatically among these processes. Python processes created in any other way will receive their own resource tracker when accessing shared memory with *track* enabled. This will cause the shared memory to be deleted by the resource tracker of the first process that terminates. To avoid this issue, users of *subprocess* or standalone Python processes should set *track* to False when there is already another process in place that does the bookkeeping. *track* is ignored on Windows, which has its own tracking and automatically deletes shared memory when all handles to it have been closed.

버전 3.13에서 변경: Added the *track* parameter.

close()

Close the file descriptor/handle to the shared memory from this instance. *close()* should be called once access to the shared memory block from this instance is no longer needed. Depending on operating system, the underlying memory may or may not be freed even if all handles to it have been closed. To ensure proper cleanup, use the *unlink()* method.

unlink()

Delete the underlying shared memory block. This should be called only once per shared memory block regardless of the number of handles to it, even in other processes. *unlink()* and *close()* can be called in any order, but trying to access data inside a shared memory block after *unlink()* may result in memory access errors, depending on platform.

This method has no effect on Windows, where the only way to delete a shared memory block is to close all handles.

buf

공유 메모리 블록의 내용에 대한 메모리 뷰.

name

공유 메모리 블록의 고유한 이름에 대한 읽기 전용 액세스.

size

공유 메모리 블록의 크기(바이트)에 대한 읽기 전용 액세스.

다음 예제는 *SharedMemory* 인스턴스의 저수준 사용을 보여줍니다:

```
>>> from multiprocessing import shared_memory
>>> shm_a = shared_memory.SharedMemory(create=True, size=10)
>>> type(shm_a.buf)
<class 'memoryview'>
>>> buffer = shm_a.buf
>>> len(buffer)
10
>>> buffer[:4] = bytearray([22, 33, 44, 55])  # 여러 바이트를 한 번에 수정합니다
>>> buffer[4] = 100                           # 한 번에 한 바이트씩 수정합니다
>>> # 기존 공유 메모리 블록에 연결합니다
>>> shm_b = shared_memory.SharedMemory(shm_a.name)
>>> import array
>>> array.array('b', shm_b.buf[:5])  # 새 array.array로 데이터를 복사합니다
array('b', [22, 33, 44, 55, 100])
>>> shm_b.buf[:5] = b'howdy'  # shm_b를 통해 bytes를 사용해서 수정합니다
>>> bytes(shm_a.buf[:5])      # shm_a를 통해 액세스합니다
b'howdy'
>>> shm_b.close()   # 각 SharedMemory 인스턴스를 닫습니다
>>> shm_a.close()
>>> shm_a.unlink()  # 공유 메모리를 해제하기 위해 오직 한 번만 unlink를 호출합니다
```

다음 예제는 두 개의 다른 파이썬 셸에서 같은 `numpy.ndarray`에 액세스하는, NumPy 배열과 함께 *SharedMemory* 클래스를 사용하는 실용적인 방법을 보여줍니다:

```
>>> # 첫 번째 파이썬 대화형 셸에서
>>> import numpy as np
>>> a = np.array([1, 1, 2, 3, 5, 8])  # 기존 NumPy 배열로 시작합니다
>>> from multiprocessing import shared_memory
>>> shm = shared_memory.SharedMemory(create=True, size=a.nbytes)
>>> # 이제 공유 메모리에 NumPy 배열을 만듭니다
>>> b = np.ndarray(a.shape, dtype=a.dtype, buffer=shm.buf)
>>> b[:] = a[:]  # 원래 데이터를 공유 메모리로 복사합니다
>>> b
array([1, 1, 2, 3, 5, 8])
>>> type(b)
<class 'numpy.ndarray'>
>>> type(a)
<class 'numpy.ndarray'>
>>> shm.name  # 이름을 지정하지 않았으므로, 우리를 위해 이름이 선택되었습니다
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
'psm_21467_46075'

>>> # 같은 셸이나 같은 기계 상의 새로운 파이썬 셸에서
>>> import numpy as np
>>> from multiprocessing import shared_memory
>>> # 기존 공유 메모리 블록에 연결합니다
>>> existing_shm = shared_memory.SharedMemory(name='psm_21467_46075')
>>> # 이 예제에서 a.shape은 (6,)이고 a.dtype은 np.int64 입니다
>>> c = np.ndarray((6,), dtype=np.int64, buffer=existing_shm.buf)
>>> c
array([1, 1, 2, 3, 5, 8])
>>> c[-1] = 888
>>> c
array([  1,   1,   2,   3,   5, 888])

>>> # 첫 번째 파이썬 대화형 셸로 돌아가면, b가 이 변경을 반영합니다
>>> b
array([  1,   1,   2,   3,   5, 888])

>>> # 두 번째 파이썬 셸 내에서 정리합니다
>>> del c  # 필요 없습니다; 단순히 배열이 더는 사용되지 않는다는 것을 강조하기 위함입니다
>>> existing_shm.close()

>>> # 첫 번째 파이썬 셸 내에서 정리합니다
>>> del b  # 필요 없습니다; 단순히 배열이 더는 사용되지 않는다는 것을 강조하기 위함입니다
>>> shm.close()
>>> shm.unlink()  # 제일 끝에서 공유 메모리 블록을 해제하고 반납합니다
```

class multiprocessing.managers.**SharedMemoryManager**([*address*[, *authkey*]])

프로세스 간 공유 메모리 블록을 관리하는 데 사용할 수 있는 *multiprocessing.managers.BaseManager*의 서브 클래스.

SharedMemoryManager 인스턴스에서 *start()*를 호출하면 새 프로세스가 시작됩니다. 이 새로운 프로세스의 유일한 목적은 이를 통해 생성된 모든 공유 메모리 블록의 수명 주기를 관리하는 것입니다. 해당 프로세스가 관리하는 모든 공유 메모리 블록의 해제를 시작시키려면, 해당 인스턴스에서 *shutdown()*을 호출하십시오. 그러면 이 프로세스에 의해 관리되는 모든 *SharedMemory* 객체에 대해 *unlink()* 호출을 일으키고, 그런 다음 프로세스 자체를 중지합니다. SharedMemoryManager를 통해 SharedMemory 인스턴스를 생성함으로써, 공유 메모리 자원을 수동으로 추적하여 해제할 필요가 없습니다.

이 클래스는 *SharedMemory* 인스턴스를 만들고 반환하는 메서드와, 공유 메모리로 지원되는 리스트류 객체(*ShareableList*)를 만드는 메서드를 제공합니다.

상속된 *address*와 *authkey* 선택적 입력 인자에 대한 설명과 이 인자를 사용하여 다른 프로세스의 기존 SharedMemoryManager 서비스에 연결하는 방법에 대해서는 *BaseManager*를 참조하십시오.

SharedMemory(*size*)

바이트로 지정된 *size* 크기의 새로운 *SharedMemory* 객체를 만들고 반환합니다.

ShareableList(*sequence*)

입력 *sequence*의 값으로 초기화된, 새 *ShareableList* 객체를 만들고 반환합니다.

다음 예제는 *SharedMemoryManager*의 기본 메커니즘을 보여줍니다:

```
>>> from multiprocessing.managers import SharedMemoryManager
>>> smm = SharedMemoryManager()
>>> smm.start()  # 공유 메모리 블록을 관리하는 프로세스를 시작합니다
>>> sl = smm.ShareableList(range(4))
>>> sl
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
ShareableList([0, 1, 2, 3], name='psm_6572_7512')
>>> raw_shm = smm.SharedMemory(size=128)
>>> another_sl = smm.ShareableList('alpha')
>>> another_sl
ShareableList(['a', 'l', 'p', 'h', 'a'], name='psm_6572_12221')
>>> smm.shutdown()  # sl, raw_shm, 및 another_sl 에 대해 unlink()를 호출합니다
```

다음 예제는 `with` 문을 통해 *SharedMemoryManager* 객체를 사용하여 더는 필요하지 않은 모든 공유 메모리 블록이 해제되도록 하는, 잠재적으로 더 편리한 패턴을 보여줍니다:

```
>>> with SharedMemoryManager() as smm:
...     sl = smm.ShareableList(range(2000))
...     # 작업을 두 프로세스로 나누고, 부분적인 결과를 sl에 저장합니다
...     p1 = Process(target=do_work, args=(sl, 0, 1000))
...     p2 = Process(target=do_work, args=(sl, 1000, 2000))
...     p1.start()
...     p2.start()  # multiprocessing.Pool이 더 효율적일 수 있습니다
...     p1.join()
...     p2.join()   # 두 프로세스가 모두 작업을 끝낼 때까지 기다립니다
...     total_result = sum(sl)  # 이제 sl에 있는 부분 결과들을 통합합니다
```

`with` 문에서 *SharedMemoryManager*를 사용할 때, `with` 문의 코드 블록 실행이 완료되면 해당 관리자를 사용하여 만들어진 공유 메모리 블록이 모두 해제됩니다.

class `multiprocessing.shared_memory.`**`ShareableList`**(*sequence=None*, **, name=None*)

Provide a mutable list-like object where all values stored within are stored in a shared memory block. This constrains storable values to the following built-in data types:

- *int* (부호있는 64-비트)
- *float*
- *bool*
- *str* (less than 10M bytes each when encoded as UTF-8)
- *bytes* (less than 10M bytes each)
- `None`

It also notably differs from the built-in *list* type in that these lists can not change their overall length (i.e. no `append()`, `insert()`, etc.) and do not support the dynamic creation of new `ShareableList` instances via slicing.

*sequence*는 새로운 `ShareableList`를 값으로 가득 채우는 데 사용됩니다. 고유한 공유 메모리 이름으로 이미 존재하는 `ShareableList`에 대신 연결하려면 `None`으로 설정하십시오.

*name*은 *SharedMemory*에 대한 정의에서 설명한 대로, 요청된 공유 메모리의 고유한 이름입니다. 기존 `ShareableList`에 연결할 때, *sequence*를 `None`으로 설정하고 공유 메모리 블록의 고유한 이름을 지정하십시오.

> **참고**
>
> A known issue exists for *bytes* and *str* values. If they end with `\x00` nul bytes or characters, those may be *silently stripped* when fetching them by index from the `ShareableList`. This `.rstrip(b'\x00')` behavior is considered a bug and may go away in the future. See gh-106939.

For applications where rstripping of trailing nulls is a problem, work around it by always unconditionally appending an extra non-0 byte to the end of such values when storing and unconditionally removing it when fetching:

```
>>> from multiprocessing import shared_memory
>>> nul_bug_demo = shared_memory.ShareableList(['?\x00', b'\x03\x02\x01\x00\
↪x00\x00'])
>>> nul_bug_demo[0]
'?'
>>> nul_bug_demo[1]
b'\x03\x02\x01'
>>> nul_bug_demo.shm.unlink()
>>> padded = shared_memory.ShareableList(['?\x00\x07', b'\x03\x02\x01\x00\x00\
↪x00\x07'])
>>> padded[0][:-1]
'?\x00'
>>> padded[1][:-1]
b'\x03\x02\x01\x00\x00\x00'
>>> padded.shm.unlink()
```

count(*value*)

*value*의 발생 횟수를 반환합니다.

index(*value*)

*value*의 첫 번째 인덱스 위치를 반환합니다. *value*가 없으면 `ValueError`를 발생시킵니다.

format

현재 저장된 모든 값이 사용하는 `struct` 패킹 형식을 포함하는 읽기 전용 어트리뷰트.

shm

값이 저장되는 `SharedMemory` 인스턴스.

다음 예제는 `ShareableList` 인스턴스의 기본 사용을 보여줍니다.:

```
>>> from multiprocessing import shared_memory
>>> a = shared_memory.ShareableList(['howdy', b'HoWdY', -273.154, 100, None, True,␣
↪42])
>>> [ type(entry) for entry in a ]
[<class 'str'>, <class 'bytes'>, <class 'float'>, <class 'int'>, <class 'NoneType'>
↪, <class 'bool'>, <class 'int'>]
>>> a[2]
-273.154
>>> a[2] = -78.5
>>> a[2]
-78.5
>>> a[2] = 'dry ice'  # Changing data types is supported as well
>>> a[2]
'dry ice'
>>> a[2] = 'larger than previously allocated storage space'
Traceback (most recent call last):
  ...
ValueError: exceeds available storage for existing str
>>> a[2]
'dry ice'
>>> len(a)
7
>>> a.index(42)
6
>>> a.count(b'howdy')
0
>>> a.count(b'HoWdY')
1
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> a.shm.close()
>>> a.shm.unlink()
>>> del a  # Use of a ShareableList after call to unlink() is unsupported
```

다음 예는 하나, 둘 또는 여러 프로세스가 그 뒤에 있는 공유 메모리 블록의 이름을 제공하여 같은 `ShareableList`에 액세스하는 방법을 보여줍니다:

```
>>> b = shared_memory.ShareableList(range(5))         # In a first process
>>> c = shared_memory.ShareableList(name=b.shm.name)  # In a second process
>>> c
ShareableList([0, 1, 2, 3, 4], name='...')
>>> c[-1] = -999
>>> b[-1]
-999
>>> b.shm.close()
>>> c.shm.close()
>>> c.shm.unlink()
```

The following examples demonstrates that `ShareableList` (and underlying `SharedMemory`) objects can be pickled and unpickled if needed. Note, that it will still be the same shared object. This happens, because the deserialized object has the same unique name and is just attached to an existing object with the same name (if the object is still alive):

```
>>> import pickle
>>> from multiprocessing import shared_memory
>>> sl = shared_memory.ShareableList(range(10))
>>> list(sl)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> deserialized_sl = pickle.loads(pickle.dumps(sl))
>>> list(deserialized_sl)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> sl[0] = -1
>>> deserialized_sl[1] = -2
>>> list(sl)
[-1, -2, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(deserialized_sl)
[-1, -2, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> sl.shm.close()
>>> sl.shm.unlink()
```

18.4 concurrent 패키지

This package contains the following modules:

- `concurrent.futures` – 병렬 작업 시작하기
- `concurrent.interpreters` – Multiple interpreters in the same process

18.5 concurrent.futures — 병렬 작업 실행하기

Added in version 3.2.

Source code: Lib/concurrent/futures/thread.py, Lib/concurrent/futures/process.py, and Lib/concurrent/futures/interpreter.py

---

`concurrent.futures` 모듈은 비동기적으로 콜러블을 실행하는 고수준 인터페이스를 제공합니다.

The asynchronous execution can be performed with threads, using `ThreadPoolExecutor` or `InterpreterPoolExecutor`, or separate processes, using `ProcessPoolExecutor`. Each implements the same interface, which is defined by the abstract `Executor` class.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

18.5.1 Executor 객체

class concurrent.futures.**Executor**

비동기적으로 호출을 실행하는 메서드를 제공하는 추상 클래스입니다. 직접 사용해서는 안 되며, 구체적인 하위 클래스를 통해 사용해야 합니다.

submit(*fn*, /, **args*, ***kwargs*)

콜러블 *fn* 이 `fn(*args, **kwargs)` 처럼 실행되도록 예약하고, 콜러블 객체의 실행을 나타내는 `Future` 객체를 반환합니다.

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map(*fn*, **iterables*, *timeout=None*, *chunksize=1*, *buffersize=None*)

`map(fn, *iterables)` 과 비슷하지만, 다음과 같은 차이가 있습니다:

- The *iterables* are collected immediately rather than lazily, unless a *buffersize* is specified to limit the number of submitted tasks whose results have not yet been yielded. If the buffer is full, iteration over the *iterables* pauses until a result is yielded from the buffer.
- *fn* 는 비동기적으로 실행되며 *fn* 에 대한 여러 호출이 동시에 이루어질 수 있습니다.

반환된 이터레이터는 `__next__()` 가 호출되었을 때, `Executor.map()` 에 대한 최초 호출에서 *timeout* 초 후에도 결과를 사용할 수 없는 경우 `TimeoutError` 를 발생시킵니다. *timeout* 은 int 또는 float가 될 수 있습니다. *timeout* 이 지정되지 않았거나 `None` 인 경우, 대기 시간에는 제한이 없습니다.

fn 호출이 예외를 일으키면, 값을 이터레이터에서 꺼낼 때 해당 예외가 발생합니다.

When using `ProcessPoolExecutor`, this method chops *iterables* into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer. For very long iterables, using a large value for *chunksize* can significantly improve performance compared to the default size of 1. With `ThreadPoolExecutor` and `InterpreterPoolExecutor`, *chunksize* has no effect.

버전 3.5에서 변경: Added the *chunksize* parameter.

버전 3.14에서 변경: Added the *buffersize* parameter.

shutdown(*wait=True*, **, cancel_futures=False*)

현재 계류 중인 퓨처가 실행 완료될 때, 사용 중인 모든 자원을 해제해야 한다는 것을 실행기에 알립니다. 종료(shutdown) 후에 이루어지는 `Executor.submit()` 과 `Executor.map()` 호출은 `RuntimeError` 를 발생시킵니다.

wait 가 `True` 면, 계류 중인 모든 퓨처가 실행을 마치고 실행기와 관련된 자원이 해제될 때까지 이 메서드는 돌아오지 않습니다. *wait* 가 `False` 면, 이 메서드는 즉시 돌아오고 실행기와 연관된 자원은 계류 중인 모든 퓨처가 실행을 마칠 때 해제됩니다. *wait* 의 값과 관계없이, 모든 계류 중인 퓨처가 실행을 마칠 때까지 전체 파이썬 프로그램이 종료되지 않습니다.

*cancel_futures*가 `True`이면, 이 메서드는 실행기가 실행을 시작시키지 않은 계류 중인 모든 퓨처를 취소합니다. *cancel_futures*의 값과 관계없이 완료되었거나 실행 중인 퓨처는 취소되지 않습니다.

*cancel_futures*와 *wait*가 모두 `True`이면, 이 메서드가 반환하기 전에 실행기가 실행을 시작한 모든 퓨처가 완료됩니다. 나머지 퓨처는 취소됩니다.

You can avoid having to call this method explicitly if you use the executor as a *context manager* via the `with` statement, which will shutdown the `Executor` (waiting as if `Executor.shutdown()` were called with *wait* set to `True`):

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

버전 3.9에서 변경: *cancel_futures*를 추가했습니다.

18.5.2 ThreadPoolExecutor

`ThreadPoolExecutor`는 스레드 풀을 사용하여 호출을 비동기적으로 실행하는 `Executor` 서브 클래스입니다.

`Future`와 관련된 콜러블 객체가 다른 `Future`의 결과를 기다릴 때 교착 상태가 발생할 수 있습니다. 예를 들면:

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b 는 a 를 기다리기 때문에 완료되지 않습니다.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a 는 b 를 기다리기 때문에 완료되지 않습니다.
    return 6


executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

그리고:

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # 작업자 스레드가 하나뿐인데, wait_on_future 를 실행하고 있으므로
    # f 는 완료되지 않습니다.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class concurrent.futures.**ThreadPoolExecutor**(*max_workers=None*, *thread_name_prefix=''*, *initializer=None*, *initargs=()*)

최대 *max_workers* 스레드의 풀을 사용하여 호출을 비동기적으로 실행하는 `Executor` 서브 클래스.

All threads enqueued to `ThreadPoolExecutor` will be joined before the interpreter can exit. Note that the exit handler which does this is executed *before* any exit handlers added using `atexit`. This means exceptions

in the main thread must be caught and handled in order to signal threads to exit gracefully. For this reason, it is recommended that `ThreadPoolExecutor` not be used for long-running tasks.

initializer 는 각 작업자 스레드의 시작 부분에서 호출되는 선택적 콜러블입니다; *initargs* 는 initializer 에 전달되는 인자들의 튜플입니다. *initializer* 가 예외를 발생시키는 경우, 현재 계류 중인 모든 작업과 풀에 추가로 작업을 제출하려는 시도는 `BrokenThreadPool` 을 발생시킵니다.

버전 3.5에서 변경: *max_workers* 가 `None` 이거나 주어지지 않았다면, 기본값으로 기계의 프로세서 수에 `5` 를 곱한 값을 사용합니다. `ThreadPoolExecutor` 가 CPU 작업보다는 I/O를 동시에 진행하는데 자주 쓰이고, 작업자의 수가 `ProcessPoolExecutor` 보다 많아야 한다고 가정하고 있습니다.

버전 3.6에서 변경: 디버깅 편의를 위해 사용자가 풀이 만드는 작업자 스레드의 `threading.Thread` 이름을 제어 할 수 있도록, *thread_name_prefix* 매개 변수를 추가했습니다.

버전 3.7에서 변경: *initializer* 및 *initargs* 인자가 추가되었습니다.

버전 3.8에서 변경: *max_workers*의 기본값은 `min(32, os.cpu_count() + 4)` 로 변경됩니다. 이 기본값은 I/O 병목 작업을 위해 최소 5개의 작업자를 유지합니다. GIL을 반납하는 CPU 병목 작업을 위해 최대 32개의 CPU 코어를 사용합니다. 또한 많은 코어를 가진 시스템에서 매우 큰 자원을 묵시적으로 사용하는 것을 방지합니다.

ThreadPoolExecutor는 이제 *max_workers* 작업자 스레드를 시작하기 전에 유휴 작업자 스레드를 재사용합니다.

버전 3.13에서 변경: Default value of *max_workers* is changed to `min(32, (os.process_cpu_count() or 1) + 4)`.

ThreadPoolExecutor 예제

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://nonexistent-subdomain.python.org/']

# 페이지 하나를 가져오고 URL 과 내용을 보고합니다
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# with 문을 사용하여 스레드가 즉시 정리되도록 할 수 있습니다
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # 로드 작업을 시작하고 각 퓨처의 해당 URL을 기록합니다
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

18.5.3 InterpreterPoolExecutor

The `InterpreterPoolExecutor` class uses a pool of interpreters to execute calls asynchronously. It is a `ThreadPoolExecutor` subclass, which means each worker is running in its own thread. The difference here is that each worker has its own interpreter, and runs each task using that interpreter.

The biggest benefit to using interpreters instead of only threads is true multi-core parallelism. Each interpreter has its own *Global Interpreter Lock*, so code running in one interpreter can run on one CPU core, while code in another interpreter runs unblocked on a different core.

The tradeoff is that writing concurrent code for use with multiple interpreters can take extra effort. However, this is because it forces you to be deliberate about how and when interpreters interact, and to be explicit about what data is shared between interpreters. This results in several benefits that help balance the extra effort, including true multi-core parallelism, For example, code written this way can make it easier to reason about concurrency. Another major benefit is that you don' t have to deal with several of the big pain points of using threads, like race conditions.

Each worker' s interpreter is isolated from all the other interpreters. "Isolated" means each interpreter has its own runtime state and operates completely independently. For example, if you redirect `sys.stdout` in one interpreter, it will not be automatically redirected to any other interpreter. If you import a module in one interpreter, it is not automatically imported in any other. You would need to import the module separately in interpreter where you need it. In fact, each module imported in an interpreter is a completely separate object from the same module in a different interpreter, including `sys`, `builtins`, and even `__main__`.

Isolation means a mutable object, or other data, cannot be used by more than one interpreter at the same time. That effectively means interpreters cannot actually share such objects or data. Instead, each interpreter must have its own copy, and you will have to synchronize any changes between the copies manually. Immutable objects and data, like the builtin singletons, strings, and tuples of immutable objects, don' t have these limitations.

Communicating and synchronizing between interpreters is most effectively done using dedicated tools, like those proposed in **PEP 734**. One less efficient alternative is to serialize with `pickle` and then send the bytes over a shared `socket` or `pipe`.

class `concurrent.futures.`**`InterpreterPoolExecutor`**(*max_workers=None*, *thread_name_prefix=''*, *initializer=None*, *initargs=()*)

A `ThreadPoolExecutor` subclass that executes calls asynchronously using a pool of at most *max_workers* threads. Each thread runs tasks in its own interpreter. The worker interpreters are isolated from each other, which means each has its own runtime state and that they can' t share any mutable objects or other data. Each interpreter has its own *Global Interpreter Lock*, which means code run with this executor has true multi-core parallelism.

The optional *initializer* and *initargs* arguments have the same meaning as for `ThreadPoolExecutor`: the initializer is run when each worker is created, though in this case it is run in the worker' s interpreter. The executor serializes the *initializer* and *initargs* using `pickle` when sending them to the worker' s interpreter.

> **참고**
>
> The executor may replace uncaught exceptions from *initializer* with `ExecutionFailed`.

Other caveats from parent `ThreadPoolExecutor` apply here.

`submit()` and `map()` work like normal, except the worker serializes the callable and arguments using `pickle` when sending them to its interpreter. The worker likewise serializes the return value when sending it back.

When a worker' s current task raises an uncaught exception, the worker always tries to preserve the exception as-is. If that is successful then it also sets the `__cause__` to a corresponding `ExecutionFailed` instance, which contains a summary of the original exception. In the uncommon case that the worker is not able to preserve the original as-is then it directly preserves the corresponding `ExecutionFailed` instance instead.

18.5.4 ProcessPoolExecutor

ProcessPoolExecutor 클래스는 프로세스 풀을 사용하여 호출을 비동기적으로 실행하는 *Executor* 서브 클래스입니다. *ProcessPoolExecutor* 는 *multiprocessing* 모듈을 사용합니다. 전역 인터프리터 록 을 피할 수 있도록 하지만, 오직 피클 가능한 객체만 실행되고 반환될 수 있음을 의미합니다.

`__main__` 모듈은 작업자 서브 프로세스가 임포트 할 수 있어야 합니다. 즉, *ProcessPoolExecutor* 는 대화형 인터프리터에서 작동하지 않습니다.

ProcessPoolExecutor 에 제출된 콜러블에서 *Executor* 나 *Future* 메서드를 호출하면 교착 상태가 발생합니다.

Note that the restrictions on functions and arguments needing to picklable as per *multiprocessing.Process* apply when using *submit()* and *map()* on a *ProcessPoolExecutor*. A function defined in a REPL or a lambda should not be expected to work.

class concurrent.futures.**ProcessPoolExecutor**(*max_workers=None*, *mp_context=None*, *initializer=None*, *initargs=()*, *max_tasks_per_child=None*)

최대 *max_workers* 프로세스의 풀을 사용하여 호출을 비동기적으로 실행하는 *Executor* 서브 클래스. *max_workers* 가 `None` 이거나 주어지지 않았다면, *os.process_cpu_count()*를 기본값으로 사용합니다. *max_workers* 가 `0` 보다 작거나 같으면 *ValueError* 가 발생합니다. 윈도우에서, *max_workers*는 `61`보다 작거나 같아야 합니다. 그렇지 않으면 *ValueError*가 발생합니다. *max_workers*가 `None` 이면, 더 많은 프로세서를 사용할 수 있다 할지라도 선택된 기본값은 최대 `61`이 될 것입니다. *mp_context* 는 *multiprocessing* 컨텍스트이거나 `None` 일 수 있습니다. 작업자들을 만드는데 사용될 것입니다. *mp_context* 가 `None` 이거나 주어지지 않으면 기본 *multiprocessing* 컨텍스트가 사용됩니다. 컨텍스트 및 시작 방법을 참조하세요.

initializer 는 각 작업자 프로세스의 시작 부분에서 호출되는 선택적 콜러블입니다; *initargs* 는 initializer 에 전달되는 인자들의 튜플입니다. *initializer* 가 예외를 발생시키는 경우, 현재 계류 중인 모든 작업과 풀에 추가로 작업을 제출하려는 시도는 *BrokenProcessPool* 을 발생시킵니다.

max_tasks_per_child is an optional argument that specifies the maximum number of tasks a single process can execute before it will exit and be replaced with a fresh worker process. By default *max_tasks_per_child* is `None` which means worker processes will live as long as the pool. When a max is specified, the "spawn" multiprocessing start method will be used by default in absence of a *mp_context* parameter. This feature is incompatible with the "fork" start method.

버전 3.3에서 변경: 작업자 프로세스 중 하나가 갑자기 종료되면, *BrokenProcessPool* 오류가 발생합니다. 이전에는, 동작이 정의되지 않았지만, 실행기나 그 퓨처에 대한 연산이 종종 멈추거나 교착 상태에 빠졌습니다.

버전 3.7에서 변경: *mp_context* 인자가 추가되어 사용자가 풀에서 만드는 작업자 프로세스의 시작 방법을 제어 할 수 있습니다.

initializer 및 *initargs* 인자가 추가되었습니다.

버전 3.11에서 변경: *max_tasks_per_child* 인자가 추가되어 사용자가 풀에 있는 작업자의 수명을 제어 할 수 있습니다.

버전 3.12에서 변경: On POSIX systems, if your application has multiple threads and the *multiprocessing* context uses the `"fork"` start method: The *os.fork()* function called internally to spawn workers may raise a *DeprecationWarning*. Pass a *mp_context* configured to use a different start method. See the *os.fork()* documentation for further explanation.

버전 3.13에서 변경: *max_workers* uses *os.process_cpu_count()* by default, instead of *os.cpu_count()*.

버전 3.14에서 변경: The default process start method (see 컨텍스트 및 시작 방법) changed away from *fork*. If you require the *fork* start method for *ProcessPoolExecutor* you must explicitly pass `mp_context=multiprocessing.get_context("fork")`.

terminate_workers()

Attempt to terminate all living worker processes immediately by calling *Process.terminate* on each

of them. Internally, it will also call `Executor.shutdown()` to ensure that all other resources associated with the executor are freed.

After calling this method the caller should no longer submit tasks to the executor.

Added in version 3.14.

kill_workers()

Attempt to kill all living worker processes immediately by calling `Process.kill` on each of them. Internally, it will also call `Executor.shutdown()` to ensure that all other resources associated with the executor are freed.

After calling this method the caller should no longer submit tasks to the executor.

Added in version 3.14.

ProcessPoolExecutor 예제

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

18.5.5 Future 객체

`Future` 클래스는 콜러블 객체의 비동기 실행을 캡슐화합니다. `Future` 인스턴스는 `Executor.submit()` 에 의해 생성됩니다.

class concurrent.futures.**Future**

콜러블 객체의 비동기 실행을 캡슐화합니다. `Future` 인스턴스는 `Executor.submit()` 에 의해 생성되며 테스트를 제외하고는 직접 생성되어서는 안 됩니다.

cancel()

호출을 취소하려고 시도합니다. 호출이 현재 실행 중이거나 실행 종료했고 취소할 수 없는 경우 메서드는 `False` 를 반환하고, 그렇지 않으면 호출이 취소되고 메서드는 `True` 를 반환합니다.

cancelled()

호출이 성공적으로 취소되었으면 `True` 를 반환합니다.

running()

호출이 현재 실행 중이고 취소할 수 없는 경우 `True` 를 반환합니다.

done()

호출이 성공적으로 취소되었거나 실행이 완료되었으면 `True` 를 반환합니다.

result(*timeout=None*)

호출이 반환한 값을 돌려줍니다. 호출이 아직 완료되지 않는 경우, 이 메서드는 *timeout* 초까지 대기합니다. *timeout* 초 내에 호출이 완료되지 않으면 `TimeoutError` 가 발생합니다. *timeout* 은 int 또는 float가 될 수 있습니다. *timeout* 이 지정되지 않았거나 `None` 인 경우, 대기 시간에는 제한이 없습니다.

완료하기 전에 퓨처가 취소되면 `CancelledError` 가 발생합니다.

호출이 예외를 일으키는 경우, 이 메서드는 같은 예외를 발생시킵니다.

exception(*timeout=None*)

호출이 일으킨 예외를 돌려줍니다. 호출이 아직 완료되지 않는 경우, 이 메서드는 *timeout* 초까지 대기합니다. *timeout* 초 내에 호출이 완료되지 않으면 `TimeoutError` 가 발생합니다. *timeout* 은 int 또는 float가 될 수 있습니다. *timeout* 이 지정되지 않았거나 `None` 인 경우, 대기 시간에는 제한이 없습니다.

완료하기 전에 퓨처가 취소되면 `CancelledError` 가 발생합니다.

호출이 예외 없이 완료되면, `None` 이 반환됩니다.

add_done_callback(*fn*)

콜러블 *fn* 을 퓨처에 연결합니다. *fn* 은 퓨처가 취소되거나 실행이 종료될 때 퓨처를 유일한 인자로 호출됩니다.

추가된 콜러블은 추가된 순서대로 호출되며, 항상 콜러블을 추가한 프로세스에 속하는 스레드에서 호출됩니다. 콜러블이 `Exception` 서브 클래스를 발생시키면, 로그 되고 무시됩니다. 콜러블이 `BaseException` 서브 클래스를 발생시키면, 동작은 정의되지 않습니다.

퓨처가 이미 완료되었거나 취소된 경우 *fn* 이 즉시 호출됩니다.

다음 `Future` 메서드는 단위 테스트와 `Executor` 의 구현을 위한 것입니다.

set_running_or_notify_cancel()

이 메서드는 `Future`와 관련된 작업을 실행하기 전에 `Executor` 구현에 의해서만 호출되거나 단위 테스트에서만 호출되어야 합니다.

메서드가 `False` 를 반환하면, `Future` 가 취소된 것입니다. 즉 `Future.cancel()` 이 호출되었고 `True`를 반환했습니다. `Future` 완료를 기다리는 (즉, `as_completed()` 또는 `wait()`를 통해) 모든 스레드는 깨어납니다.

메서드가 `True` 를 반환하면, `Future` 가 취소되지 않았고 실행 상태로 진입했습니다. 즉 `Future.running()` 을 호출하면 `True` 가 반환됩니다.

이 메서드는 한 번만 호출 할 수 있으며, `Future.set_result()` 또는 `Future.set_exception()` 이 호출 된 후에는 호출할 수 없습니다.

set_result(*result*)

`Future`와 관련된 작업 결과를 *result* 로 설정합니다.

이 메서드는 `Executor` 구현과 단위 테스트에서만 사용해야 합니다.

버전 3.8에서 변경: 이 메서드는 `Future`가 이미 완료되었으면 `concurrent.futures.InvalidStateError`를 발생시킵니다.

set_exception(*exception*)

*Future*와 관련된 작업 결과를 *Exception* *exception* 으로 설정합니다.

이 메서드는 *Executor* 구현과 단위 테스트에서만 사용해야 합니다.

버전 3.8 에서 변경: 이 메서드는 *Future*가 이미 완료되었으면 *concurrent.futures.InvalidStateError*를 발생시킵니다.

18.5.6 모듈 함수

concurrent.futures.**wait**(*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)

fs 로 주어진 여러 (서로 다른 *Executor* 인스턴스가 만든 것들도 가능합니다) *Future* 인스턴스들이 완료할 때까지 기다립니다. *fs* 로 주어진 중복 퓨처는 제거되고 한 번만 반환됩니다. 집합들의 이름있는 2-튜플을 돌려줍니다. done 이라는 이름의 첫 번째 집합은 대기가 끝나기 전에 완료된 (끝났거나 취소된) 퓨처를 담고 있습니다. not_done 이라는 이름의 두 번째 집합은 완료되지 않은 (계류 중이거나 실행 중인) 퓨처를 담고 있습니다.

timeout 은 반환하기 전에 대기 할 최대 시간(초)을 제어하는 데 사용할 수 있습니다. *timeout* 은 int 또는 float가 될 수 있습니다. *timeout* 이 지정되지 않았거나 None 인 경우, 대기 시간에는 제한이 없습니다.

return_when 은, 이 함수가 언제 반환되어야 하는지를 나타냅니다. 다음 상수 중 하나여야 합니다:

상수	설명
concurrent.futures.**FIRST_COMPLETED**	퓨처가 어느 하나라도 끝나거나 취소될 때 함수가 반환됩니다.
concurrent.futures.**FIRST_EXCEPTION**	어느 한 퓨처가 예외를 일으켜 완료하면 함수가 반환됩니다. 어떤 퓨처도 예외를 발생시키지 않으면 *ALL_COMPLETED*와 같습니다.
concurrent.futures.**ALL_COMPLETED**	모든 퓨처가 끝나거나 취소되면 함수가 반환됩니다.

concurrent.futures.**as_completed**(*fs*, *timeout=None*)

fs 로 주어진 여러 (서로 다른 *Executor* 인스턴스가 만든 것들도 가능합니다) 퓨처들이 완료되는 대로 (끝났거나 취소된 퓨처) 일드 하는 *Future* 인스턴스의 이터레이터를 반환합니다. *fs* 에 중복된 퓨처가 들어있으면 한 번만 반환됩니다. *as_completed()* 가 호출되기 전에 완료한 모든 퓨처들이 먼저 일드 됩니다. 반환된 이터레이터는, *__next__()* 가 호출되고, *as_completed()* 호출 시점으로부터 *timeout* 초 후에 결과를 얻을 수 없는 경우 *TimeoutError* 를 발생시킵니다. *timeout* 은 int 또는 float가 될 수 있습니다. *timeout* 이 지정되지 않았거나 None 인 경우, 대기 시간에는 제한이 없습니다.

더 보기

PEP 3148 – 퓨처 - 계산을 비동기적으로 실행
파이썬 표준 라이브러리에 포함하기 위해, 이 기능을 설명한 제안.

18.5.7 예외 클래스

exception concurrent.futures.**CancelledError**

퓨처가 취소될 때 발생합니다.

exception concurrent.futures.**TimeoutError**

*TimeoutError*의 폐지된 별칭, 퓨처 연산이 지정된 시간제한을 초과할 때 발생합니다.

버전 3.11 에서 변경: This class was made an alias of *TimeoutError*.

exception concurrent.futures.**BrokenExecutor**

RuntimeError 에서 파생됩니다, 이 예외 클래스는 어떤 이유로 실행기가 망가져서 새 작업을 제출하거나 실행할 수 없을 때 발생합니다.

Added in version 3.7.

exception concurrent.futures.**InvalidStateError**

퓨처에 현재 상태에서 허용되지 않는 연산이 수행될 때 발생합니다.

Added in version 3.8.

exception concurrent.futures.thread.**BrokenThreadPool**

BrokenExecutor 에서 파생됩니다, 이 예외 클래스는 *ThreadPoolExecutor* 의 작업자 중 하나가 초기화에 실패했을 때 발생합니다.

Added in version 3.7.

exception concurrent.futures.interpreter.**BrokenInterpreterPool**

Derived from *BrokenThreadPool*, this exception class is raised when one of the workers of a *InterpreterPoolExecutor* has failed initializing.

Added in version 3.14.

exception concurrent.futures.interpreter.**ExecutionFailed**

Raised from *InterpreterPoolExecutor* when the given initializer fails or from *submit()* when there' s an uncaught exception from the submitted task.

Added in version 3.14.

exception concurrent.futures.process.**BrokenProcessPool**

BrokenExecutor 에서 파생됩니다 (예전에는 *RuntimeError*), 이 예외 클래스는 *ProcessPoolExecutor* 의 작업자 중 하나가 깨끗하지 못한 방식으로 (예를 들어, 외부에서 강제 종료된 경우) 종료되었을 때 발생합니다.

Added in version 3.3.

18.6 concurrent.interpreters — Multiple interpreters in the same process

Added in version 3.14.

Source code: Lib/concurrent/interpreters

The concurrent.interpreters module constructs higher-level interfaces on top of the lower level _interpreters module.

The module is primarily meant to provide a basic API for managing interpreters (AKA "subinterpreters") and running things in them. Running mostly involves switching to an interpreter (in the current thread) and calling a function in that execution context.

For concurrency, interpreters themselves (and this module) don' t provide much more than isolation, which on its own isn' t useful. Actual concurrency is available separately through *threads* See *below*

더 보기

InterpreterPoolExecutor

combines threads with interpreters in a familiar interface.

isolating-extensions-howto

how to update an extension module to support multiple interpreters

PEP 554

PEP 734

PEP 684

가용성: not WASI.

This module does not work or is not available on WebAssembly. See 웹어셈블리 플랫폼 for more information.

18.6.1 Key details

Before we dive in further, there are a small number of details to keep in mind about using multiple interpreters:

- *isolated*, by default
- no implicit threads
- not all PyPI packages support use in multiple interpreters yet

18.6.2 Introduction

An "interpreter" is effectively the execution context of the Python runtime. It contains all of the state the runtime needs to execute a program. This includes things like the import state and builtins. (Each thread, even if there' s only the main thread, has some extra runtime state, in addition to the current interpreter, related to the current exception and the bytecode eval loop.)

The concept and functionality of the interpreter have been a part of Python since version 2.2, but the feature was only available through the C-API and not well known, and the *isolation* was relatively incomplete until version 3.12.

Multiple Interpreters and Isolation

A Python implementation may support using multiple interpreters in the same process. CPython has this support. Each interpreter is effectively isolated from the others (with a limited number of carefully managed process-global exceptions to the rule).

That isolation is primarily useful as a strong separation between distinct logical components of a program, where you want to have careful control of how those components interact.

> **참고**
>
> Interpreters in the same process can technically never be strictly isolated from one another since there are few restrictions on memory access within the same process. The Python runtime makes a best effort at isolation but extension modules may easily violate that. Therefore, do not use multiple interpreters in security-sensitive situations, where they shouldn' t have access to each other' s data.

Running in an Interpreter

Running in a different interpreter involves switching to it in the current thread and then calling some function. The runtime will execute the function using the current interpreter' s state. The `concurrent.interpreters` module provides a basic API for creating and managing interpreters, as well as the switch-and-call operation.

No other threads are automatically started for the operation. There is *a helper* for that though. There is another dedicated helper for calling the builtin `exec()` in an interpreter.

When `exec()` (or `eval()`) are called in an interpreter, they run using the interpreter' s `__main__` module as the "globals" namespace. The same is true for functions that aren' t associated with any module. This is the same as how scripts invoked from the command-line run in the `__main__` module.

Concurrency and Parallelism

As noted earlier, interpreters do not provide any concurrency on their own. They strictly represent the isolated execution context the runtime will use *in the current thread*. That isolation makes them similar to processes, but they still enjoy in-process efficiency, like threads.

All that said, interpreters do naturally support certain flavors of concurrency. There' s a powerful side effect of that isolation. It enables a different approach to concurrency than you can take with async or threads. It' s a similar concurrency model to CSP or the actor model, a model which is relatively easy to reason about.

You can take advantage of that concurrency model in a single thread, switching back and forth between interpreters, Stackless-style. However, this model is more useful when you combine interpreters with multiple threads. This mostly involves starting a new thread, where you switch to another interpreter and run what you want there.

Each actual thread in Python, even if you' re only running in the main thread, has its own *current* execution context. Multiple threads can use the same interpreter or different ones.

At a high level, you can think of the combination of threads and interpreters as threads with opt-in sharing.

As a significant bonus, interpreters are sufficiently isolated that they do not share the *GIL*, which means combining threads with multiple interpreters enables full multi-core parallelism. (This has been the case since Python 3.12.)

Communication Between Interpreters

In practice, multiple interpreters are useful only if we have a way to communicate between them. This usually involves some form of message passing, but can even mean sharing data in some carefully managed way.

With this in mind, the `concurrent.interpreters` module provides a `queue.Queue` implementation, available through `create_queue()`.

"Sharing" Objects

Any data actually shared between interpreters loses the thread-safety provided by the *GIL*. There are various options for dealing with this in extension modules. However, from Python code the lack of thread-safety means objects can' t actually be shared, with a few exceptions. Instead, a copy must be created, which means mutable objects won' t stay in sync.

By default, most objects are copied with `pickle` when they are passed to another interpreter. Nearly all of the immutable builtin objects are either directly shared or copied efficiently. For example:

- `None`
- `bool` (`True` and `False`)
- `bytes`
- `str`
- `int`
- `float`
- `tuple` (of similarly supported objects)

There is a small number of Python types that actually share mutable data between interpreters:

- `memoryview`
- `Queue`

18.6.3 Reference

This module defines the following functions:

`concurrent.interpreters.`**`list_all`**`()`

Return a `list` of `Interpreter` objects, one for each existing interpreter.

concurrent.interpreters.**get_current**()

Return an *Interpreter* object for the currently running interpreter.

concurrent.interpreters.**get_main**()

Return an *Interpreter* object for the main interpreter. This is the interpreter the runtime created to run the *REPL* or the script given at the command-line. It is usually the only one.

concurrent.interpreters.**create**()

Initialize a new (idle) Python interpreter and return a *Interpreter* object for it.

concurrent.interpreters.**create_queue**()

Initialize a new cross-interpreter queue and return a *Queue* object for it.

Interpreter objects

class concurrent.interpreters.**Interpreter**(*id*)

A single interpreter in the current process.

Generally, *Interpreter* shouldn' t be called directly. Instead, use *create()* or one of the other module functions.

id

(read-only)

The underlying interpreter' s ID.

whence

(read-only)

A string describing where the interpreter came from.

is_running()

Return `True` if the interpreter is currently executing code in its `__main__` module and `False` otherwise.

close()

Finalize and destroy the interpreter.

prepare_main(*ns=None*, ***kwargs*)

Bind objects in the interpreter' s `__main__` module.

Some objects are actually shared and some are copied efficiently, but most are copied via *pickle*. See *"Sharing" Objects*.

exec(*code*, */*, *dedent=True*)

Run the given source code in the interpreter (in the current thread).

call(*callable*, */*, **args*, ***kwargs*)

Return the result of calling running the given function in the interpreter (in the current thread).

call_in_thread(*callable*, */*, **args*, ***kwargs*)

Run the given function in the interpreter (in a new thread).

Exceptions

exception concurrent.interpreters.**InterpreterError**

This exception, a subclass of *Exception*, is raised when an interpreter-related error happens.

exception concurrent.interpreters.**InterpreterNotFoundError**

This exception, a subclass of *InterpreterError*, is raised when the targeted interpreter no longer exists.

exception concurrent.interpreters.**ExecutionFailed**

This exception, a subclass of *InterpreterError*, is raised when the running code raised an uncaught exception.

excinfo

A basic snapshot of the exception raised in the other interpreter.

exception concurrent.interpreters.**NotShareableError**

This exception, a subclass of *TypeError*, is raised when an object cannot be sent to another interpreter.

Communicating Between Interpreters

class concurrent.interpreters.**Queue**(*id*)

A wrapper around a low-level, cross-interpreter queue, which implements the *queue.Queue* interface. The underlying queue can only be created through *create_queue()*.

Some objects are actually shared and some are copied efficiently, but most are copied via *pickle*. See *"Sharing" Objects*.

id

(read-only)

The queue' s ID.

exception concurrent.interpreters.**QueueEmptyError**

This exception, a subclass of *queue.Empty*, is raised from `Queue.get()` and `Queue.get_nowait()` when the queue is empty.

exception concurrent.interpreters.**QueueFullError**

This exception, a subclass of *queue.Full*, is raised from `Queue.put()` and `Queue.put_nowait()` when the queue is full.

18.6.4 Basic usage

Creating an interpreter and running code in it:

```
from concurrent import interpreters

interp = interpreters.create()

# Run in the current OS thread.

interp.exec('print("spam!")')

interp.exec("""if True:
    print('spam!')
    """)

from textwrap import dedent
interp.exec(dedent("""
    print('spam!')
    """))

def run(arg):
    return arg

res = interp.call(run, 'spam!')
print(res)

def run():
    print('spam!')

interp.call(run)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# Run in new OS thread.

t = interp.call_in_thread(run)
t.join()
```

18.7 subprocess — 서브 프로세스 관리

소스 코드: Lib/subprocess.py

---

subprocess 모듈은 새로운 프로세스를 생성하고, 그들의 입력/출력/에러 파이프에 연결하고, 반환 코드를 얻을 수 있도록 합니다. 이 모듈은 몇 가지 이전 모듈과 함수를 대체하려고 합니다:

```
os.system
os.spawn*
```

subprocess 모듈을 사용하여 이러한 모듈과 함수를 교체하는 방법에 대한 정보는 다음 섹션에서 확인할 수 있습니다.

더 보기

PEP 324 – subprocess 모듈을 제안하는 PEP

가용성: not Android, not iOS, not WASI.

이 모듈은 모바일 플랫폼이나 웹어셈블리 플랫폼에서 지원되지 않습니다.

18.7.1 subprocess 모듈 사용하기

서브 프로세스 호출에 권장되는 접근법은 처리할 수 있는 모든 사용 사례에 *run()* 함수를 사용하는 것입니다. 고급 사용 사례의 경우, 하부 *Popen* 인터페이스를 직접 사용할 수 있습니다.

subprocess.**run**(*args, *, stdin=None, input=None, stdout=None, stderr=None, capture_output=False, shell=False, cwd=None, timeout=None, check=False, encoding=None, errors=None, text=None, env=None, universal_newlines=None, **other_popen_kwargs*)

*args*가 기술하는 명령을 실행합니다. 명령이 완료될 때까지 기다린 다음, *CompletedProcess* 인스턴스를 반환합니다.

위에 표시된 인자는 가장 일반적인 인자로, 아래 자주 사용되는 인자에서 설명됩니다 (따라서 약식 서명에 키워드 전용 표기법을 사용합니다). 전체 함수 서명은 *Popen* 생성자와 거의 같습니다 - 이 함수에 대한 대부분의 인자는 해당 인터페이스로 전달됩니다. (*timeout*, *input*, *check* 및 *capture_output* 은 아닙니다.)

*capture_output*이 참이면, 표준 출력 (stdout) 과 표준 에러 (stderr) 가 캡처됩니다. 사용되면, 내부 *Popen* 객체는 자동으로 *stdout*과 *stderr*이 모두 *PIPE*로 설정되어 만들어집니다. *stdout*과 *stderr* 인자는 *capture_output*과 동시에 제공되지 않을 수 있습니다. 두 스트림을 모두 캡처하여 하나로 결합하려면, *capture_output*을 사용하는 대신, *stdout*을 *PIPE*로 *stderr*을 *STDOUT*으로 설정하십시오.

*timeout*은 초 단위로 지정될 수 있으며, 내부적으로 *Popen.communicate()*로 전달됩니다. 시간제한이 만료되면, 자식 프로세스를 죽이고 기다립니다. 자식 프로세스가 종료된 후 *TimeoutExpired* 예외가 다시 발생합니다. 많은 플랫폼 API에서 초기 프로세스 생성 자체는 중단될 수 없기 때문에, 프로세스 생성에 소요되는 시간이 지나기 전에 시간제한 예외를 본다는 보장이 없습니다.

input 인자는 *Popen.communicate()*로 전달되고, 그래서 서브 프로세스의 표준 입력으로 전달됩니다. 사용된다면 바이트 시퀀스이거나, *encoding*이나 *errors*가 지정되었거나 *text*가 참이면 문자열이어

야 합니다. 사용하면, 내부 `Popen` 객체가 자동으로 *stdin*을 `PIPE`로 설정하여 만들어지고 *stdin* 인자는 사용되지 않을 수 있습니다.

*check*가 참이고, 프로세스가 0이 아닌 종료 코드로 종료되면, `CalledProcessError` 예외가 발생합니다. 해당 예외의 어트리뷰트가 인자 및 종료 코드와 캡처되었다면 stdout과 stderr을 담습니다.

*encoding*이나 *errors*가 지정되거나, *text*가 참이면, stdin, stdout 및 stderr의 파일 객체는 지정된 *encoding*과 *errors* 또는 `io.TextIOWrapper` 기본값을 사용하여 텍스트 모드로 열립니다. *universal_newlines* 인자는 *text*와 동등하고 이전 버전과의 호환성을 위해 제공됩니다. 기본적으로, 파일 객체는 바이너리 모드로 열립니다.

*env*가 `None`이 아니면, 새 프로세스의 환경 변수를 정의하는 매핑이어야 합니다; 이것은 현재 프로세스의 환경을 상속하는 기본 동작 대신 사용됩니다. `Popen`으로 직접 전달됩니다. 이 매핑은 모든 플랫폼에서 문자열에서 문자열로, POSIX 플랫폼에서는 바이트열에서 바이트열로 일 수 있으며, 이는 `os.environ`이나 `os.environb`와 유사합니다.

예:

```
>>> subprocess.run(["ls", "-l"])  # 출력을 캡처하지 않습니다
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], capture_output=True)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n', stderr=b'')
```

Added in version 3.5.

버전 3.6에서 변경: *encoding*과 *errors* 매개 변수를 추가했습니다

버전 3.7에서 변경: *universal_newlines*의 더 이해하기 쉬운 별칭으로 *text* 매개 변수를 추가했습니다. *capture_output* 매개 변수를 추가했습니다.

버전 3.12에서 변경: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

class subprocess.**CompletedProcess**

완료된 프로세스를 나타내는, `run()`의 반환 값.

args

프로세스를 시작하는 데 사용된 인자. 리스트나 문자열일 수 있습니다.

returncode

자식 프로세스의 종료 상태. 일반적으로, 종료 상태 0은 성공적으로 실행되었음을 나타냅니다.

음수 값 `-N`은 자식이 시그널 `N`에 의해 종료되었음을 나타냅니다 (POSIX 전용).

stdout

자식 프로세스에서 캡처된 stdout. 바이트 시퀀스, 또는 `run()`이 encoding, errors, 또는 text=True로 호출되었으면 문자열. stdout이 캡처되지 않았으면 `None`.

`stderr=subprocess.STDOUT`으로 프로세스를 실행했으면, stdout과 stderr이 이 어트리뷰트에 결합하고, `stderr`은 `None`이 됩니다.

stderr

자식 프로세스에서 캡처된 stderr. 바이트 시퀀스, 또는 `run()`이 encoding, errors, 또는 text=True로 호출되었으면 문자열. stderr이 캡처되지 않았으면 `None`.

check_returncode()

returncode가 0이 아니면, CalledProcessError를 발생시킵니다.

Added in version 3.5.

subprocess.**DEVNULL**

Popen의 *stdin*, *stdout* 또는 *stderr* 인자로 사용할 수 있고 특수 파일 os.devnull이 사용될 것임을 나타내는 특수 값.

Added in version 3.3.

subprocess.**PIPE**

Popen의 *stdin*, *stdout* 또는 *stderr* 인자로 사용할 수 있고 표준 스트림에 대한 파이프를 열어야 함을 나타내는 특수 값. Popen.communicate()에서 가장 유용합니다.

subprocess.**STDOUT**

Popen의 *stderr* 인자로 사용할 수 있고 표준 에러가 표준 출력과 같은 핸들로 가야 함을 나타내는 특수 값.

exception subprocess.**SubprocessError**

이 모듈의 다른 모든 예외에 대한 베이스 클래스.

Added in version 3.3.

exception subprocess.**TimeoutExpired**

자식 프로세스를 기다리는 동안 시간제한이 만료될 때 발생하는 SubprocessError의 서브 클래스.

cmd

자식 프로세스를 생성하는 데 사용된 명령.

timeout

초 단위의 시간제한.

output

Output of the child process if it was captured by run() or check_output(). Otherwise, None. This is always bytes when any output was captured regardless of the text=True setting. It may remain None instead of b'' when no output was observed.

stdout

output의 별칭, stderr과의 대칭을 위한 것입니다.

stderr

Stderr output of the child process if it was captured by run(). Otherwise, None. This is always bytes when stderr output was captured regardless of the text=True setting. It may remain None instead of b'' when no stderr output was observed.

Added in version 3.3.

버전 3.5에서 변경: *stdout*과 *stderr* 어트리뷰트가 추가되었습니다

exception subprocess.**CalledProcessError**

check_call(), check_output() 또는 run() (check=True와 함께) 에 의해 실행되는 프로세스가 0이 아닌 종료 상태를 반환할 때 발생하는 SubprocessError 의 서브 클래스.

returncode

자식 프로세스의 종료 상태. 시그널로 인해 프로세스가 종료되었으면, 음의 시그널 번호가 됩니다.

cmd

자식 프로세스를 생성하는 데 사용된 명령.

output

run()이나 check_output()에 의해 캡처되었다면 자식 프로세스의 출력. 그렇지 않으면, None.

stdout

output의 별칭, `stderr`과의 대칭을 위한 것입니다.

stderr

`run()`에 의해 캡처되었다면 자식 프로세스의 stderr 출력. 그렇지 않으면, `None`.

버전 3.5에서 변경: *stdout*과 *stderr* 어트리뷰트가 추가되었습니다

자주 사용되는 인자

다양한 사용 사례를 지원하기 위해, `Popen` 생성자(및 편의 함수)는 많은 선택적 인자를 받아들입니다. 가장 일반적인 사용 사례에서, 이러한 인자 중 많은 부분을 기본값으로 안전하게 남겨둘 수 있습니다. 가장 흔히 필요한 인자는 다음과 같습니다:

*args*는 모든 호출에 필요하며 문자열 또는 프로그램 인자의 시퀀스여야 합니다. 인자의 시퀀스를 제공하는 것이 일반적으로 선호되는데, 모듈이 필요한 인자의 이스케이프와 인용을 처리하도록 하기 때문입니다 (예를 들어 파일 이름에 공백을 허용하도록). 단일 문자열을 전달하면, *shell*이 `True`(아래를 참조하십시오)이거나, 그렇지 않으면 문자열은 단순히 인자를 지정하지 않고 실행할 프로그램의 이름이어야 합니다.

stdin, *stdout* 및 *stderr*은 각각 실행된 프로그램의 표준 입력, 표준 출력 및 표준 에러 파일 핸들을 지정합니다. 유효한 값은 `None`, `PIPE`, `DEVNULL`, 기존 파일 기술자 (양의 정수) 및 유효한 파일 기술자를 가진 기존 파일 객체입니다. 기본 설정 `None`으로는, 리디렉션은 발생하지 않습니다. `PIPE`는 자식에 대한 새 파이프를 만들어야 함을 나타냅니다. `DEVNULL`은 특수 파일 `os.devnull`이 사용될 것임을 나타냅니다. 또한, *stderr*은 `STDOUT`일 수 있으며, 이는 자식 프로세스의 stderr 데이터가 *stdout*과 같은 파일 핸들로 캡처되어야 함을 나타냅니다.

*encoding*이나 *errors*가 지정되거나, *text*(*universal_newlines*라고도 합니다)가 참이면, 파일 객체 *stdin*, *stdout* 및 *stderr*은 호출에 지정된 *encoding*과 *errors* 또는 `io.TextIOWrapper`의 기본값을 사용하여 텍스트 모드로 열립니다.

*stdin*의 경우, 입력의 줄 종료 문자 `'\n'`이 기본 줄 구분자 `os.linesep`으로 변환됩니다. *stdout*과 *stderr*의 경우, 출력의 모든 줄 종료가 `'\n'`으로 변환됩니다. 자세한 정보는 생성자에 대한 *newline* 인자가 `None`일 때에 관한 `io.TextIOWrapper` 클래스의 설명서를 참조하십시오.

텍스트 모드를 사용하지 않으면, *stdin*, *stdout* 및 *stderr*이 바이너리 스트림으로 열립니다. 인코딩이나 줄 종료 변환이 수행되지 않습니다.

버전 3.6에서 변경: *encoding*과 *errors* 매개 변수를 추가했습니다.

버전 3.7에서 변경: *text* 매개 변수를 *universal_newlines*의 별칭으로 추가했습니다.

> **참고**
>
> 파일 객체 `Popen.stdin`, `Popen.stdout` 및 `Popen.stderr`의 newlines 어트리뷰트는 `Popen.communicate()` 메서드에 의해 갱신되지 않습니다.

*shell*이 `True`이면, 지정된 명령이 셸을 통해 실행됩니다. 이것은 대부분의 시스템 셸에서 제공하는 것보다 향상된 제어 흐름을 위해 파이썬을 주로 사용하면서, 여전히 셸 파이프, 파일명 와일드카드, 환경 변수 확장 및 사용자 홈 디렉터리로의 ~ 확장과 같은 다른 셸 기능에 대한 편리한 액세스를 원할 때 유용할 수 있습니다. 그러나, 파이썬 자체가 많은 셸과 유사한 기능의 구현을 제공함에 유의하십시오 (특히, `glob`, `fnmatch`, `os.walk()`, `os.path.expandvars()`, `os.path.expanduser()` 및 `shutil`).

버전 3.3에서 변경: *universal_newlines*가 `True`일 때, 클래스는 `locale.getpreferredencoding()` 대신 인코딩 `locale.getpreferredencoding(False)`를 사용합니다. 이 변경에 대한 자세한 내용은 `io.TextIOWrapper` 클래스를 참조하십시오.

> **참고**
>
> `shell=True`를 사용하기 전에 보안 고려 사항 섹션을 읽으십시오.

이러한 옵션들은, 다른 모든 옵션과 함께, `Popen` 생성자 설명서에 더 자세히 설명되어 있습니다.

Popen 생성자

이 모듈에서 하부 프로세스 생성과 관리는 `Popen` 클래스에 의해 처리됩니다. 개발자가 편의 함수가 다루지 않는 덜 일반적인 사례를 처리할 수 있도록 큰 유연성을 제공합니다.

class subprocess.**Popen**(*args, bufsize=-1, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=True, shell=False, cwd=None, env=None, universal_newlines=None, startupinfo=None, creationflags=0, restore_signals=True, start_new_session=False, pass_fds=(), *, group=None, extra_groups=None, user=None, umask=-1, encoding=None, errors=None, text=None, pipesize=-1, process_group=None*)

새 프로세스에서 자식 프로그램을 실행합니다. POSIX에서, 클래스는 자식 프로그램을 실행하는 데 `os.execvpe()`와 유사한 동작을 사용합니다. 윈도우에서 클래스는 윈도우 `CreateProcess()` 함수를 사용합니다. `Popen`에 대한 인자는 다음과 같습니다.

*args*는 프로그램 인자의 시퀀스이거나 그렇지 않으면 단일한 문자열이나 경로류 객체여야 합니다. 기본적으로, *args*가 시퀀스이면 실행할 프로그램은 *args*의 첫 번째 항목입니다. *args*가 문자열이면, 해석은 플랫폼에 따라 다르며 아래에 설명되어 있습니다. 기본 동작과의 추가 차이점에 관해서는 *shell*과 *executable* 인자를 참조하십시오. 별도의 언급이 없는 한, *args*를 시퀀스로 전달하는 것이 좋습니다.

> **경고**
>
> For maximum reliability, use a fully qualified path for the executable. To search for an unqualified name on `PATH`, use `shutil.which()`. On all platforms, passing `sys.executable` is the recommended way to launch the current Python interpreter again, and use the `-m` command-line format to launch an installed module.
>
> Resolving the path of *executable* (or the first item of *args*) is platform dependent. For POSIX, see `os.execvpe()`, and note that when resolving or searching for the executable path, *cwd* overrides the current working directory and *env* can override the `PATH` environment variable. For Windows, see the documentation of the `lpApplicationName` and `lpCommandLine` parameters of WinAPI `CreateProcess`, and note that when resolving or searching for the executable path with `shell=False`, *cwd* does not override the current working directory and *env* cannot override the `PATH` environment variable. Using a full path avoids all of these variations.

시퀀스로 외부 프로그램에 어떤 인자를 전달하는 예는 다음과 같습니다:

```
Popen(["/usr/bin/git", "commit", "-m", "Fixes a bug."])
```

POSIX에서, *args*가 문자열이면, 문자열은 실행할 프로그램의 이름이나 경로로 해석됩니다. 그러나, 이것은 프로그램에 인자를 전달하지 않을 때만 수행할 수 있습니다.

> **참고**
>
> 특히 복잡한 경우에, 셸 명령을 인자의 시퀀스로 나누는 방법이 명확하지 않을 수 있습니다. `shlex.split()`는 *args*의 올바른 토큰화를 결정하는 방법을 보여줄 수 있습니다:
>
> ```
> >>> import shlex, subprocess
> >>> command_line = input()
> /bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
> >>> args = shlex.split(command_line)
> >>> print(args)
> ['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd',
> ↪"echo '$MONEY'"]
> >>> p = subprocess.Popen(args) # 성공!
> ```
>
> 특히 셸에서 공백으로 구분된 옵션(가령 *-input*)과 인자(가령 *eggs.txt*)는 별도의 리스트 요소로

> 들어가지만, 셸에서 사용될 때 인용(quoting)이나 역 슬래시 이스케이프가 필요한 인자(가령 스페이스가 포함된 파일명이나 위에 표시된 *echo* 명령)는 단일 리스트 요소임에 유의하십시오.

윈도우에서, *args*가 시퀀스이면, 윈도우에서 인자 시퀀스를 문자열로 변환하기에 설명된 방식으로 문자열로 변환됩니다. 하부 `CreateProcess()`가 문자열에서 작동하기 때문입니다.

버전 3.6에서 변경: POSIX에서, *args* 매개 변수는 *shell*이 `False`이면 경로류 객체나 경로류 객체를 포함하는 시퀀스를 받아들입니다.

버전 3.8에서 변경: 윈도우에서, *args* 매개 변수는 *shell*이 `False`이면 경로류 객체나 바이트열과 경로류 객체를 포함하는 시퀀스를 받아들입니다.

shell 인자(기본값은 `False`)는 셸을 실행할 프로그램으로 사용할지를 지정합니다. *shell*이 `True`이면, *args*를 시퀀스가 아닌 문자열로 전달하는 것이 좋습니다.

POSIX에서 `shell=True`일 때, 셸의 기본값은 `/bin/sh`입니다. *args*가 문자열이면, 문자열은 셸을 통해 실행할 명령을 지정합니다. 이것은 문자열이 프롬프트에서 입력할 때와 똑같이 포맷되어야 한다는 것을 의미합니다. 예를 들어, 스페이스가 포함된 파일명을 인용하거나 역 슬래시 이스케이핑하는 것을 포함합니다. *args*가 시퀀스이면, 첫 번째 항목이 명령 문자열을 지정하고, 추가 항목은 셸 자체에 대한 추가 인자로 처리됩니다. 즉, *Popen*은 다음과 동등한 것을 수행합니다:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

윈도우에서 `shell=True`일 때, `COMSPEC` 환경 변수는 기본 셸을 지정합니다. 윈도우에서 `shell=True`를 지정해야 하는 유일한 경우는 실행하려는 명령이 셸에 내장되었을 때입니다(예를 들어 **dir**이나 **copy**). 배치 파일이나 콘솔 기반 실행 파일을 실행하기 위해 `shell=True`가 필요하지 않습니다.

> **참고**
>
> `shell=True`를 사용하기 전에 보안 고려 사항 섹션을 읽으십시오.

*bufsize*는 stdin/stdout/stderr 파이프 파일 객체를 만들 때 *open()* 함수에 해당 인자로 제공됩니다:

- `0`은 버퍼링 되지 않음을 의미합니다(읽기와 쓰기는 한 번의 시스템 호출이며 짧게 반환할 수 있습니다)
- `1`은 줄 버퍼링을 의미합니다(`text=True`이거나 `universal_newlines=True`인 경우만 사용할 수 있습니다)
- 다른 양수 값은 대략 그 크기의 버퍼를 사용함을 의미합니다.
- 음의 bufsize(기본값)는 시스템 기본값 io.DEFAULT_BUFFER_SIZE가 사용됨을 의미합니다.

버전 3.3.1에서 변경: *bufsize*의 기본값은 이제 -1이 되어 대부분의 코드가 기대하는 동작과 일치하도록 기본적으로 버퍼링을 활성화합니다. 파이썬 3.2.4와 3.3.1 이전 버전에서는 버퍼링 되지 않고 짧은 읽기를 허용하는 `0`으로 기본값이 잘못 설정되었습니다. 이것은 의도하지 않은 것이며 대부분의 코드가 기대하는 파이썬 2의 동작과 일치하지 않았습니다.

executable 인자는 실행할 대체 프로그램을 지정합니다. 거의 필요하지 않습니다. `shell=False`일 때, *executable*은 *args*에 의해 지정된 프로그램을 대체합니다. 그러나, 원래 *args*는 여전히 프로그램으로 전달됩니다. 대부분의 프로그램은 *args*에 의해 지정된 프로그램을 명령 이름으로 취급합니다, 그래서 실제로 실행되는 프로그램과 다를 수 있습니다. POSIX에서, *args* 이름은 **ps**와 같은 유틸리티에서 실행 파일의 표시 이름이 됩니다. `shell=True`이면, POSIX에서 *executable* 인자는 기본 `/bin/sh`의 대체 셸을 지정합니다.

버전 3.6에서 변경: POSIX에서 *executable* 매개 변수는 경로류 객체를 받아들입니다.

버전 3.8에서 변경: 윈도우에서 *executable* 매개 변수는 바이트열과 경로류 객체를 받아들입니다.

버전 3.12에서 변경: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

stdin, *stdout* 및 *stderr*은 각각 실행된 프로그램의 표준 입력, 표준 출력 및 표준 에러 파일 핸들을 지정합니다. 유효한 값은 `None`, `PIPE`, `DEVNULL`, 기존 파일 기술자 (양의 정수) 및 유효한 파일 기술자를 가진 기존 파일 객체입니다. 기본 설정 `None`으로는, 리디렉션은 발생하지 않습니다. `PIPE`는 자식에 대한 새 파이프를 만들어야 함을 나타냅니다. `DEVNULL`은 특수 파일 `os.devnull`이 사용될 것임을 나타냅니다. 또한, *stderr*은 `STDOUT`일 수 있으며, 이는 응용 프로그램의 stderr 데이터가 *stdout*과 같은 파일 핸들로 캡처되어야 함을 나타냅니다.

*preexec_fn*이 콜러블 객체로 설정되면, 이 객체는 자식 프로세스가 실행되기 직전에 자식 프로세스에서 호출됩니다. (POSIX 전용)

경고

preexec_fn 매개 변수는 응용 프로그램에 스레드가 있을 때 사용하기에 안전하지 않습니다. exec가 호출되기 전에 자식 프로세스가 교착 상태에 빠질 수 있습니다.

참고

자식의 환경을 수정해야 하면 *preexec_fn*에서 하지 말고 *env* 매개 변수를 사용하십시오. *start_new_session*과 *process_group* 매개 변수는 자식에서 `os.setsid()`나 `os.setpgid()`를 호출하려고 *preexec_fn*을 사용하는 코드를 대체해야 합니다.

버전 3.8에서 변경: 서브 인터프리터에서 *preexec_fn* 매개 변수가 더는 지원되지 않습니다. 서브 인터프리터에서 매개 변수를 사용하면 `RuntimeError`가 발생합니다. 새로운 제약 사항은 mod_wsgi, uWSGI 및 기타 내장(embedded) 환경에 배치된 응용 프로그램에 영향을 줄 수 있습니다.

*close_fds*가 참이면, `0`, `1` 및 `2`를 제외한 모든 파일 기술자가 자식 프로세스가 실행되기 전에 닫힙니다. 그렇지 않고 *close_fds*가 거짓이면, 파일 기술자는 파일 기술자의 상속에 설명된 대로 상속 가능 플래그를 따릅니다.

윈도우에서, *close_fds*가 참이면 `STARTUPINFO.lpAttributeList`의 `handle_list` 요소에 명시적으로 전달되거나 표준 핸들 리디렉션에 의하지 않는 한 자식 프로세스가 핸들을 상속하지 않습니다.

버전 3.2에서 변경: *close_fds*의 기본값은 `False`에서 위에서 설명한 것으로 변경되었습니다.

버전 3.7에서 변경: 윈도우에서 표준 핸들을 리디렉션 할 때 *close_fds*의 기본값이 `False`에서 `True`로 변경되었습니다. 이제 표준 핸들을 리디렉션 할 때 *close_fds*를 `True`로 설정할 수 있습니다.

*pass_fds*는 부모와 자식 사이에 열려있는 파일 기술자의 선택적 시퀀스입니다. *pass_fds*를 제공하면 *close_fds*가 `True`가 됩니다. (POSIX 전용)

버전 3.2에서 변경: *pass_fds* 매개 변수가 추가되었습니다.

*cwd*가 `None`이 아니면, 함수는 자식을 실행하기 전에 작업 디렉터리를 *cwd*로 변경합니다. *cwd*는 문자열, 바이트열 또는 경로류 객체일 수 있습니다. POSIX 에서, 함수는 실행 파일 경로가 상대 경로이면 *cwd*를 기준으로 *executable*(또는 *args*의 첫 번째 항목)을 찾습니다.

버전 3.6에서 변경: POSIX에서 *cwd* 매개 변수는 경로류 객체를 받아들입니다.

버전 3.7에서 변경: 윈도우에서 *cwd* 매개 변수는 경로류 객체를 받아들입니다.

버전 3.8에서 변경: 윈도우에서 *cwd* 매개 변수는 바이트열 객체를 받아들입니다.

*restore_signals*가 참(기본값)이면 파이썬이 SIG_IGN으로 설정한 모든 시그널은 실행 전에 자식 프로세스에서 SIG_DFL로 복원됩니다. 현재 여기에는 SIGPIPE, SIGXFZ 및 SIGXFSZ 시그널이 포함됩니다. (POSIX 전용)

버전 3.2에서 변경: *restore_signals*가 추가되었습니다.

*start_new_session*이 참이면 서브 프로세스를 실행하기 전에 자식 프로세스에서 `setsid()` 시스템 호출이 수행됩니다.

가용성: POSIX

버전 3.2에서 변경: *start_new_session*이 추가되었습니다.

*process_group*이 음이 아닌 정수면, 서브 프로세스를 실행하기 전에 자식 프로세스에서 `setpgid(0, value)` 시스템 호출이 수행됩니다.

가용성: POSIX

버전 3.11에서 변경: *process_group*이 추가되었습니다.

*group*이 `None`이 아니면, 서브 프로세스를 실행하기 전에 자식 프로세스에서 setregid() 시스템 호출이 수행됩니다. 제공된 값이 문자열이면, *grp.getgrnam()*을 통해 조회되고 `gr_gid`의 값이 사용됩니다. 값이 정수이면, 그대로 전달됩니다. (POSIX 전용)

가용성: POSIX

Added in version 3.9.

*extra_groups*가 `None`이 아니면, 서브 프로세스를 실행하기 전에 자식 프로세스에서 setgroups() 시스템 호출이 수행됩니다. *extra_groups*에 제공된 문자열은 *grp.getgrnam()*을 통해 조회되며 `gr_gid`의 값이 사용됩니다. 정숫값은 그대로 전달됩니다. (POSIX 전용)

가용성: POSIX

Added in version 3.9.

*user*가 `None`이 아니면, 서브 프로세스를 실행하기 전에 자식 프로세스에서 setreuid() 시스템 호출이 수행됩니다. 제공된 값이 문자열이면, *pwd.getpwnam()*을 통해 조회되고 `pw_uid`의 값이 사용됩니다. 값이 정수이면, 그대로 전달됩니다. (POSIX 전용)

가용성: POSIX

Added in version 3.9.

*umask*가 음이 아니면, 서브 프로세스를 실행하기 전에 자식 프로세스에서 umask() 시스템 호출이 수행됩니다.

가용성: POSIX

Added in version 3.9.

*env*가 `None`이 아니면, 새 프로세스의 환경 변수를 정의하는 매핑이어야 합니다; 이것은 현재 프로세스의 환경을 상속하는 기본 동작 대신 사용됩니다. 이 매핑은 모든 플랫폼에서 문자열에서 문자열로, POSIX 플랫폼에서는 바이트열에서 바이트열로 일 수 있으며, 이는 *os.environ*이나 *os.environb*와 유사합니다.

> **참고**
>
> 지정되면, *env*는 프로그램 실행에 필요한 모든 변수를 제공해야 합니다. 윈도우에서, side-by-side assembly를 실행하려면 지정된 *env*에 유효한 `SystemRoot`가 **반드시** 포함되어야 합니다.

*encoding*이나 *errors*가 지정되거나, *text*가 참이면, 파일 객체 *stdin*, *stdout* 및 *stderr*은 위의 자주 사용되는 인자에서 설명한 대로 지정된 *encoding*과 *errors*로 텍스트 모드로 열립니다. *universal_newlines* 인자는 *text*와 동등하며 이전 버전과의 호환성을 위해 제공됩니다. 기본적으로, 파일 객체는 바이너리 모드로 열립니다.

Added in version 3.6: *encoding*과 *errors*가 추가되었습니다.

Added in version 3.7: *text*가 *universal_newlines*의 더 가독성 있는 별칭으로 추가되었습니다.

주어지면, *startupinfo*는 *STARTUPINFO* 객체가 되며, 하부 `CreateProcess` 함수로 전달됩니다.

주어지면, *creationflags*는 다음 플래그 중 하나 이상일 수 있습니다:

- *CREATE_NEW_CONSOLE*
- *CREATE_NEW_PROCESS_GROUP*
- *ABOVE_NORMAL_PRIORITY_CLASS*

- *BELOW_NORMAL_PRIORITY_CLASS*
- *HIGH_PRIORITY_CLASS*
- *IDLE_PRIORITY_CLASS*
- *NORMAL_PRIORITY_CLASS*
- *REALTIME_PRIORITY_CLASS*
- *CREATE_NO_WINDOW*
- *DETACHED_PROCESS*
- *CREATE_DEFAULT_ERROR_MODE*
- *CREATE_BREAKAWAY_FROM_JOB*

pipesize can be used to change the size of the pipe when *PIPE* is used for *stdin*, *stdout* or *stderr*. The size of the pipe is only changed on platforms that support this (only Linux at this time of writing). Other platforms will ignore this parameter.

버전 3.10에서 변경: *pipesize* 매개 변수를 추가했습니다.

Popen 객체는 `with` 문을 통해 컨텍스트 관리자로 지원됩니다; 빠져나갈 때, 표준 파일 기술자가 닫히고 프로세스를 기다립니다.

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

Popen과 이 모듈에서 이를 사용하는 다른 함수는 인자 `executable`, `args`, `cwd` 및 `env`로 감사 이벤트 `subprocess.Popen`을 발생시킵니다. `args`의 값은 플랫폼에 따라 단일 문자열이나 문자열 리스트일 수 있습니다.

버전 3.2에서 변경: 컨텍스트 관리자 지원이 추가되었습니다.

버전 3.6에서 변경: 자식 프로세스가 여전히 실행 중이면 이제 Popen 파괴자(destructor)가 *ResourceWarning* 경고를 발생시킵니다.

버전 3.8에서 변경: Popen은 성능 향상을 위해 때에 따라 *os.posix_spawn()*을 사용할 수 있습니다. 리눅스용 윈도우 서브 시스템(Windows Subsystem for Linux)과 QEMU 사용자 에뮬레이션(QEMU User Emulation)에서, *os.posix_spawn()*을 사용하는 Popen 생성자는 더는 프로그램 누락과 같은 에러에 대한 예외를 발생시키지 않지만, 자식 프로세스는 0이 아닌 *returncode*로 실패합니다.

예외

새 프로그램이 실행을 시작하기 전에 자식 프로세스에서 발생한 예외는 부모에서 다시 발생합니다.

The most common exception raised is *OSError*. This occurs, for example, when trying to execute a non-existent file. Applications should prepare for *OSError* exceptions. Note that, when `shell=True`, *OSError* will be raised by the child only if the selected shell itself was not found. To determine if the shell failed to find the requested application, it is necessary to check the return code or output from the subprocess.

*Popen*이 유효하지 않은 인자로 호출되면 *ValueError*가 발생합니다.

호출된 프로세스가 0이 아닌 반환 코드를 반환하면 *check_call()*과 *check_output()*은 *CalledProcessError*를 발생시킵니다.

*run()*과 *Popen.communicate()*와 같은 *timeout* 매개 변수를 받아들이는 모든 함수와 메서드는 프로세스가 종료되기 전에 시간제한이 만료되면 *TimeoutExpired*를 발생시킵니다.

이 모듈에 정의된 예외는 모두 *SubprocessError*를 상속합니다.

Added in version 3.3: *SubprocessError* 베이스 클래스가 추가되었습니다.

18.7.2 보안 고려 사항

다른 popen 함수와 달리, 이 라이브러리는 시스템 셸을 호출하도록 묵시적으로 선택하지 않습니다. 이는 셸 메타 문자를 포함한 모든 문자를 자식 프로세스로 안전하게 전달할 수 있음을 의미합니다. `shell=True`를 통해 셸을 명시적으로 호출하면, 셸 주입 취약점을 피하고자 모든 공백과 메타 문자를 적절하게 인용하는 것은 응용 프로그램의 책임입니다. 일부 플랫폼에서, 이 이스케이핑을 위해 `shlex.quote()`를 사용할 수 있습니다.

On Windows, batch files (`*.bat` or `*.cmd`) may be launched by the operating system in a system shell regardless of the arguments passed to this library. This could result in arguments being parsed according to shell rules, but without any escaping added by Python. If you are intentionally launching a batch file with arguments from untrusted sources, consider passing `shell=True` to allow Python to escape special characters. See gh-114539 for additional discussion.

18.7.3 Popen 객체

`Popen` 클래스의 인스턴스에는 다음과 같은 메서드가 있습니다:

`Popen.`**`poll`**`()`

자식 프로세스가 종료되었는지 확인합니다. `returncode` 어트리뷰트를 설정하고 반환합니다. 그렇지 않으면, `None`을 반환합니다.

`Popen.`**`wait`**`(`*timeout=None*`)`

자식 프로세스가 종료될 때까지 기다립니다. `returncode` 어트리뷰트를 설정하고 반환합니다.

timeout 초 후에 프로세스가 종료되지 않으면, `TimeoutExpired` 예외를 발생시킵니다. 이 예외를 잡고 다시 기다리는 것은 안전합니다.

> **참고**
>
> 이는 `stdout=PIPE`나 `stderr=PIPE`를 사용하고 자식 프로세스가 파이프에 너무 많은 출력을 보내서 OS 파이프 버퍼가 더 많은 데이터를 받아들일 때까지 기다리느라 블록하면 교착 상태에 빠집니다. 파이프를 사용할 때 이를 피하려면 `Popen.communicate()`를 사용하십시오.

> **참고**
>
> `timeout` 매개 변수가 `None`이 아닐 때, (POSIX 에서) 이 함수는 비지 루프(비 블로킹 호출과 짧은 잠자기)를 사용하여 구현됩니다. 비동기 대기에는 `asyncio` 모듈을 사용하십시오: `asyncio.create_subprocess_exec`를 참조하십시오.

버전 3.3에서 변경: *timeout*이 추가되었습니다.

`Popen.`**`communicate`**`(`*input=None, timeout=None*`)`

프로세스와 상호 작용합니다: stdin에 데이터를 보냅니다. 파일 끝에 도달할 때까지 stdout과 stderr에서 데이터를 읽습니다. 프로세스가 종료될 때까지 기다리고, `returncode` 어트리뷰트를 설정합니다. 선택적 *input* 인자는 자식 프로세스로 전송될 데이터이거나, 자식으로 데이터를 보내지 않으면 `None`이어야 합니다. 스트림이 텍스트 모드로 열렸으면, *input*은 문자열이어야 합니다. 그렇지 않으면, 바이트열이어야 합니다.

`communicate()`는 튜플 `(stdout_data, stderr_data)`를 반환합니다. 스트림이 텍스트 모드로 열렸으면 데이터는 문자열이 됩니다. 그렇지 않으면 바이트열입니다.

프로세스의 stdin으로 데이터를 보내려면, `stdin=PIPE`를 사용하여 Popen 객체를 만들어야 함에 유의하십시오. 마찬가지로, 결과 튜플에서 `None` 이외의 것을 얻으려면, `stdout=PIPE` 및/또는 `stderr=PIPE`도 제공해야 합니다.

timeout 초 후에 프로세스가 종료되지 않으면, `TimeoutExpired` 예외가 발생합니다. 이 예외를 잡고 통신을 다시 시도해도 출력이 손실되지 않습니다.

시간제한이 만료되면 자식 프로세스를 죽이지 않습니다, 올바르게 정리하려면 제대로 작동하는 응용 프로그램은 자식 프로세스를 죽이고 통신을 완료해야 합니다:

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

> **참고**
>
> 읽은 데이터는 메모리에 버퍼링 되므로, 데이터 크기가 크거나 무제한이면 이 메서드를 사용하지 마십시오.

버전 3.3에서 변경: *timeout*이 추가되었습니다.

`Popen.send_signal(signal)`

시그널 *signal*을 자식에게 보냅니다.

프로세스가 완료되었으면 아무것도 하지 않습니다.

> **참고**
>
> 윈도우에서, SIGTERM은 `terminate()`의 별칭입니다. CTRL_C_EVENT와 CTRL_BREAK_EVENT는 `CREATE_NEW_PROCESS_GROUP`을 포함하는 *creationflags* 매개 변수로 시작된 프로세스로 보낼 수 있습니다.

`Popen.terminate()`

자식을 멈춥니다. POSIX OS에서 이 메서드는 `SIGTERM`을 자식에게 보냅니다. 윈도우에서는 자식을 중지하기 위해 Win32 API 함수 `TerminateProcess()`가 호출됩니다.

`Popen.kill()`

자식을 죽입니다. POSIX OS에서 이 함수는 SIGKILL을 자식에게 보냅니다. 윈도우에서 `kill()`은 `terminate()`의 별칭입니다.

The following attributes are also set by the class for you to access. Reassigning them to new values is unsupported:

`Popen.args`

`Popen`으로 전달된 *args* 인자 – 프로그램 인자의 시퀀스거나 단일 문자열.

Added in version 3.3.

`Popen.stdin`

stdin 인자가 `PIPE`이면, 이 어트리뷰트는 `open()`이 반환한 쓰기 가능한 스트림 객체입니다. *encoding*이나 *errors* 인자가 지정되었거나 *text*나 *universal_newlines* 인자가 `True`이면, 스트림은 텍스트 스트림이고, 그렇지 않으면 바이트 스트림입니다. *stdin* 인자가 `PIPE`가 아니면, 이 어트리뷰트는 `None`입니다.

`Popen.stdout`

stdout 인자가 `PIPE`이면, 이 어트리뷰트는 `open()`이 반환한 읽기 가능한 스트림 객체입니다. 스트림에서 읽으면 자식 프로세스의 출력을 제공합니다. *encoding*이나 *errors* 인자가 지정되었거나 *text*나 *universal_newlines* 인자가 `True`이면, 스트림은 텍스트 스트림이고, 그렇지 않으면 바이트 스트림입니다. *stdout* 인자가 `PIPE`가 아니면, 이 어트리뷰트는 `None`입니다.

`Popen.stderr`

stderr 인자가 `PIPE`이면, 이 어트리뷰트는 `open()`이 반환한 읽기 가능한 스트림 객체입니다. 스트림에서 읽으면 자식 프로세스의 에러 출력을 제공합니다. *encoding*이나 *errors* 인자가 지정되었거나 *text*

나 *universal_newlines* 인자가 `True`이면, 스트림은 텍스트 스트림이고, 그렇지 않으면 바이트 스트림입니다. *stderr* 인자가 `PIPE`가 아니면, 어트리뷰트는 `None`입니다.

> **경고**
>
> 다른 OS 파이프 버퍼가 차고 자식 프로세스를 블로킹하는 것으로 인한 교착 상태를 피하려면 `.stdin.write`, `.stdout.read` 또는 `.stderr.read` 대신 `communicate()`를 사용하십시오.

`Popen.`**`pid`**

자식 프로세스의 프로세스 ID

shell 인자를 `True`로 설정하면, 이것은 생성된 셸의 프로세스 ID입니다.

`Popen.`**`returncode`**

자식 반환 코드. 초기값은 `None`이며, `returncode`는 `poll()`, `wait()` 또는 `communicate()` 메서드가 프로세스가 종료되었음을 감지하면 설정합니다.

A `None` value indicates that the process hadn' t yet terminated at the time of the last method call.

음수 값 `-N`은 자식이 시그널 `N`에 의해 종료되었음을 나타냅니다 (POSIX 전용).

18.7.4 윈도우 Popen 도우미

`STARTUPINFO` 클래스와 다음 상수는 윈도우에서만 사용 가능합니다.

class `subprocess.`**`STARTUPINFO`**(**, dwFlags=0, hStdInput=None, hStdOutput=None, hStdError=None, wShowWindow=0, lpAttributeList=None*)

윈도우 STARTUPINFO 구조체의 부분적인 지원이 `Popen` 생성에 사용됩니다. 다음 어트리뷰트는 키워드 전용 인자로 전달하여 설정할 수 있습니다.

버전 3.7에서 변경: 키워드 전용 인자 지원이 추가되었습니다.

`dwFlags`

프로세스가 창을 만들 때 특정 `STARTUPINFO` 어트리뷰트가 사용되는지를 결정하는 비트 필드.

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_
↪USESHOWWINDOW
```

`hStdInput`

`dwFlags`가 `STARTF_USESTDHANDLES`를 지정하면, 이 어트리뷰트는 프로세스의 표준 입력 핸들입니다. `STARTF_USESTDHANDLES`가 지정되지 않으면, 표준 입력의 기본값은 키보드 버퍼입니다.

`hStdOutput`

`dwFlags`가 `STARTF_USESTDHANDLES`를 지정하면, 이 어트리뷰트는 프로세스의 표준 출력 핸들입니다. 그렇지 않으면, 이 어트리뷰트가 무시되고 표준 출력의 기본값은 콘솔 창의 버퍼입니다.

`hStdError`

`dwFlags`가 `STARTF_USESTDHANDLES`를 지정하면, 이 어트리뷰트는 프로세스의 표준 에러 핸들입니다. 그렇지 않으면, 이 어트리뷰트는 무시되고 표준 에러의 기본값은 콘솔 창의 버퍼입니다.

`wShowWindow`

`dwFlags`가 `STARTF_USESHOWWINDOW`를 지정하면, 이 어트리뷰트는 `SW_SHOWDEFAULT`를 제외하고 ShowWindow 함수의 `nCmdShow` 매개 변수에 지정할 수 있는 값 중 어느 것이건 될 수 있습니다. 그렇지 않으면, 이 어트리뷰트는 무시됩니다.

이 어트리뷰트에 `SW_HIDE`가 제공됩니다. `Popen`이 `shell=True`와 함께 호출될 때 사용됩니다.

lpAttributeList

STARTUPINFOEX에 제공된 대로 프로세스 생성을 위한 추가 어트리뷰트 딕셔너리, UpdateProcThreadAttribute를 참조하십시오.

지원되는 어트리뷰트:

handle_list

상속될 핸들의 시퀀스. 비어 있지 않으면 *close_fds*는 참이어야 합니다.

Popen 생성자에 전달될 때 *os.set_handle_inheritable()*로 핸들을 일시적으로 상속할 수 있도록 만들어야 합니다, 그렇지 않으면 윈도우 에러 ERROR_INVALID_PARAMETER (87)로 *OSError*가 발생합니다.

> **경고**
>
> 다중 스레드 프로세스에서, 이 기능을 *os.system()*과 같은 모든 핸들을 상속하는 다른 프로세스 생성 함수에 대한 동시 호출과 결합할 때 상속 가능으로 표시된 핸들의 누수를 피하도록 주의하십시오. 이것은 일시적으로 상속 가능한 핸들을 만드는 표준 핸들 리디렉션에도 적용됩니다.

Added in version 3.7.

윈도우 상수

subprocess 모듈은 다음 상수를 노출합니다.

subprocess.**STD_INPUT_HANDLE**

표준 입력 장치. 처음에는, 콘솔 입력 버퍼입니다, CONIN$.

subprocess.**STD_OUTPUT_HANDLE**

표준 출력 장치. 처음에는, 활성 콘솔 화면 버퍼입니다, CONOUT$.

subprocess.**STD_ERROR_HANDLE**

표준 에러 장치. 처음에는, 활성 콘솔 화면 버퍼입니다, CONOUT$.

subprocess.**SW_HIDE**

창을 숨깁니다. 다른 창이 활성화됩니다.

subprocess.**STARTF_USESTDHANDLES**

STARTUPINFO.hStdInput, *STARTUPINFO.hStdOutput* 및 *STARTUPINFO.hStdError* 어트리뷰트에 추가 정보가 포함되었음을 지정합니다.

subprocess.**STARTF_USESHOWWINDOW**

STARTUPINFO.wShowWindow 어트리뷰트에 추가 정보가 포함되었음을 지정합니다.

subprocess.**STARTF_FORCEONFEEDBACK**

A *STARTUPINFO.dwFlags* parameter to specify that the *Working in Background* mouse cursor will be displayed while a process is launching. This is the default behavior for GUI processes.

Added in version 3.13.

subprocess.**STARTF_FORCEOFFFEEDBACK**

A *STARTUPINFO.dwFlags* parameter to specify that the mouse cursor will not be changed when launching a process.

Added in version 3.13.

subprocess.**CREATE_NEW_CONSOLE**

새로운 프로세스는 부모의 콘솔을 상속(기본값)하는 대신 새로운 콘솔을 갖습니다.

subprocess.**CREATE_NEW_PROCESS_GROUP**

새 프로세스 그룹이 만들어지도록 지정하는 *Popen* creationflags 매개 변수. 이 플래그는 서브프로세스에 *os.kill()*을 사용하기 위해 필요합니다.

*CREATE_NEW_CONSOLE*이 지정되면 이 플래그는 무시됩니다.

subprocess.**ABOVE_NORMAL_PRIORITY_CLASS**

새 프로세스가 평균 우선순위를 초과하도록 지정하는 *Popen* creationflags 매개 변수.

Added in version 3.7.

subprocess.**BELOW_NORMAL_PRIORITY_CLASS**

새 프로세스가 평균 우선순위보다 낮도록 지정하는 *Popen* creationflags 매개 변수.

Added in version 3.7.

subprocess.**HIGH_PRIORITY_CLASS**

새 프로세스가 높은 우선순위를 갖도록 지정하는 *Popen* creationflags 매개 변수입니다.

Added in version 3.7.

subprocess.**IDLE_PRIORITY_CLASS**

새 프로세스가 유휴 (가장 낮은) 우선순위를 갖도록 지정하는 *Popen* creationflags 매개 변수.

Added in version 3.7.

subprocess.**NORMAL_PRIORITY_CLASS**

새 프로세스가 보통 우선순위를 갖도록 지정하는 *Popen* creationflags 매개 변수. (기본값)

Added in version 3.7.

subprocess.**REALTIME_PRIORITY_CLASS**

새 프로세스가 실시간 우선순위를 갖도록 지정하는 *Popen* creationflags 매개 변수. REALTIME_PRIORITY_CLASS를 사용하면 마우스 입력, 키보드 입력 및 백그라운드 디스크 플러시를 관리하는 시스템 스레드가 중단되므로 거의 사용하지 않아야 합니다. 이 클래스는 하드웨어와 직접 "대화" 하거나 중단이 제한되어야 하는 짧은 작업을 수행하는 응용 프로그램에 적합할 수 있습니다.

Added in version 3.7.

subprocess.**CREATE_NO_WINDOW**

새 프로세스가 창을 만들지 않도록 지정하는 *Popen* creationflags 매개 변수.

Added in version 3.7.

subprocess.**DETACHED_PROCESS**

새 프로세스가 부모의 콘솔을 상속하지 않도록 지정하는 *Popen* creationflags 매개 변수. 이 값은 CREATE_NEW_CONSOLE과 함께 사용할 수 없습니다.

Added in version 3.7.

subprocess.**CREATE_DEFAULT_ERROR_MODE**

새 프로세스가 호출하는 프로세스의 에러 모드를 상속하지 않도록 지정하는 *Popen* creationflags 매개 변수. 대신, 새 프로세스는 기본 에러 모드를 갖습니다. 이 기능은 하드(hard) 에러가 비활성화된 상태로 실행되는 다중 스레드 셸 응용 프로그램에 특히 유용합니다.

Added in version 3.7.

subprocess.**CREATE_BREAKAWAY_FROM_JOB**

새 프로세스가 잡(job)과 연관되지 않도록 지정하는 *Popen* creationflags 매개 변수.

Added in version 3.7.

18.7.5 오래된 고수준 API

파이썬 3.5 이전에는, 이 세 가지 함수가 subprocess에 대한 고수준 API로 구성되었습니다. 이제 많은 경우에 `run()`을 사용할 수 있지만, 많은 기존 코드가 이러한 함수를 호출합니다.

`subprocess.call`(*args, *, stdin=None, stdout=None, stderr=None, shell=False, cwd=None, timeout=None, **other_popen_kwargs*)

*args*로 기술된 명령을 실행합니다. 명령이 완료될 때까지 기다린 후 `returncode` 어트리뷰트를 반환합니다.

stdout이나 stderr을 캡처해야 하는 코드는 대신 `run()`을 사용해야 합니다:

```
run(...).returncode
```

stdout이나 stderr을 억제하려면, `DEVNULL` 값을 제공하십시오.

위에 표시된 인자는 단지 몇 가지 일반적인 인자입니다. 전체 함수 서명은 `Popen` 생성자의 서명과 같습니다 - 이 함수는 *timeout* 이외의 모든 제공된 인자를 해당 인터페이스로 직접 전달합니다.

> **참고**
>
> 이 함수에 `stdout=PIPE`나 `stderr=PIPE`를 사용하지 마십시오. 파이프를 읽지 않기 때문에 자식 프로세스가 OS 파이프 버퍼를 가득 채우기 충분한 출력을 생성하면 자식 프로세스가 블록 됩니다.

버전 3.3에서 변경: *timeout*이 추가되었습니다.

버전 3.12에서 변경: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

`subprocess.check_call`(*args, *, stdin=None, stdout=None, stderr=None, shell=False, cwd=None, timeout=None, **other_popen_kwargs*)

인자로 명령을 실행합니다. 명령이 완료될 때까지 기다립니다. 반환 코드가 0이면 반환합니다, 그렇지 않으면 `CalledProcessError`를 발생시킵니다. `CalledProcessError` 객체는 `returncode` 어트리뷰트에 반환 코드를 갖습니다. `check_call()`이 프로세스를 시작하지 못하면 발생된 예외를 확산합니다.

stdout이나 stderr을 캡처해야 하는 코드는 대신 `run()`을 사용해야 합니다:

```
run(..., check=True)
```

stdout이나 stderr을 억제하려면, `DEVNULL` 값을 제공하십시오.

위에 표시된 인자는 단지 몇 가지 일반적인 인자입니다. 전체 함수 서명은 `Popen` 생성자의 서명과 같습니다 - 이 함수는 *timeout* 이외의 모든 제공된 인자를 해당 인터페이스로 직접 전달합니다.

> **참고**
>
> 이 함수에 `stdout=PIPE`나 `stderr=PIPE`를 사용하지 마십시오. 파이프를 읽지 않기 때문에 자식 프로세스가 OS 파이프 버퍼를 가득 채우기 충분한 출력을 생성하면 자식 프로세스가 블록 됩니다.

버전 3.3에서 변경: *timeout*이 추가되었습니다.

버전 3.12에서 변경: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

`subprocess.check_output`(*args, *, stdin=None, stderr=None, shell=False, cwd=None, encoding=None, errors=None, universal_newlines=None, timeout=None, text=None, **other_popen_kwargs*)

인자로 명령을 실행하고 출력을 반환합니다.

반환 코드가 0이 아니면 *CalledProcessError* 가 발생합니다. *CalledProcessError* 객체는 *returncode* 어트리뷰트에 반환 코드가 있고 *output* 어트리뷰트에 출력이 있습니다.

이것은 다음과 동등합니다:

```
run(..., check=True, stdout=PIPE).stdout
```

위에 표시된 인자는 단지 몇 가지 일반적인 인자입니다. 전체 함수 서명은 *run()*의 서명과 거의 같습니다 - 대부분의 인자는 해당 인터페이스로 직접 전달됩니다. *run()* 동작과 비교할 때 한가지 API 편차가 있습니다: `input=None`을 전달하면 부모의 표준 입력 파일 핸들을 사용하는 대신 `input=b''` (또는 다른 인자에 따라 `input=''`) 와 같게 동작합니다.

기본적으로, 이 함수는 데이터를 인코딩된 바이트열로 반환합니다. 출력 데이터의 실제 인코딩은 호출되는 명령에 따라 달라질 수 있어서, 텍스트로의 디코딩은 종종 응용 프로그램 수준에서 처리해야 합니다.

자주 사용되는 인자와 *run()*에 설명된 대로 *text*, *encoding*, *errors* 또는 *universal_newlines*를 `True`로 설정하면 이 동작을 재정의할 수 있습니다.

결과에서 표준 에러도 캡처하려면 `stderr=subprocess.STDOUT`을 사용하십시오:

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

Added in version 3.1.

버전 3.3에서 변경: *timeout*이 추가되었습니다.

버전 3.4에서 변경: *input* 키워드 인자에 대한 지원이 추가되었습니다.

버전 3.6에서 변경: *encoding*과 *errors*가 추가되었습니다. 자세한 내용은 *run()*을 참조하십시오.

Added in version 3.7: *text*가 *universal_newlines*의 더 가독성 있는 별칭으로 추가되었습니다.

버전 3.12에서 변경: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

18.7.6 이전 함수를 subprocess 모듈로 교체하기

이 섹션에서, "a는 b가 됩니다"는 b가 a의 대체로 사용될 수 있음을 의미합니다.

> **참고**
>
> 이 섹션의 모든 "a" 함수는 실행되는 프로그램을 찾을 수 없으면 (다소간) 조용히 실패합니다; "b" 대체는 대신 *OSError*를 발생시킵니다.
>
> 또한, 요청된 연산이 0이 아닌 반환 코드를 생성하면 *check_output()*을 사용한 대체는 *CalledProcessError*로 실패합니다. 출력은 여전히 발생한 예외의 *output* 어트리뷰트로 사용 가능합니다.

다음 예에서는, 관련 함수들을 *subprocess* 모듈에서 이미 임포트 했다고 가정합니다.

/bin/sh 셸 명령 치환 교체하기

```
output=$(mycmd myarg)
```

는 다음처럼 됩니다:

```
output = check_output(["mycmd", "myarg"])
```

셸 파이프라인 교체하기

```
output=$(dmesg | grep hda)
```

는 다음처럼 됩니다:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # p2가 종료되면 p1이 SIGPIPE를 수신하도록 허용합니다.
output = p2.communicate()[0]
```

p2가 p1보다 먼저 종료될 때 p1이 SIGPIPE를 수신하려면 p2를 시작한 후 `p1.stdout.close()` 호출이 중요합니다.

또는, 신뢰할 수 있는 입력의 경우, 셸의 자체 파이프라인 지원을 계속 사용할 수 있습니다:

```
output=$(dmesg | grep hda)
```

는 다음처럼 됩니다:

```
output = check_output("dmesg | grep hda", shell=True)
```

os.system() 교체하기

```
sts = os.system("mycmd" + " myarg")
# 는 다음처럼 됩니다
retcode = call("mycmd" + " myarg", shell=True)
```

노트:

- 보통 셸을 통해 프로그램을 호출할 필요는 없습니다.
- The *call()* return value is encoded differently to that of *os.system()*.
- The *os.system()* function ignores SIGINT and SIGQUIT signals while the command is running, but the caller must do this separately when using the *subprocess* module.

더욱 현실적인 예는 다음과 같습니다:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

os.spawn 패밀리 교체하기

P_NOWAIT 예:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT 예:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

벡터 예:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

환경 예:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

Replacing os.popen()

반환 코드 처리는 다음과 같이 번역됩니다:

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

18.7.7 레거시 셸 호출 함수

이 모듈은 2.x commands 모듈의 다음과 같은 레거시 함수도 제공합니다. 이러한 연산은 시스템 셸을 묵시적으로 호출하고 보안과 예외 처리 일관성과 관련하여 위에서 설명한 보증 중 어느 것도 이러한 함수에 유효하지 않습니다.

subprocess.**getstatusoutput**(*cmd*, **, *encoding=None*, *errors=None*)

셸에서 *cmd*를 실행한 후 (exitcode, output)을 반환합니다.

Popen.check_output() 으로 셸에서 문자열 *cmd*를 실행하고 2-튜플 (exitcode, output)을 반환합니다. 출력을 디코딩하는 데 *encoding*과 *errors*가 사용됩니다; 자세한 내용은 자주 사용되는 인자에 대한 참고 사항을 참조하십시오.

후행 줄 바꿈이 출력에서 제거됩니다. 명령의 종료 코드는 서브 프로세스의 반환 코드로 해석될 수 있습니다. 예:

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
```

가용성: Unix, Windows.

버전 3.3.4에서 변경: 윈도우 지원이 추가되었습니다.

이 함수는 이제 파이썬 3.3.3과 이전 버전에서와같이 (status, output) 대신 (exitcode, output) 을 반환합니다. exitcode는 `returncode`와 같은 값을 갖습니다.

버전 3.11에서 변경: *encoding*과 *errors* 매개 변수를 추가했습니다.

`subprocess.`**`getoutput`**(*cmd*, **, encoding=None, errors=None*)

셸에서 *cmd*를 실행한 출력(stdout과 stderr)을 반환합니다.

종료 코드를 무시하고 반환 값은 명령의 출력을 포함하는 문자열인 것을 제외하고, `getstatusoutput()` 과 유사합니다. 예:

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

가용성: Unix, Windows.

버전 3.3.4에서 변경: 윈도우 지원이 추가되었습니다

버전 3.11에서 변경: *encoding*과 *errors* 매개 변수를 추가했습니다.

18.7.8 노트

Timeout Behavior

When using the `timeout` parameter in functions like `run()`, `Popen.wait()`, or `Popen.communicate()`, users should be aware of the following behaviors:

1. **Process Creation Delay**: The initial process creation itself cannot be interrupted on many platform APIs. This means that even when specifying a timeout, you are not guaranteed to see a timeout exception until at least after however long process creation takes.
2. **Extremely Small Timeout Values**: Setting very small timeout values (such as a few milliseconds) may result in almost immediate `TimeoutExpired` exceptions because process creation and system scheduling inherently require time.

윈도우에서 인자 시퀀스를 문자열로 변환하기

윈도우에서, *args* 시퀀스는 다음 규칙을 사용하여 구문 분석할 수 있는 문자열로 변환됩니다 (MS C 런타임에서 사용하는 규칙에 해당합니다):

1. 인자는 스페이스나 탭인 공백으로 구분됩니다.
2. 큰따옴표로 묶인 문자열은 포함된 공백과 관계없이 단일 인자로 해석됩니다. 인용된 문자열을 인자에 포함할 수 있습니다.
3. 백 슬래시가 앞에 붙는 큰따옴표는 리터럴 큰따옴표로 해석됩니다.
4. 역 슬래시는 큰따옴표 바로 앞에 오지 않는 한 문자 그대로 해석됩니다.
5. 역 슬래시가 큰따옴표 바로 앞에 있으면, 모든 역 슬래시 쌍은 리터럴 역 슬래시로 해석됩니다. 역 슬래시 수가 홀수이면, 규칙 3에 설명된 대로 마지막 역 슬래시는 다음 큰따옴표를 이스케이프 합니다.

더 보기

shlex
명령 줄을 구문 분석하고 이스케이프 하는 함수를 제공하는 모듈.

Disable use of `posix_spawn()`

On Linux, *subprocess* defaults to using the `vfork()` system call internally when it is safe to do so rather than `fork()`. This greatly improves performance.

```
subprocess._USE_POSIX_SPAWN = False  # See CPython issue gh-NNNNNN.
```

It is safe to set this to false on any Python version. It will have no effect on older or newer versions where unsupported. Do not assume the attribute is available to read. Despite the name, a true value does not indicate the corresponding function will be used, only that it may be.

Please file issues any time you have to use these private knobs with a way to reproduce the issue you were seeing. Link to that issue from a comment in your code.

Added in version 3.8: `_USE_POSIX_SPAWN`

18.8 sched — 이벤트 스케줄러

소스 코드: Lib/sched.py

---

sched 모듈은 범용 이벤트 스케줄러를 구현하는 클래스를 정의합니다:

class sched.**scheduler**(*timefunc=time.monotonic*, *delayfunc=time.sleep*)

scheduler 클래스는 이벤트 스케줄링을 위한 일반적인 인터페이스를 정의합니다. "외부 세계"를 실제로 다루기 위해 두 개의 함수를 요구합니다 — *timefunc*는 인자 없이 호출할 수 있어야 하고, 숫자(단위가 무엇이든, "시간")를 반환합니다. *delayfunc* 함수는 하나의 인자로 호출 가능해야 하며, *timefunc*의 출력과 호환되어야 하고, 그 시간 동안 지연시켜야 합니다. *delayfunc*는 다중 스레드 응용 프로그램에서 다른 스레드가 실행할 기회를 주기 위해 각 이벤트가 실행된 후 `0` 인자로 호출되기도 합니다.

버전 3.3에서 변경: *timefunc* 와 *delayfunc* 매개 변수는 선택적입니다.

버전 3.3에서 변경: *scheduler* 클래스는 다중 스레드 환경에서 안전하게 사용할 수 있습니다.

예제:

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',))
...     # 우선순위가 더 높음에도 불구하고, enter() 가 상대적이라서 'keyword' 는
↪'positional' 다음에 실행됩니다
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.enterabs(1_650_000_000, 10, print_time, argument=("first enterabs",))
...     s.enterabs(1_650_000_000, 5, print_time, argument=("second enterabs",))
...     s.run()
...     print(time.time())
...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> print_some_times()
1652342830.3640375
From print_time 1652342830.3642538 second enterabs
From print_time 1652342830.3643398 first enterabs
From print_time 1652342835.3694863 positional
From print_time 1652342835.3696074 keyword
From print_time 1652342840.369612 default
1652342840.3697174
```

18.8.1 스케줄러 객체

`scheduler` 인스턴스에는 다음과 같은 메서드와 어트리뷰트가 있습니다:

scheduler.**enterabs**(*time*, *priority*, *action*, *argument*=(), *kwargs*={})

새 이벤트를 예약합니다. *time* 인자는 생성자에 전달된 *timefunc* 함수의 반환 값과 호환되는 숫자형이어야 합니다. 같은 *time*으로 예약된 이벤트는 *priority* 순으로 실행됩니다. 낮은 숫자는 높은 우선순위를 나타냅니다.

이벤트를 실행하는 것은 `action(*argument, **kwargs)`를 실행하는 것을 의미합니다. *argument*는 *action*에 대한 위치 인자가 들어있는 시퀀스입니다. *kwargs*는 *action*에 대한 키워드 인자가 들어있는 딕셔너리입니다.

반환 값은 나중에 이벤트를 취소하는 데 사용할 수 있는 이벤트입니다 (`cancel()` 참조).

버전 3.3에서 변경: *argument* 매개 변수는 선택적입니다.

버전 3.3에서 변경: *kwargs* 매개 변수가 추가되었습니다.

scheduler.**enter**(*delay*, *priority*, *action*, *argument*=(), *kwargs*={})

delay 시간 단위 후로 이벤트를 예약합니다. 상대 시간 이외의 다른 인자, 효과 및 반환 값은 `enterabs()`와 같습니다.

버전 3.3에서 변경: *argument* 매개 변수는 선택적입니다.

버전 3.3에서 변경: *kwargs* 매개 변수가 추가되었습니다.

scheduler.**cancel**(*event*)

큐에서 이벤트를 제거합니다. *event*가 현재 큐에 없으면, 이 메서드는 `ValueError`를 발생시킵니다.

scheduler.**empty**()

이벤트 큐가 비어있으면 `True`를 반환합니다.

scheduler.**run**(*blocking*=*True*)

모든 예약된 이벤트를 실행합니다. 이 메서드는 다음 이벤트를 (생성자에 전달된 *delayfunc* 함수를 사용하여) 기다린 다음 예약된 이벤트가 소진될 때까지 계속 실행합니다.

*blocking*이 거짓이면 시간이 도래한 (있다면) 예약된 이벤트를 모두 실행한 다음, 스케줄러에서 다음 예약된 호출까지의 (있다면) 대기시간을 반환합니다.

action 이나 *delayfunc*는 예외를 발생시킬 수 있습니다. 두 경우 모두, 스케줄러는 일관된 상태를 유지하고 예외를 전파합니다. *action*에 의해 예외가 발생하면, 이후에 `run()`을 호출할 때 이벤트를 실행하려고 하지 않습니다.

일련의 이벤트가 다음 이벤트 이전에 사용 가능한 시간보다 실행하는 데 더 오래 걸리면, 스케줄러는 단순히 지연됩니다. 어떤 이벤트도 삭제되지 않습니다; 더는 적절하지 않은 이벤트를 취소할 책임은 호출 코드에 있습니다.

버전 3.3에서 변경: *blocking* 매개 변수가 추가되었습니다.

scheduler.**queue**

남은 이벤트의 리스트를 실행될 순서대로 반환하는 읽기 전용 어트리뷰트입니다. 각 이벤트는 다음과 같은 필드를 갖는 네임드 튜플로 표시됩니다: time, priority, action, argument, kwargs.

18.9 queue — 동기화된 큐 클래스

소스 코드: Lib/queue.py

---

queue 모듈은 다중 생산자, 다중 소비자 큐를 구현합니다. 정보가 여러 스레드 간에 안전하게 교환되어야 할 때 스레드 프로그래밍에서 특히 유용합니다. 이 모듈의 Queue 클래스는 필요한 모든 로킹 개념을 구현합니다.

모듈은 항목을 꺼내는 순서 만 다른 3가지 유형의 큐를 구현합니다. FIFO 큐에서는, 추가된 첫 번째 작업이 처음으로 꺼내지는 작업입니다. LIFO 큐에서는, 가장 최근에 추가된 항목이 처음으로 꺼내지는 항목입니다 (스택처럼 작동합니다). 우선순위 (priority) 큐에서는, 항목이 정렬된 상태로 유지되고 (heapq 모듈을 사용합니다) 가장 낮은 값을 갖는 항목이 먼저 꺼내집니다.

내부적으로, 이러한 3가지 유형의 큐는 록을 사용하여 경쟁 스레드를 일시적으로 블록합니다; 그러나, 스레드 내에서의 재진입을 처리하도록 설계되지는 않았습니다.

또한, 이 모듈은 "간단한" FIFO 큐 유형인 SimpleQueue를 구현합니다. 이 특정 구현은 작은 기능을 포기하는 대신 추가 보장을 제공합니다.

queue 모듈은 다음 클래스와 예외를 정의합니다:

class queue.**Queue**(*maxsize=0*)

FIFO 큐의 생성자. *maxsize*는 큐에 배치할 수 있는 항목 수에 대한 상한을 설정하는 정수입니다. 일단, 이 크기에 도달하면, 큐 항목이 소비될 때까지 삽입이 블록 됩니다. *maxsize*가 0보다 작거나 같으면, 큐 크기는 무한합니다.

class queue.**LifoQueue**(*maxsize=0*)

LIFO 큐의 생성자. *maxsize*는 큐에 배치할 수 있는 항목 수에 대한 상한을 설정하는 정수입니다. 일단, 이 크기에 도달하면, 큐 항목이 소비될 때까지 삽입이 블록 됩니다. *maxsize*가 0보다 작거나 같으면, 큐 크기는 무한합니다.

class queue.**PriorityQueue**(*maxsize=0*)

우선순위 큐의 생성자. *maxsize*는 큐에 배치할 수 있는 항목 수에 대한 상한을 설정하는 정수입니다. 일단, 이 크기에 도달하면, 큐 항목이 소비될 때까지 삽입이 블록 됩니다. *maxsize*가 0보다 작거나 같으면, 큐 크기는 무한합니다.

가장 낮은 값을 갖는 항목이 먼저 꺼내집니다 (가장 낮은 값을 갖는 항목은 `min(entries)` 에 의해 반환될 항목입니다). 항목의 전형적인 패턴은 `(priority_number, data)` 형식의 튜플입니다.

data 요소를 비교할 수 없으면, 데이터는 데이터 항목을 무시하고 우선순위 숫자만 비교하는 클래스로 감쌀 수 있습니다:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

class queue.**SimpleQueue**

상한 없는 FIFO 큐의 생성자. 단순 큐에는 작업 추적과 같은 고급 기능이 없습니다.

Added in version 3.7.

exception queue.**Empty**

비 블로킹 get()(또는 get_nowait())이 비어있는 Queue 객체에 호출될 때 발생하는 예외.

exception queue.**Full**

비 블로킹 put()(또는 put_nowait())이 가득 찬 Queue 객체에 호출될 때 발생하는 예외.

exception queue.**ShutDown**

put()(또는 put())이 종료된 Queue 객체에 호출될 때 발생하는 예외.

Added in version 3.13.

18.9.1 큐 객체

큐 객체(Queue, LifoQueue 또는 PriorityQueue)는 아래에 설명된 공용 메서드를 제공합니다.

Queue.**qsize**()

큐의 대략의 크기를 돌려줍니다. 주의하십시오, qsize() > 0 은 후속 get() 이 블록 되지 않는다는 것을 보장하지 않으며, qsize() < maxsize 도 put() 이 블록 되지 않는다고 보장하지 않습니다.

Queue.**empty**()

큐가 비어 있으면 True를, 그렇지 않으면 False를 반환합니다. empty() 가 True를 반환하면, put() 에 대한 후속 호출이 블록 되지 않는다고 보장하는 것은 아닙니다. 마찬가지로 empty() 가 False를 반환하면, get() 에 대한 후속 호출이 블록 되지 않는다고 보장하는 것은 아닙니다.

Queue.**full**()

큐가 가득 차면 True를, 그렇지 않으면 False를 반환합니다. full() 이 True를 반환하면, get() 에 대한 후속 호출이 블록 되지 않는다고 보장하는 것은 아닙니다. 마찬가지로 full() 이 False를 반환하면, put() 에 대한 후속 호출이 블록 되지 않는다고 보장하는 것은 아닙니다.

Queue.**put**(*item*, *block=True*, *timeout=None*)

큐에 *item*을 넣습니다. 선택적 인자 *block*이 참이고 *timeout*이 None(기본값)이면, 사용 가능한 슬롯이 확보될 때까지 필요하면 블록합니다. *timeout*이 양수면, 최대 *timeout* 초 동안 블록하고 그 시간 내에 사용 가능한 슬롯이 없으면 Full 예외가 발생합니다. 그렇지 않으면 (*block*이 거짓), 빈 슬롯이 즉시 사용할 수 있으면 큐에 항목을 넣고, 그렇지 않으면 Full 예외를 발생시킵니다 (이때 *timeout*은 무시됩니다).

Raises ShutDown if the queue has been shut down.

Queue.**put_nowait**(*item*)

put(item, block=False) 와 동등합니다.

Queue.**get**(*block=True*, *timeout=None*)

큐에서 항목을 제거하고 반환합니다. 선택적 인자 *block*이 참이고 *timeout*이 None(기본값)이면, 항목이 사용 가능할 때까지 필요하면 블록합니다. *timeout*이 양수면, 최대 *timeout* 초 동안 블록하고 그 시간 내에 사용 가능한 항목이 없으면 Empty 예외가 발생합니다. 그렇지 않으면 (*block*이 거짓), 즉시 사용할 수 있는 항목이 있으면 반환하고, 그렇지 않으면 Empty 예외를 발생시킵니다 (이때 *timeout*은 무시됩니다).

POSIX 시스템에서 3.0 이전에서, 윈도우의 모든 버전에서, *block*이 참이고 *timeout*이 None이면, 이 연산은 하부 록에 대한 중단되지 않는(uninterruptible) 대기로 들어갑니다. 이는 어떤 예외도 발생할 수 없음을 뜻하고, 특히 SIGINT가 KeyboardInterrupt를 일으키지 않습니다.

Raises ShutDown if the queue has been shut down and is empty, or if the queue has been shut down immediately.

Queue.**get_nowait**()

get(False) 와 동등합니다.

큐에 넣은 작업이 데몬 소비자 스레드에 의해 완전히 처리되었는지를 추적하는 것을 지원하는 두 가지 메서드가 제공됩니다.

Queue.**task_done**()

앞서 큐에 넣은 작업이 완료되었음을 나타냅니다. 큐 소비자 스레드에서 사용됩니다. 작업을 꺼내는 데 사용되는 get() 마다, 후속 task_done() 호출은 작업에 대한 처리가 완료되었음을 큐에 알려줍니다.

join() 이 현재 블로킹 중이면, 모든 항목이 처리되면 (큐로 put() 된 모든 항목에 대해 task_done() 호출이 수신되었음을 뜻합니다) 재개됩니다.

shutdown(immediate=True) calls task_done() for each remaining item in the queue.

큐에 있는 항목보다 더 많이 호출되면 *ValueError*를 발생시킵니다.

Queue.**join**()

큐의 모든 항목을 꺼내서 처리할 때까지 블록합니다.

완료되지 않은 작업 카운트는 항목이 큐에 추가될 때마다 올라갑니다. 소비자 스레드가 *task_done()*을 호출해서 항목을 꺼내고 작업이 모두 완료되었음을 나타낼 때마다 카운트가 내려갑니다. 완료되지 않은 작업 카운트가 0으로 떨어지면, *join()*이 블록 해제됩니다.

큐에 포함된 작업이 완료될 때까지 대기하는 방법의 예:

```
import threading
import queue

q = queue.Queue()

def worker():
    while True:
        item = q.get()
        print(f'Working on {item}')
        print(f'Finished {item}')
        q.task_done()

# 작업자 스레드를 켭니다.
threading.Thread(target=worker, daemon=True).start()

# 작업자에게 30개의 작업 요청을 보냅니다.
for item in range(30):
    q.put(item)

# 모든 작업이 완료될 때까지 블록합니다.
q.join()
print('All work completed')
```

Terminating queues

Queue objects can be made to prevent further interaction by shutting them down.

Queue.**shutdown**(*immediate=False*)

Shut down the queue, making *get()* and *put()* raise *ShutDown*.

By default, *get()* on a shut down queue will only raise once the queue is empty. Set *immediate* to true to make *get()* raise immediately instead.

All blocked callers of *put()* and *get()* will be unblocked. If *immediate* is true, a task will be marked as done for each remaining item in the queue, which may unblock callers of *join()*.

Added in version 3.13.

18.9.2 SimpleQueue 객체

SimpleQueue 객체는 아래에서 설명하는 공용 메서드를 제공합니다.

SimpleQueue.**qsize**()

큐의 대략의 크기를 돌려줍니다. 주의하십시오, qsize() > 0 은 후속 get() 이 블록 되지 않는다는 것을 보장하지 않습니다.

SimpleQueue.**empty**()

큐가 비어 있으면 `True`를, 그렇지 않으면 `False`를 반환합니다. empty() 가 `False`를 반환하면, get() 에 대한 후속 호출이 블록 되지 않는다는 것을 보장하지는 않습니다.

SimpleQueue.**put**(*item*, *block=True*, *timeout=None*)

*item*을 큐에 넣습니다. 이 메서드는 결코 블록하지 않고 항상 성공합니다 (메모리 할당 실패와 같은 잠재적 저수준 에러 제외). 선택적 인자 *block*과 *timeout*은 무시되고 `Queue.put()`과의 호환성을 위해서만 제공됩니다.

CPython 구현 상세: This method has a C implementation which is reentrant. That is, a `put()` or `get()` call can be interrupted by another `put()` call in the same thread without deadlocking or corrupting internal state inside the queue. This makes it appropriate for use in destructors such as `__del__` methods or `weakref` callbacks.

SimpleQueue.**put_nowait**(*item*)

`put(item, block=False)`과 동등합니다, `Queue.put_nowait()`와의 호환성을 위해 제공됩니다.

SimpleQueue.**get**(*block=True*, *timeout=None*)

큐에서 항목을 제거하고 반환합니다. 선택적 인자 *block*이 참이고 *timeout*이 `None`(기본값)이면, 항목이 사용 가능할 때까지 필요하면 블록합니다. *timeout*이 양수면, 최대 *timeout* 초 동안 블록하고 그 시간 내에 사용 가능한 항목이 없으면 `Empty` 예외가 발생합니다. 그렇지 않으면 (*block*이 거짓), 즉시 사용할 수 있는 항목이 있으면 반환하고, 그렇지 않으면 `Empty` 예외를 발생시킵니다 (이때 *timeout*은 무시됩니다).

SimpleQueue.**get_nowait**()

`get(False)`와 동등합니다.

더 보기

***multiprocessing.Queue* 클래스**
(다중 스레드 대신) 다중 프로세스 문맥에서 사용하기 위한 큐 클래스.

`collections.deque`는 록을 필요로하지 않고 인덱싱을 지원하는 빠른 원자적 `append()`와 `popleft()` 연산을 제공하는 크기 제한 없는 큐의 대체 구현입니다.

18.10 contextvars — 컨텍스트 변수

이 모듈은 컨텍스트-로컬 상태를 관리, 저장, 액세스하기 위한 API를 제공합니다. `ContextVar` 클래스는 컨텍스트 변수 를 선언하고 사용하는 데 쓰입니다. `copy_context()` 함수와 `Context` 클래스는 비동기 프레임워크에서 현재 컨텍스트를 관리하는 데 사용해야 합니다.

상태가 있는 컨텍스트 관리자는 동시성 코드에서 상태가 예기치 않게 다른 코드로 유출되는 것을 방지하기 위해 `threading.local()` 대신 컨텍스트 변수를 사용해야 합니다.

자세한 내용은 **PEP 567**을 참조하십시오.

Added in version 3.7.

18.10.1 컨텍스트 변수

class contextvars.**ContextVar**(*name*[, *, *default*])

이 클래스는 새로운 컨텍스트 변수를 선언하는 데 사용됩니다. 예:

```
var: ContextVar[int] = ContextVar('var', default=42)
```

필수 *name* 매개 변수는 인트로스팩션 및 디버그 목적으로 사용됩니다.

선택적 키워드 전용 *default* 매개 변수는 변수에 대한 값이 현재 컨텍스트에서 발견되지 않으면 `ContextVar.get()` 에 의해 반환됩니다.

중요: 컨텍스트 변수는 최상위 모듈 수준에서 만들어져야 하고 클로저에서 만들어져서는 안 됩니다. `Context` 객체는 컨텍스트 변수에 대해 강한 참조를 유지해서 컨텍스트 변수가 제대로 가비지 수집되지 못하게 합니다.

name

변수의 이름. 읽기 전용 프로퍼티입니다.

Added in version 3.7.1.

get([*default*])

현재 컨텍스트의 컨텍스트 변수에 대한 값을 반환합니다.

현재 컨텍스트에서 변수에 대한 값이 없는 경우 메서드는:

- 제공된 경우 메서드의 *default* 인자 값을 반환합니다; 또는
- 생성 시에 제공된 경우, 컨텍스트 변수의 기본값을 반환합니다; 또는
- `LookupError` 를 발생시킵니다.

set(*value*)

현재 컨텍스트에서 컨텍스트 변수의 새 값을 설정하려면 호출합니다.

필수 *value* 인자는 컨텍스트 변수의 새 값입니다.

`ContextVar.reset()` 메서드를 통해 변수를 이전 값으로 복원하는 데 사용할 수 있는 `Token` 객체를 반환합니다.

reset(*token*)

token 을 생성 한 `ContextVar.set()` 이 사용되기 전의 값으로 컨텍스트 변수를 재설정합니다.

예를 들면:

```
var = ContextVar('var')

token = var.set('new value')
# 'var'를 사용하는 코드; var.get() 은 'new value'를 반환합니다.
var.reset(token)

# reset 호출 후에 var에는 값이 다시 없으므로, var.get() 은
# LookupError를 발생시킵니다.
```

class contextvars.**Token**

Token 객체는 `ContextVar.set()` 메서드에 의해 반환됩니다. `ContextVar.reset()` 메서드에 전달해서 변수의 값을 해당 *set* 이전의 값으로 되돌릴 수 있습니다.

The token supports context manager protocol to restore the corresponding context variable value at the exit from `with` block:

```
var = ContextVar('var', default='default value')

with var.set('new value'):
    assert var.get() == 'new value'

assert var.get() == 'default value'
```

Added in version 3.14: Added support for usage as a context manager.

var

읽기 전용 프로퍼티. 토큰을 생성 한 `ContextVar` 객체를 가리 킵니다.

old_value

읽기 전용 프로퍼티. 토큰을 생성 한 `ContextVar.set()` 메서드 호출 전 변수의 값으로 설정됩니다. 호출 전에 변수가 설정되지 않았으면 `Token.MISSING`을 가리킵니다.

MISSING

`Token.old_value` 에 의해 사용되는 표지 객체.

18.10.2 수동 컨텍스트 관리

contextvars.**copy_context**()

현재 *Context* 객체의 복사본을 반환합니다.

다음 코드 조각은 현재 컨텍스트의 복사본을 가져와서 모든 변수와 그 변수에 설정된 값을 출력합니다:

```
ctx: Contexts = copy_context()
print(list(ctx.items()))
```

이 함수는 $O(1)$ 복잡도를 갖고 있습니다. 즉, 몇 가지 컨텍스트 변수가 있는 컨텍스트와 컨텍스트 변수가 잔뜩 있는 컨텍스트에 대해 똑같이 빠르게 작동합니다.

class contextvars.**Context**

ContextVars 에서 그 값으로의 매핑.

Context() 는 값이 없는 빈 컨텍스트를 만듭니다. 현재 컨텍스트의 복사본을 얻으려면 *copy_context()* 함수를 사용하십시오.

Each thread has its own effective stack of Context objects. The *current context* is the Context object at the top of the current thread' s stack. All Context objects in the stacks are considered to be *entered*.

Entering a context, which can be done by calling its *run()* method, makes the context the current context by pushing it onto the top of the current thread' s context stack.

Exiting from the current context, which can be done by returning from the callback passed to the *run()* method, restores the current context to what it was before the context was entered by popping the context off the top of the context stack.

Since each thread has its own context stack, *ContextVar* objects behave in a similar fashion to *threading.local()* when values are assigned in different threads.

Attempting to enter an already entered context, including contexts entered in other threads, raises a *RuntimeError*.

After exiting a context, it can later be re-entered (from any thread).

Any changes to *ContextVar* values via the *ContextVar.set()* method are recorded in the current context. The *ContextVar.get()* method returns the value associated with the current context. Exiting a context effectively reverts any changes made to context variables while the context was entered (if needed, the values can be restored by re-entering the context).

Context는 *collections.abc.Mapping* 인터페이스를 구현합니다.

run(*callable*, **args*, ***kwargs*)

컨텍스트에 진입하고, callable(*args, **kwargs) 를 실행한 다음, 컨텍스트에서 빠져나옵니다. *callable*의 반환값을 반환하거나, 예외가 발생하면 예외를 전파합니다.

예제:

```
import contextvars

var = contextvars.ContextVar('var')
var.set('spam')
print(var.get())  # 'spam'

ctx = contextvars.copy_context()

def main():
    # 'var' 는 'copy_context()' 와 'ctx.run(main)' 을
    # 호출하기 전에 'spam' 으로 설정되었습니다. 그래서:
    print(var.get())  # 'spam'
    print(ctx[var])  # 'spam'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    var.set('ham')

    # 이제, 'var' 를 'ham' 으로 설정한 후에:
    print(var.get())  # 'ham'
    print(ctx[var])  # 'ham'

# 'main' 함수가 'var' 에 적용한 모든 변경은 'ctx' 에 포함됩니다.
ctx.run(main)

# 'main()' 함수는 'ctx' 컨텍스트에서 실행되었으므로,
# 'var' 에 대한 변경은 'ctx' 에 포함되어 있습니다:
print(ctx[var])  # 'ham'

# 그러나, 'ctx' 외부에서, 'var' 는 여전히 'spam' 으로 설정되어있습니다:
print(var.get())  # 'spam'
```

copy()

컨텍스트 객체의 얕은 복사본을 반환합니다.

var in context

context 에 *var* 의 값이 설정되었으면 `True` 를, 그렇지 않으면 *False*를 반환합니다.

context[var]

var `ContextVar` 변수의 값을 돌려줍니다. 컨텍스트 객체에 변수가 설정되어 있지 않으면 `KeyError` 가 발생합니다.

get(*var*[, *default*])

컨텍스트 객체에 *var* 의 값이 있으면, *var* 의 값을 돌려줍니다. 그렇지 않으면 *default* 를 반환합니다. *default* 가 주어지지 않으면 `None` 을 반환합니다.

iter(context)

컨텍스트 객체에 저장된 변수에 대한 이터레이터를 반환합니다.

len(proxy)

컨텍스트 객체에 설정된 변수의 개수를 반환합니다.

keys()

컨텍스트 객체의 모든 변수 목록을 반환합니다.

values()

컨텍스트 객체의 모든 변수의 값 목록을 반환합니다.

items()

컨텍스트 객체에서 모든 변수와 해당 값을 포함하는 2-튜플의 목록을 반환합니다.

18.10.3 asyncio 지원

컨텍스트 변수는 `asyncio` 에서 기본적으로 지원되며 추가 구성없이 사용할 수 있습니다. 예를 들어, 이것은 컨텍스트 변수를 사용하여, 원격 클라이언트의 주소를 해당 클라이언트를 처리하는 Task에서 사용할 수 있도록 하는 간단한 메아리 서버입니다:

```
import asyncio
import contextvars

client_addr_var = contextvars.ContextVar('client_addr')

def render_goodbye():
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    # 현재 처리중인 클라이언트의 주소를 이 함수에 명시 적으로 전달하지 않고도
    # 액세스 할 수 있습니다.

    client_addr = client_addr_var.get()
    return f'Good bye, client @ {client_addr}\r\n'.encode()

async def handle_request(reader, writer):
    addr = writer.transport.get_extra_info('socket').getpeername()
    client_addr_var.set(addr)

    # 우리가 호출하는 모든 코드에서 이제 'client_addr_var.get()' 을
    # 호출하여 클라이언트의 주소를 가져올 수 있습니다.

    while True:
        line = await reader.readline()
        print(line)
        if not line.strip():
            break

    writer.write(b'HTTP/1.1 200 OK\r\n')  # status line
    writer.write(b'\r\n')  # headers
    writer.write(render_goodbye())  # body
    writer.close()

async def main():
    srv = await asyncio.start_server(
        handle_request, '127.0.0.1', 8081)

    async with srv:
        await srv.serve_forever()

asyncio.run(main())

# 테스트하려면 telnet이나 curl을 사용할 수 있습니다:
#     telnet 127.0.0.1 8081
#     curl 127.0.0.1:8081
```

다음은 위 서비스 중 일부에 대한 지원 모듈입니다:

18.11 _thread — 저수준 스레드 API

이 모듈은 다중 스레드(경량 프로세스 (*light-weight processes*) 나 태스크 (*tasks*) 라고도 합니다)로 작업하는데 필요한 저수준 기본 요소를 제공합니다 — 전역 데이터 공간을 공유하는 여러 개의 제어 스레드를 뜻합니다. 동기화를 위해서, 간단한 록(뮤텍스 (*mutexes*) 나 이진 세마포어 (*binary semaphores*) 라고도 합니다)이 제공됩니다. `threading` 모듈은 이 모듈 위에 구축되어 사용하기 쉬운 고수준의 스레딩 API를 제공합니다.

버전 3.7에서 변경: 이 모듈은 선택 사항이었지만, 이제는 항상 사용할 수 있습니다.

이 모듈은 다음 상수와 함수를 정의합니다:

exception _thread.**error**

스레드 특정 에러에서 발생합니다.

버전 3.3에서 변경: 이것은 이제 내장 `RuntimeError`의 동의어입니다.

_thread.**LockType**

이것은 록 객체의 형입니다.

_thread.**start_new_thread**(*function*, *args*[, *kwargs*])

새 스레드를 시작하고 식별자를 반환합니다. 스레드는 인자 목록 *args*(튜플이어야 합니다)로 함수 *function*을 실행합니다. 선택적 *kwargs* 인자는 키워드 인자 딕셔너리를 지정합니다.

함수가 반환되면, 스레드는 조용히 종료합니다.

함수가 처리되지 않은 예외로 종료되면, 예외를 처리하기 위해 `sys.unraisablehook()`이 호출됩니다. 훅 인자의 *object* 어트리뷰트는 *function*입니다. 기본적으로, 스택 트레이스가 인쇄된 다음 스레드가 종료합니다 (하지만 다른 스레드는 계속 실행됩니다).

함수가 `SystemExit` 예외를 발생시키면, 조용히 무시됩니다.

Raises an *auditing event* `_thread.start_new_thread` with arguments `function`, `args`, `kwargs`.

버전 3.8에서 변경: `sys.unraisablehook()`은 이제 처리되지 않은 예외를 처리하는 데 사용됩니다.

_thread.**interrupt_main**(*signum=signal.SIGINT*, /)

메인 스레드에 도달한 시그널의 효과를 시뮬레이트합니다. 스레드는 이 함수를 사용하여 메인 스레드를 인터럽트 할 수 있지만, 중단이 즉시 일어난다는 보장은 없습니다.

If given, *signum* is the number of the signal to simulate. If *signum* is not given, `signal.SIGINT` is simulated.

주어진 시그널이 파이썬에 의해 처리되지 않으면 (`signal.SIG_DFL`이나 `signal.SIG_IGN`으로 설정됩니다), 이 함수는 아무것도 하지 않습니다.

버전 3.10에서 변경: The *signum* argument is added to customize the signal number.

> **참고**
>
> This does not emit the corresponding signal but schedules a call to the associated handler (if it exists). If you want to truly emit the signal, use `signal.raise_signal()`.

_thread.**exit**()

`SystemExit` 예외를 발생시킵니다. 잡히지 않으면, 스레드가 조용히 종료되도록 합니다.

_thread.**allocate_lock**()

새로운 록 객체를 반환합니다. 록의 메서드는 아래에 설명되어 있습니다. 록은 초기에 잠금 해제되어 있습니다.

_thread.**get_ident**()

현재 스레드의 '스레드 식별자(thread identifier)' 를 반환합니다. 이것은 0이 아닌 정수입니다. 그 값은 직접적인 의미가 없습니다; 이것은 예를 들어 스레드 특정 데이터의 딕셔너리를 인덱싱하는 데 사용되는 매직 쿠키로 사용하려는 의도입니다. 스레드 식별자는 스레드가 종료되고 다른 스레드가 만들어질 때 재활용될 수 있습니다.

_thread.**get_native_id**()

커널에 의해 할당된 현재 스레드의 네이티브 정수 스레드 ID를 반환합니다. 이것은 음수가 아닌 정수입니다. 이 값은 시스템 전반에 걸쳐 이 특정 스레드를 고유하게 식별하는 데 사용될 수 있습니다 (스레드가 종료될 때까지, 그 후에는 값이 OS에 의해 재활용될 수 있습니다).

가용성: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX, DragonFlyBSD, GNU/kFreeBSD.

Added in version 3.8.

버전 3.13에서 변경: Added support for GNU/kFreeBSD.

_thread.**stack_size**([*size*])

새로운 스레드를 만들 때 사용된 스레드의 스택 크기를 반환합니다. 선택적 *size* 인자는 이후에 만들어지는 스레드에 사용할 스택 크기를 지정하며, 0(플랫폼 또는 구성된 기본값을 사용합니다)이나 최소 32,768(32 KiB)의 양의 정숫값이어야 합니다. *size*를 지정하지 않으면 0이 사용됩니다. 스레드 스택 크기 변경이 지원되지 않으면, `RuntimeError`가 발생합니다. 지정된 스택 크기가 유효하지 않으면,

ValueError가 발생하고, 스택 크기는 변경되지 않습니다. 32 KiB는 현재 인터프리터 자체에 충분한 스택 공간을 보장하기 위해 지원되는 최소 스택 크기 값입니다. 일부 플랫폼에서는 스택 크기 값에 특별한 제한이 있을 수 있습니다, 가령 최소 스택 크기 > 32KB를 요구하거나 시스템 메모리 페이지 크기의 배수로 할당하는 것을 요구할 수 있습니다 - 자세한 내용은 플랫폼 설명서를 참조하십시오 (4 KiB 페이지가 일반적입니다; 더 구체적인 정보가 없으면 스택 크기로 4096의 배수를 사용하는 것이 제안된 방법입니다).

가용성: Windows, pthreads.

Unix platforms with POSIX threads support.

`_thread.TIMEOUT_MAX`

Lock.acquire의 *timeout* 매개 변수에 허용되는 최댓값. 이 값보다 큰 timeout을 지정하면 OverflowError가 발생합니다.

Added in version 3.2.

록 객체에는 다음과 같은 메서드가 있습니다:

`lock.acquire(blocking=True, timeout=-1)`

선택적 인자가 아무것도 없으면, 이 메서드는 조건 없이 록을 획득합니다, 필요하면 다른 스레드가 록을 해제할 때까지 대기합니다 (한 번에 하나의 스레드만 록을 획득할 수 있습니다 — 이것이 록의 존재 이유입니다).

blocking 인자가 있으면, 행동은 그 값에 따라 다릅니다: 거짓이면 대기하지 않고 즉시 획득할 수 있을 때만 록이 획득되고, 참이면 위와 같이 록이 조건 없이 획득됩니다.

부동 소수점 *timeout* 인자가 있고 양수이면, 반환하기 전에 대기할 최대 시간을 초로 지정합니다. 음의 *timeout* 인자는 제한 없는 대기를 지정합니다. *blocking*이 거짓이면 *timeout*을 지정할 수 없습니다.

록이 성공적으로 획득되면 반환 값은 `True`이고, 그렇지 않으면 `False`입니다.

버전 3.2에서 변경: *timeout* 매개 변수가 추가되었습니다.

버전 3.2에서 변경: 록 획득은 이제 POSIX의 시그널에 의해 중단될 수 있습니다.

버전 3.14에서 변경: Lock acquires can now be interrupted by signals on Windows.

`lock.release()`

록을 해제합니다. 록은 반드시 이전에 획득된 것이어야 하지만, 반드시 같은 스레드에 의해 획득된 것일 필요는 없습니다.

`lock.locked()`

록의 상태를 반환합니다: 어떤 스레드에 의해 획득되었으면 `True`, 그렇지 않으면 `False`를 반환합니다.

이러한 메서드 외에도, `with` 문을 통해 록 객체를 사용할 수도 있습니다, 예를 들어:

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
```

주의 사항:

- Interrupts always go to the main thread (the KeyboardInterrupt exception will be received by that thread.)
- sys.exit()를 호출하거나 SystemExit 예외를 발생시키는 것은 _thread.exit()를 호출하는 것과 동등합니다.
- 메인 스레드가 종료할 때, 다른 스레드가 살아남는지는 시스템이 정의합니다. 대부분 시스템에서, `try … finally` 절을 실행하거나 객체 파괴자(destructor)를 실행하지 않고 강제 종료됩니다.

CHAPTER 19

네트워킹과 프로세스 간 통신

이 장에서 설명하는 모듈은 네트워킹과 프로세스 간 통신을 위한 메커니즘을 제공합니다.

어떤 모듈은 같은 기계에 있는 두 개의 프로세스에서만 작동합니다 (예를 들어, `signal`과 `mmap`). 다른 모듈은 두 개 이상의 프로세스가 여러 기계에서 통신하는 데 사용할 수 있는 네트워킹 프로토콜을 지원합니다.

이 장에서 설명하는 모듈 목록은 다음과 같습니다:

19.1 asyncio — 비동기 I/O

Hello World!

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

asyncio는 **async/await** 구문을 사용하여 **동시성** 코드를 작성하는 라이브러리입니다.

asyncio는 고성능 네트워크 및 웹 서버, 데이터베이스 연결 라이브러리, 분산 작업 큐 등을 제공하는 여러 파이썬 비동기 프레임워크의 기반으로 사용됩니다.

asyncio는 종종 IO 병목이면서 고수준의 **구조화된** 네트워크 코드에 가장 적합합니다.

더 보기

a-conceptual-overview-of-asyncio
Explanation of the fundamentals of asyncio.

asyncio는 다음과 같은 작업을 위한 **고수준** API 집합을 제공합니다:

- 파이썬 코루틴들을 동시에 실행하고 실행을 완전히 제어할 수 있습니다.
- 네트워크 *IO*와 *IPC*를 수행합니다;
- 자식 프로세스를 제어합니다;
- 큐를 통해 작업을 분산합니다;
- 동시성 코드를 동기화합니다;

또한, 라이브러리와 프레임워크 개발자가 다음과 같은 작업을 할 수 있도록 하는 **저수준** API가 있습니다:

- 네트워킹, 자식 프로세스 실행, *OS* 시그널 처리 등의 비동기 API를 제공하는 이벤트 루프를 만들고 관리합니다.
- 트랜스포트를 사용하여 효율적인 프로토콜을 구현합니다.
- 콜백 기반 라이브러리와 async/await 구문을 사용한 코드 간에 다리를 놓습니다.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

asyncio REPL

You can experiment with an `asyncio` concurrent context in the *REPL*:

```
$ python -m asyncio
asyncio REPL ...
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio
>>> await asyncio.sleep(10, result='hello')
'hello'
```

Raises an *auditing event* `cpython.run_stdin` with no arguments.

버전 3.12.5에서 변경: (also 3.11.10, 3.10.15, 3.9.20, and 3.8.20) Emits audit events.

버전 3.13에서 변경: Uses PyREPL if possible, in which case `PYTHONSTARTUP` is also executed. Emits audit events.

레퍼런스

19.1.1 Runners

Source code: Lib/asyncio/runners.py

This section outlines high-level asyncio primitives to run asyncio code.

They are built on top of an *event loop* with the aim to simplify async code usage for common wide-spread scenarios.

- *Running an asyncio Program*
- *Runner context manager*
- *Handling Keyboard Interruption*

Running an asyncio Program

`asyncio.run`(*coro*, *, *debug=None*, *loop_factory=None*)

Execute *coro* in an asyncio event loop and return the result.

The argument can be any awaitable object.

This function runs the awaitable, taking care of managing the asyncio event loop, *finalizing asynchronous generators*, and closing the executor.

This function cannot be called when another asyncio event loop is running in the same thread.

If *debug* is `True`, the event loop will be run in debug mode. `False` disables debug mode explicitly. `None` is used to respect the global 디버그 모드 settings.

If *loop_factory* is not `None`, it is used to create a new event loop; otherwise `asyncio.new_event_loop()` is used. The loop is closed at the end. This function should be used as a main entry point for asyncio programs, and should ideally only be called once. It is recommended to use *loop_factory* to configure the event loop instead of policies. Passing `asyncio.EventLoop` allows running asyncio without the policy system.

The executor is given a timeout duration of 5 minutes to shutdown. If the executor hasn' t finished within that duration, a warning is emitted and the executor is closed.

Example:

```
async def main():
    await asyncio.sleep(1)
    print('hello')

asyncio.run(main())
```

Added in version 3.7.

버전 3.9에서 변경: Updated to use `loop.shutdown_default_executor()`.

버전 3.10에서 변경: *debug* is `None` by default to respect the global debug mode settings.

버전 3.12에서 변경: Added *loop_factory* parameter.

버전 3.14에서 변경: *coro* can be any awaitable object.

> **참고**
>
> The `asyncio` policy system is deprecated and will be removed in Python 3.16; from there on, an explicit *loop_factory* is needed to configure the event loop.

Runner context manager

class `asyncio.`**`Runner`**(*, *debug=None*, *loop_factory=None*)

A context manager that simplifies *multiple* async function calls in the same context.

Sometimes several top-level async functions should be called in the same *event loop* and `contextvars.Context`.

If *debug* is `True`, the event loop will be run in debug mode. `False` disables debug mode explicitly. `None` is used to respect the global 디버그 모드 settings.

loop_factory could be used for overriding the loop creation. It is the responsibility of the *loop_factory* to set the created loop as the current one. By default `asyncio.new_event_loop()` is used and set as current event loop with `asyncio.set_event_loop()` if *loop_factory* is `None`.

Basically, `asyncio.run()` example can be rewritten with the runner usage:

```
async def main():
    await asyncio.sleep(1)
    print('hello')

with asyncio.Runner() as runner:
    runner.run(main())
```

Added in version 3.11.

run(*coro*, *, *context=None*)

Execute *coro* in the embedded event loop.

The argument can be any awaitable object.

If the argument is a coroutine, it is wrapped in a Task.

An optional keyword-only *context* argument allows specifying a custom `contextvars.Context` for the code to run in. The runner' s default context is used if context is `None`.

Returns the awaitable' s result or raises an exception.

This function cannot be called when another asyncio event loop is running in the same thread.

버전 3.14에서 변경: *coro* can be any awaitable object.

close()

Close the runner.

Finalize asynchronous generators, shutdown default executor, close the event loop and release embedded `contextvars.Context`.

get_loop()

Return the event loop associated with the runner instance.

> **참고**
>
> `Runner` uses the lazy initialization strategy, its constructor doesn' t initialize underlying low-level structures.
>
> Embedded *loop* and *context* are created at the `with` body entering or the first call of `run()` or `get_loop()`.

Handling Keyboard Interruption

Added in version 3.11.

When `signal.SIGINT` is raised by `Ctrl-C`, `KeyboardInterrupt` exception is raised in the main thread by default. However this doesn' t work with `asyncio` because it can interrupt asyncio internals and can hang the program from exiting.

To mitigate this issue, `asyncio` handles `signal.SIGINT` as follows:

1. `asyncio.Runner.run()` installs a custom `signal.SIGINT` handler before any user code is executed and removes it when exiting from the function.
2. The `Runner` creates the main task for the passed coroutine for its execution.
3. When `signal.SIGINT` is raised by `Ctrl-C`, the custom signal handler cancels the main task by calling `asyncio.Task.cancel()` which raises `asyncio.CancelledError` inside the main task. This causes the Python stack to unwind, `try/except` and `try/finally` blocks can be used for resource cleanup. After the main task is cancelled, `asyncio.Runner.run()` raises `KeyboardInterrupt`.
4. A user could write a tight loop which cannot be interrupted by `asyncio.Task.cancel()`, in which case the second following `Ctrl-C` immediately raises the `KeyboardInterrupt` without cancelling the main task.

19.1.2 코루틴과 태스크

이 절에서는 코루틴과 태스크로 작업하기 위한 고급 asyncio API에 관해 설명합니다.

- 코루틴
- 어웨이터블

- 태스크 만들기
- 태스크 취소
- 태스크 그룹
- 잠자기
- 동시에 태스크 실행하기
- *Eager Task Factory*
- 취소로부터 보호하기
- 시간제한
- 대기 프리미티브
- 스레드에서 실행하기
- 다른 스레드에서 예약하기
- 인트로스펙션
- *Task* 객체

코루틴

소스 코드: Lib/asyncio/coroutines.py

---

async/await 문법으로 선언된 코루틴은 asyncio 응용 프로그램을 작성하는 기본 방법입니다. 예를 들어, 다음 코드 조각은 "hello" 를 인쇄하고, 1초 동안 기다린 다음, "world" 를 인쇄합니다:

```
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
...     print('world')

>>> asyncio.run(main())
hello
world
```

단지 코루틴을 호출하는 것으로 실행되도록 예약하는 것은 아닙니다:

```
>>> main()
<coroutine object main at 0x1053bb7c8>
```

코루틴을 실제로 실행하기 위해, asyncio가 다음과 같은 메커니즘을 제공합니다:

- 최상위 진입점 "main()" 함수를 실행하는 `asyncio.run()` 함수 (위의 예를 보세요.)
- 코루틴을 기다리기. 다음 코드 조각은 1초를 기다린 후 "hello" 를 인쇄한 다음 또 2초를 기다린 후 "world" 를 인쇄합니다:

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
async def main():
    print(f"started at {time.strftime('%X')}")

    await say_after(1, 'hello')
    await say_after(2, 'world')

    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

예상 출력:

```
started at 17:13:52
hello
world
finished at 17:13:55
```

- 코루틴을 asyncio 태스크로 동시에 실행하는 *asyncio.create_task()* 함수.

  위의 예를 수정해서 두 개의 `say_after` 코루틴을 동시에 실행해 봅시다:

```
async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world'))

    print(f"started at {time.strftime('%X')}")

    # 두 태스크가 모두 완료할 때까지 기다립니다 (2초 정도
    # 걸려야 합니다.)
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")
```

  예상 출력은 이제 코드 조각이 이전보다 1초 빠르게 실행되었음을 보여줍니다:

```
started at 17:14:32
hello
world
finished at 17:14:34
```

- The *asyncio.TaskGroup* class provides a more modern alternative to *create_task()*. Using this API, the last example becomes:

```
async def main():
    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task(
            say_after(1, 'hello'))

        task2 = tg.create_task(
            say_after(2, 'world'))

        print(f"started at {time.strftime('%X')}")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    # 컨텍스트 매니저를 빠져나갈 때 어웨이트는 묵시적입니다.

    print(f"finished at {time.strftime('%X')}")
```

The timing and output should be the same as for the previous version.

Added in version 3.11: `asyncio.TaskGroup`.

어웨이터블

우리는 객체가 `await` 표현식에서 사용될 수 있을 때 **어웨이터블** 객체라고 말합니다. 많은 asyncio API는 어웨이터블을 받아들이도록 설계되었습니다.

어웨이터블 객체에는 세 가지 주요 유형이 있습니다: **코루틴**, **태스크** 및 **퓨처**.

코루틴

파이썬 코루틴은 어웨이터블이므로 다른 코루틴에서 기다릴 수 있습니다:

```
import asyncio

async def nested():
    return 42

async def main():
    # 단지 "nested()" 를 호출하면 아무 일도 일어나지 않습니다.
    # 코루틴 객체가 생성되었지만 기다리지 않았습니다.
    # 따라서 *전혀 실행되지 않습니다*.
    nested()  # will raise a "RuntimeWarning".

    # 이제 다른 식으로 해봅시다, 기다립니다:
    print(await nested())  # "42" 를 인쇄합니다.

asyncio.run(main())
```

중요

이 설명서에서 "코루틴" 이라는 용어는 두 가지 밀접한 관련 개념에 사용될 수 있습니다:

- 코루틴 함수: `async def` 함수;
- 코루틴 객체: 코루틴 함수를 호출하여 반환된 객체.

태스크

태스크는 코루틴을 동시에 예약하는 데 사용됩니다.

코루틴이 `asyncio.create_task()` 와 같은 함수를 사용하여 태스크로 싸일 때 코루틴은 곧 실행되도록 자동으로 예약됩니다:

```
import asyncio

async def nested():
    return 42

async def main():
    # nested() 가 곧 "main()" 과 동시에 실행되도록
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    # 예약합니다.
    task = asyncio.create_task(nested())

    # "task" 는 이제 "nested()" 를 취소하는 데 사용하거나,
    # 단순히 완료할 때까지 기다릴 수 있습니다:
    await task

asyncio.run(main())
```

퓨처

*Future*는 비동기 연산의 **최종 결과**를 나타내는 특별한 **저수준** 어웨이터블 객체입니다.

Future 객체를 기다릴 때, 그것은 코루틴이 Future가 다른 곳에서 해결될 때까지 기다릴 것을 뜻합니다.

콜백 기반 코드를 async/await와 함께 사용하려면 asyncio의 Future 객체가 필요합니다.

일반적으로 응용 프로그램 수준 코드에서 Future 객체를 만들 **필요는 없습니다**.

때때로 라이브러리와 일부 asyncio API에 의해 노출되는 Future 객체를 기다릴 수 있습니다:

```
async def main():
    await function_that_returns_a_future_object()

    # 이것도 유효합니다:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

Future 객체를 반환하는 저수준 함수의 좋은 예는 `loop.run_in_executor()`입니다.

태스크 만들기

소스 코드: Lib/asyncio/tasks.py

---

asyncio.**create_task**(*coro*, *, *name=None*, *context=None*, *eager_start=None*, ***kwargs*)

coro 코루틴을 `Task`로 감싸고 실행을 예약합니다. Task 객체를 반환합니다.

The full function signature is largely the same as that of the `Task` constructor (or factory) - all of the keyword arguments to this function are passed through to that interface.

An optional keyword-only *context* argument allows specifying a custom `contextvars.Context` for the *coro* to run in. The current context copy is created when no *context* is provided.

An optional keyword-only *eager_start* argument allows specifying if the task should execute eagerly during the call to create_task, or be scheduled later. If *eager_start* is not passed the mode set by `loop.set_task_factory()` will be used.

`get_running_loop()`에 의해 반환된 루프에서 태스크가 실행되고, 현재 스레드에 실행 중인 루프가 없으면 `RuntimeError`가 발생합니다.

> **참고**
>
> `asyncio.TaskGroup.create_task()` is a new alternative leveraging structural concurrency; it allows for waiting for a group of related tasks with strong safety guarantees.

중요

Save a reference to the result of this function, to avoid a task disappearing mid-execution. The event loop only keeps weak references to tasks. A task that isn' t referenced elsewhere may get garbage collected at any time, even before it' s done. For reliable "fire-and-forget" background tasks, gather them in a collection:

```
background_tasks = set()

for i in range(10):
    task = asyncio.create_task(some_coro(param=i))

    # 태스크를 집합에 추가합니다. 이는 강한 참조를 만듭니다.
    background_tasks.add(task)

    # 완료된 태스크에 대한 참조를 영원히 유지하지 않도록 하려면,
    # 각 태스크가 완료 후에 집합에서 자신의 참조를 제거하도록
    # 하십시오:
    task.add_done_callback(background_tasks.discard)
```

Added in version 3.7.

버전 3.8에서 변경: *name* 매개 변수가 추가되었습니다.

버전 3.11에서 변경: *context* 매개 변수가 추가되었습니다.

버전 3.14에서 변경: Added the *eager_start* parameter by passing on all *kwargs*.

태스크 취소

Tasks can easily and safely be cancelled. When a task is cancelled, `asyncio.CancelledError` will be raised in the task at the next opportunity.

It is recommended that coroutines use `try/finally` blocks to robustly perform clean-up logic. In case `asyncio.CancelledError` is explicitly caught, it should generally be propagated when clean-up is complete. `asyncio.CancelledError` directly subclasses `BaseException` so most code will not need to be aware of it.

The asyncio components that enable structured concurrency, like `asyncio.TaskGroup` and `asyncio.timeout()`, are implemented using cancellation internally and might misbehave if a coroutine swallows `asyncio.CancelledError`. Similarly, user code should not generally call `uncancel`. However, in cases when suppressing `asyncio.CancelledError` is truly desired, it is necessary to also call `uncancel()` to completely remove the cancellation state.

태스크 그룹

Task groups combine a task creation API with a convenient and reliable way to wait for all tasks in the group to finish.

class asyncio.**TaskGroup**

An asynchronous context manager holding a group of tasks. Tasks can be added to the group using `create_task()`. All tasks are awaited when the context manager exits.

Added in version 3.11.

create_task(*coro*, *, *name=None*, *context=None*, *eager_start=None*, **kwargs)

Create a task in this task group. The signature matches that of `asyncio.create_task()`. If the task group is inactive (e.g. not yet entered, already finished, or in the process of shutting down), we will close the given `coro`.

버전 3.13에서 변경: Close the given coroutine if the task group is not active.

버전 3.14에서 변경: Passes on all *kwargs* to `loop.create_task()`

예:

```
async def main():
    async with asyncio.TaskGroup() as tg:
        task1 = tg.create_task(some_coro(...))
        task2 = tg.create_task(another_coro(...))
    print(f"Both tasks have completed now: {task1.result()}, {task2.result()}")
```

The `async with` statement will wait for all tasks in the group to finish. While waiting, new tasks may still be added to the group (for example, by passing `tg` into one of the coroutines and calling `tg.create_task()` in that coroutine). Once the last task has finished and the `async with` block is exited, no new tasks may be added to the group.

The first time any of the tasks belonging to the group fails with an exception other than *asyncio.CancelledError*, the remaining tasks in the group are cancelled. No further tasks can then be added to the group. At this point, if the body of the `async with` statement is still active (i.e., `__aexit__()` hasn' t been called yet), the task directly containing the `async with` statement is also cancelled. The resulting *asyncio.CancelledError* will interrupt an `await`, but it will not bubble out of the containing `async with` statement.

Once all tasks have finished, if any tasks have failed with an exception other than *asyncio.CancelledError*, those exceptions are combined in an *ExceptionGroup* or *BaseExceptionGroup* (as appropriate; see their documentation) which is then raised.

Two base exceptions are treated specially: If any task fails with *KeyboardInterrupt* or *SystemExit*, the task group still cancels the remaining tasks and waits for them, but then the initial *KeyboardInterrupt* or *SystemExit* is re-raised instead of *ExceptionGroup* or *BaseExceptionGroup*.

If the body of the `async with` statement exits with an exception (so `__aexit__()` is called with an exception set), this is treated the same as if one of the tasks failed: the remaining tasks are cancelled and then waited for, and non-cancellation exceptions are grouped into an exception group and raised. The exception passed into `__aexit__()`, unless it is *asyncio.CancelledError*, is also included in the exception group. The same special case is made for *KeyboardInterrupt* and *SystemExit* as in the previous paragraph.

Task groups are careful not to mix up the internal cancellation used to "wake up" their `__aexit__()` with cancellation requests for the task in which they are running made by other parties. In particular, when one task group is syntactically nested in another, and both experience an exception in one of their child tasks simultaneously, the inner task group will process its exceptions, and then the outer task group will receive another cancellation and process its own exceptions.

In the case where a task group is cancelled externally and also must raise an *ExceptionGroup*, it will call the parent task' s *cancel()* method. This ensures that a *asyncio.CancelledError* will be raised at the next `await`, so the cancellation is not lost.

Task groups preserve the cancellation count reported by *asyncio.Task.cancelling()*.

버전 3.13에서 변경: Improved handling of simultaneous internal and external cancellations and correct preservation of cancellation counts.

Terminating a Task Group

While terminating a task group is not natively supported by the standard library, termination can be achieved by adding an exception-raising task to the task group and ignoring the raised exception:

```
import asyncio
from asyncio import TaskGroup

class TerminateTaskGroup(Exception):
    """태스크 그룹 종료하려고 발생시키는 예외."""

async def force_terminate_task_group():
    """태스크 그룹의 강제 종료에 사용됩니다."""
    raise TerminateTaskGroup()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
async def job(task_id, sleep_time):
    print(f'Task {task_id}: start')
    await asyncio.sleep(sleep_time)
    print(f'Task {task_id}: done')

async def main():
    try:
        async with TaskGroup() as group:
            # 태스크를 몇 개 만듭니다
            group.create_task(job(1, 0.5))
            group.create_task(job(2, 1.5))
            # 1초 동안 잠잡니다
            await asyncio.sleep(1)
            # 그룹을 강제 종료하기 위해 예외를 일으키는 태스크를 추가합니다
            group.create_task(force_terminate_task_group())
    except* TerminateTaskGroup:
        pass

asyncio.run(main())
```

예상 출력:

```
Task 1: start
Task 2: start
Task 1: done
```

잠자기

async asyncio.**sleep**(*delay*, *result=None*)

delay 초 동안 블록합니다.

*result*가 제공되면, 코루틴이 완료될 때 호출자에게 반환됩니다.

`sleep()`은 항상 현재 태스크를 일시 중단해서 다른 태스크를 실행할 수 있도록 합니다.

Setting the delay to 0 provides an optimized path to allow other tasks to run. This can be used by long-running functions to avoid blocking the event loop for the full duration of the function call.

5초 동안 현재 날짜를 매초 표시하는 코루틴의 예:

```
import asyncio
import datetime

async def display_date():
    loop = asyncio.get_running_loop()
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

asyncio.run(display_date())
```

버전 3.10에서 변경: *loop* 매개 변수를 제거했습니다.

버전 3.13에서 변경: Raises `ValueError` if *delay* is `nan`.

동시에 태스크 실행하기

awaitable asyncio.**gather**(**aws, return_exceptions=False*)

aws 시퀀스에 있는 어웨이터블 객체를 동시에 실행합니다.

*aws*에 있는 어웨이터블이 코루틴이면 자동으로 태스크로 예약됩니다.

모든 어웨이터블이 성공적으로 완료되면, 결과는 반환된 값들이 합쳐진 리스트입니다. 결괏값의 순서는 *aws*에 있는 어웨이터블의 순서와 일치합니다.

*return_exceptions*가 `False`(기본값)면, 첫 번째 발생한 예외가 `gather()`를 기다리는 태스크로 즉시 전파됩니다. *aws* 시퀀스의 다른 어웨이터블은 **취소되지 않고** 계속 실행됩니다.

*return_exceptions*가 `True`면, 예외는 성공적인 결과처럼 처리되고, 결과 리스트에 집계됩니다.

`gather()`가 취소되면, 모든 제출된 (아직 완료되지 않은) 어웨이터블도 취소됩니다.

aws 시퀀스의 Task나 Future가 취소되면, 그것이 `CancelledError`를 일으킨 것처럼 처리됩니다 – 이때 `gather()` 호출은 취소되지 **않습니다**. 이것은 제출된 태스크/퓨처 하나를 취소하는 것이 다른 태스크/퓨처를 취소하게 되는 것을 막기 위한 것입니다.

> **참고**
>
> A new alternative to create and run tasks concurrently and wait for their completion is `asyncio.TaskGroup`. *TaskGroup* provides stronger safety guarantees than *gather* for scheduling a nesting of subtasks: if a task (or a subtask, a task scheduled by a task) raises an exception, *TaskGroup* will, while *gather* will not, cancel the remaining scheduled tasks).

예:

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({number}), currently i={i}...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")
    return f

async def main():
    # 3개의 호출을 *동시에* 예약합니다:
    L = await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )
    print(L)

asyncio.run(main())

# 예상 출력:
#
#     Task A: Compute factorial(2), currently i=2...
#     Task B: Compute factorial(3), currently i=2...
#     Task C: Compute factorial(4), currently i=2...
#     Task A: factorial(2) = 2
#     Task B: Compute factorial(3), currently i=3...
#     Task C: Compute factorial(4), currently i=3...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
#     Task B: factorial(3) = 6
#     Task C: Compute factorial(4), currently i=4...
#     Task C: factorial(4) = 24
#     [2, 6, 24]
```

> **참고**
>
> *return_exceptions*가 거짓이면, 완료로 표시된 후 gather()를 취소하는 것은 제출된 어웨이터블을 취소하지 않습니다. 예를 들어, 예외를 호출자에게 전파한 후 gather가 완료된 것으로 표시될 수 있습니다, 따라서 gather에서 (어웨이터블 중 하나에 의해 발생한) 예외를 포착한 후 `gather.cancel()`을 호출하는 것은 다른 어웨이터블을 취소하지 않습니다.

버전 3.7에서 변경: *gather* 자체가 취소되면, *return_exceptions*와 관계없이 취소가 전파됩니다.

버전 3.10에서 변경: *loop* 매개 변수를 제거했습니다.

버전 3.10부터 폐지됨: Deprecation warning is emitted if no positional arguments are provided or not all positional arguments are Future-like objects and there is no running event loop.

Eager Task Factory

`asyncio.`**`eager_task_factory`**(*loop*, *coro*, ***, *name=None*, *context=None*)

A task factory for eager task execution.

When using this factory (via `loop.set_task_factory(asyncio.eager_task_factory)`), coroutines begin execution synchronously during `Task` construction. Tasks are only scheduled on the event loop if they block. This can be a performance improvement as the overhead of loop scheduling is avoided for coroutines that complete synchronously.

A common example where this is beneficial is coroutines which employ caching or memoization to avoid actual I/O when possible.

> **참고**
>
> Immediate execution of the coroutine is a semantic change. If the coroutine returns or raises, the task is never scheduled to the event loop. If the coroutine execution blocks, the task is scheduled to the event loop. This change may introduce behavior changes to existing applications. For example, the application's task execution order is likely to change.

Added in version 3.12.

`asyncio.`**`create_eager_task_factory`**(*custom_task_constructor*)

Create an eager task factory, similar to `eager_task_factory()`, using the provided *custom_task_constructor* when creating a new task instead of the default `Task`.

custom_task_constructor must be a *callable* with the signature matching the signature of `Task.__init__`. The callable must return a `asyncio.Task`-compatible object.

This function returns a *callable* intended to be used as a task factory of an event loop via `loop.set_task_factory(factory)`).

Added in version 3.12.

취소로부터 보호하기

`awaitable` `asyncio.`**`shield`**(*aw*)

어웨이터블 객체를 취소로부터 보호합니다.

*aw*가 코루틴이면 자동으로 태스크로 예약됩니다.

다음 문장:

```
task = asyncio.create_task(something())
res = await shield(task)
```

은 다음과 동등합니다:

```
res = await something()
```

단, 그것을 포함하는 코루틴이 취소되면, `something()` 에서 실행 중인 태스크는 취소되지 않는다는 것만 예외입니다. `something()` 의 관점에서는, 취소가 일어나지 않았습니다. 호출자는 여전히 취소되었고, "await" 표현식은 여전히 `CancelledError`를 발생시킵니다.

`something()` 가 다른 수단(즉, 그 안에서 스스로)에 의해 취소되면, `shield()` 도 취소됩니다.

취소를 완전히 무시하려면(권장되지 않습니다), 다음과 같이 `shield()` 함수를 try/except 절과 결합해야 합니다:

```
task = asyncio.create_task(something())
try:
    res = await shield(task)
except CancelledError:
    res = None
```

> **중요**
>
> Save a reference to tasks passed to this function, to avoid a task disappearing mid-execution. The event loop only keeps weak references to tasks. A task that isn' t referenced elsewhere may get garbage collected at any time, even before it' s done.

버전 3.10에서 변경: *loop* 매개 변수를 제거했습니다.

버전 3.10부터 폐지됨: Deprecation warning is emitted if *aw* is not Future-like object and there is no running event loop.

시간제한

`asyncio.`**`timeout`**(*delay*)

Return an asynchronous context manager that can be used to limit the amount of time spent waiting on something.

*delay*는 `None` 또는 대기할 float/int 초 수입니다. *delay*가 `None` 이면, 시간 제한이 적용되지 않습니다; 컨텍스트 관리자가 만들어질 때 지연 시간이 아려지지 않은 경우 유용할 수 있습니다.

In either case, the context manager can be rescheduled after creation using `Timeout.reschedule()`.

예:

```
async def main():
    async with asyncio.timeout(10):
        await long_running_task()
```

If `long_running_task` takes more than 10 seconds to complete, the context manager will cancel the current task and handle the resulting `asyncio.CancelledError` internally, transforming it into a `TimeoutError` which can be caught and handled.

> **참고**
>
> The `asyncio.timeout()` context manager is what transforms the `asyncio.CancelledError` into a `TimeoutError`, which means the `TimeoutError` can only be caught *outside* of the context manager.

Example of catching `TimeoutError`:

```
async def main():
    try:
        async with asyncio.timeout(10):
            await long_running_task()
    except TimeoutError:
        print("The long operation timed out, but we've handled it.")

    print("This statement will run regardless.")
```

The context manager produced by `asyncio.timeout()` can be rescheduled to a different deadline and inspected.

class asyncio.**Timeout**(*when*)

An asynchronous context manager for cancelling overdue coroutines.

`when` should be an absolute time at which the context should time out, as measured by the event loop' s clock:

- If `when` is `None`, the timeout will never trigger.
- If `when < loop.time()`, the timeout will trigger on the next iteration of the event loop.

when() → *float* | *None*

Return the current deadline, or `None` if the current deadline is not set.

reschedule(*when:* float | None)

Reschedule the timeout.

expired() → *bool*

Return whether the context manager has exceeded its deadline (expired).

예:

```
async def main():
    try:
        # 시작 시 시간 제한을 알 수 없어서 ``None`` 을 전달합니다.
        async with asyncio.timeout(None) as cm:
            # 이제 시간 제한을 알게되어, 재조정합니다.
            new_deadline = get_running_loop().time() + 10
            cm.reschedule(new_deadline)

            await long_running_task()
    except TimeoutError:
        pass

    if cm.expired():
        print("Looks like we haven't finished on time.")
```

Timeout context managers can be safely nested.

Added in version 3.11.

asyncio.**timeout_at**(*when*)

Similar to `asyncio.timeout()`, except *when* is the absolute time to stop waiting, or `None`.

예:

```
async def main():
    loop = get_running_loop()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    deadline = loop.time() + 20
    try:
        async with asyncio.timeout_at(deadline):
            await long_running_task()
    except TimeoutError:
        print("The long operation timed out, but we've handled it.")

    print("This statement will run regardless.")
```

Added in version 3.11.

async asyncio.**wait_for**(*aw*, *timeout*)

aw 어웨이터블이 제한된 시간 내에 완료될 때까지 기다립니다.

*aw*가 코루틴이면 자동으로 태스크로 예약됩니다.

*timeout*은 `None` 또는 대기할 float 나 int 초 수입니다. *timeout*이 `None`이면 퓨처가 완료될 때까지 블록합니다.

시간 초과가 발생하면, 태스크를 취소하고 `TimeoutError`를 발생시킵니다.

태스크 취소를 피하려면, `shield()`로 감싸십시오.

이 함수는 퓨처가 실제로 취소될 때까지 대기하므로, 총 대기 시간이 *timeout*을 초과할 수 있습니다. 취소하는 동안 예외가 발생하면, 전파됩니다.

대기가 취소되면, 퓨처 *aw*도 취소됩니다.

예:

```
async def eternity():
    # 1시간 동안 잠잡니다
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # 최대 1초간 대기합니다
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except TimeoutError:
        print('timeout!')

asyncio.run(main())

# 예상 출력:
#
#     timeout!
```

버전 3.7에서 변경: 시간 초과로 인해 *aw*가 취소되면, `wait_for`는 *aw*가 취소될 때까지 대기합니다. 이전에는 `TimeoutError`가 즉시 발생했습니다.

버전 3.10에서 변경: *loop* 매개 변수를 제거했습니다.

버전 3.11에서 변경: Raises `TimeoutError` instead of `asyncio.TimeoutError`.

대기 프리미티브

async asyncio.**wait**(*aws*, *, *timeout=None*, *return_when=ALL_COMPLETED*)

aws 이터러블에 있는 `Future`와 `Task` 인스턴스를 동시에 실행하고, *return_when*에 의해 지정된 조건을 만족할 때까지 블록합니다.

aws 이터러블은 비어있을 수 없습니다.

두 집합의 태스크/퓨처를 반환합니다: `(done, pending)`.

사용법:

```
done, pending = await asyncio.wait(aws)
```

timeout(float 나 int) 을 지정하면, 반환하기 전에 대기할 최대 시간(초)을 제어할 수 있습니다.

이 함수는 *TimeoutError*를 발생시키지 않음에 유의하십시오. 시간 초과가 발생할 때 완료되지 않은 퓨처나 태스크는 단순히 두 번째 집합으로 반환됩니다.

*return_when*는 이 함수가 언제 반환해야 하는지 나타냅니다. 다음 상수 중 하나여야 합니다:

상수	설명
`asyncio.FIRST_COMPLETED`	퓨처가 하나라도 끝나거나 취소될 때 함수가 반환됩니다.
`asyncio.FIRST_EXCEPTION`	퓨처가 하나라도 예외를 일으켜 끝나면 함수가 반환됩니다. 어떤 퓨처도 예외를 일으키지 않으면 *ALL_COMPLETED*와 같습니다.
`asyncio.ALL_COMPLETED`	모든 퓨처가 끝나거나 취소되면 함수가 반환됩니다.

*wait_for()*와 달리, `wait()` 는 시간 초과가 발생할 때 퓨처를 취소하지 않습니다.

버전 3.10에서 변경: *loop* 매개 변수를 제거했습니다.

버전 3.11에서 변경: 코루틴 객체를 `wait()` 로 직접 전달하는 것은 금지됩니다.

버전 3.12에서 변경: Added support for generators yielding tasks.

`asyncio.`**`as_completed`**(*aws*, *, *timeout=None*)

aws 이터러블에 있는 어웨이터블 객체를 동시에 실행합니다. 코루틴의 이터레이터를 반환합니다. 반환된 객체는 어웨이터블이 완료될 때마다 그 결과를 얻기 위해 이터레이트 할 수 있습니다.

The object returned by `as_completed()` can be iterated as an *asynchronous iterator* or a plain *iterator*. When asynchronous iteration is used, the originally-supplied awaitables are yielded if they are tasks or futures. This makes it easy to correlate previously-scheduled tasks with their results. Example:

```
ipv4_connect = create_task(open_connection("127.0.0.1", 80))
ipv6_connect = create_task(open_connection("::1", 80))
tasks = [ipv4_connect, ipv6_connect]

async for earliest_connect in as_completed(tasks):
    # earliest_connect 가 완료했습니다. 결과는 어웨이트하거나
    # earliest_connect.result() 를 호출해서 얻을 수 있습니다.
    reader, writer = await earliest_connect

    if earliest_connect is ipv6_connect:
        print("IPv6 connection established.")
    else:
        print("IPv4 connection established.")
```

During asynchronous iteration, implicitly-created tasks will be yielded for supplied awaitables that aren' t tasks or futures.

When used as a plain iterator, each iteration yields a new coroutine that returns the result or raises the exception of the next completed awaitable. This pattern is compatible with Python versions older than 3.13:

```
ipv4_connect = create_task(open_connection("127.0.0.1", 80))
ipv6_connect = create_task(open_connection("::1", 80))
tasks = [ipv4_connect, ipv6_connect]

for next_connect in as_completed(tasks):
    # next_connect 는 원래 태스크 객체 중 하나가 아닙니다.
    # 결과 값을 얻거나 다음에 완료되는 어웨이터블의 예외를 발생시키려면
    # 반드시 어웨이트해야 합니다.
    reader, writer = await next_connect
```

A `TimeoutError` is raised if the timeout occurs before all awaitables are done. This is raised by the `async for` loop during asynchronous iteration or by the coroutines yielded during plain iteration.

버전 3.10에서 변경: *loop* 매개 변수를 제거했습니다.

버전 3.10부터 폐지됨: Deprecation warning is emitted if not all awaitable objects in the *aws* iterable are Future-like objects and there is no running event loop.

버전 3.12에서 변경: Added support for generators yielding tasks.

버전 3.13에서 변경: The result can now be used as either an *asynchronous iterator* or as a plain *iterator* (previously it was only a plain iterator).

스레드에서 실행하기

async asyncio.**to_thread**(*func*, /, **args*, ***kwargs*)

별도의 스레드에서 *func* 함수를 비동기적으로 실행합니다.

이 함수에 제공된 모든 *args 와 **kwargs 는 *func*로 직접 전달됩니다. 또한, 현재 `contextvars.Context`가 전파되어, 이벤트 루프 스레드의 컨텍스트 변수가 별도의 스레드에서 액세스 될 수 있습니다.

*func*의 최종 결과를 얻기 위해 어웨이트 할 수 있는 코루틴을 반환합니다.

이 코루틴 함수는 메인 스레드에서 실행된다면 이벤트 루프를 블록할 IO 병목 함수/메서드를 실행하는 데 주로 사용됩니다. 예를 들면:

```
def blocking_io():
    print(f"start blocking_io at {time.strftime('%X')}")
    # time.sleep()은 파일 연산과 같은 임의의 블로킹 IO 병목 연산으로 대체될
    # 수 있음에 유의하십시오.
    time.sleep(1)
    print(f"blocking_io complete at {time.strftime('%X')}")

async def main():
    print(f"started main at {time.strftime('%X')}")

    await asyncio.gather(
        asyncio.to_thread(blocking_io),
        asyncio.sleep(1))

    print(f"finished main at {time.strftime('%X')}")


asyncio.run(main())

# 예상 출력:
#
# started main at 19:50:53
# start blocking_io at 19:50:53
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# blocking_io complete at 19:50:54
# finished main at 19:50:54
```

코루틴에서 `blocking_io()` 를 직접 호출하면 그동안 이벤트 루프가 블록 되어 추가 1초의 실행 시간이 발생합니다. 대신, `asyncio.to_thread()` 를 사용하면, 이벤트 루프를 블록하지 않고 별도의 스레드에서 실행할 수 있습니다.

> **참고**
>
> *GIL*로 인해, `asyncio.to_thread()` 는 일반적으로 IO 병목 함수를 비 블로킹으로 만드는 데만 사용할 수 있습니다. 그러나, GIL을 반납하는 확장 모듈이나 GIL이 없는 대체 파이썬 구현의 경우, `asyncio.to_thread()` 를 CPU 병목 함수에도 사용할 수 있습니다.

Added in version 3.9.

다른 스레드에서 예약하기

`asyncio.`**`run_coroutine_threadsafe`**(*coro*, *loop*)

주어진 이벤트 루프에 코루틴을 제출합니다. 스레드 안전합니다.

다른 OS 스레드에서 결과를 기다리는 *`concurrent.futures.Future`*를 반환합니다.

이 함수는 이벤트 루프가 실행 중인 스레드가 아닌, 다른 OS 스레드에서 호출하기 위한 것입니다. 예:

```
def in_thread(loop: asyncio.AbstractEventLoop) -> None:
    # Run some blocking IO
    pathlib.Path("example.txt").write_text("hello world", encoding="utf8")

    # Create a coroutine
    coro = asyncio.sleep(1, result=3)

    # Submit the coroutine to a given loop
    future = asyncio.run_coroutine_threadsafe(coro, loop)

    # Wait for the result with an optional timeout argument
    assert future.result(timeout=2) == 3

async def amain() -> None:
    # Get the running loop
    loop = asyncio.get_running_loop()

    # Run something in a thread
    await asyncio.to_thread(in_thread, loop)
```

It' s also possible to run the other way around. Example:

```
@contextlib.contextmanager
def loop_in_thread() -> Generator[asyncio.AbstractEventLoop]:
    loop_fut = concurrent.futures.Future[asyncio.AbstractEventLoop]()
    stop_event = asyncio.Event()

    async def main() -> None:
        loop_fut.set_result(asyncio.get_running_loop())
        await stop_event.wait()

    with concurrent.futures.ThreadPoolExecutor(1) as tpe:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        complete_fut = tpe.submit(asyncio.run, main())
        for fut in concurrent.futures.as_completed((loop_fut, complete_fut)):
            if fut is loop_fut:
                loop = loop_fut.result()
                try:
                    yield loop
                finally:
                    loop.call_soon_threadsafe(stop_event.set)
            else:
                fut.result()

# Create a loop in another thread
with loop_in_thread() as loop:
    # Create a coroutine
    coro = asyncio.sleep(1, result=3)

    # Submit the coroutine to a given loop
    future = asyncio.run_coroutine_threadsafe(coro, loop)

    # Wait for the result with an optional timeout argument
    assert future.result(timeout=2) == 3
```

코루틴에서 예외가 발생하면, 반환된 Future에 통지됩니다. 또한, 이벤트 루프에서 태스크를 취소하는 데 사용할 수 있습니다:

```
try:
    result = future.result(timeout)
except TimeoutError:
    print('The coroutine took too long, cancelling the task...')
    future.cancel()
except Exception as exc:
    print(f'The coroutine raised an exception: {exc!r}')
else:
    print(f'The coroutine returned: {result!r}')
```

설명서의 동시성과 다중 스레드 절을 참조하십시오.

다른 asyncio 함수와 달리, 이 함수는 *loop* 인자가 명시적으로 전달되어야 합니다.

Added in version 3.5.1.

인트로스펙션

`asyncio.`**`current_task`**(*loop=None*)

현재 실행 중인 `Task` 인스턴스를 반환하거나 태스크가 실행되고 있지 않으면 `None`을 반환합니다.

*loop*가 `None`이면, 현재 루프를 가져오는 데 `get_running_loop()`가 사용됩니다.

Added in version 3.7.

`asyncio.`**`all_tasks`**(*loop=None*)

루프에 의해 실행되는 아직 완료되지 않은 `Task` 객체 집합을 반환합니다.

*loop*가 `None`이면, 현재 루프를 가져오는 데 `get_running_loop()`가 사용됩니다.

Added in version 3.7.

`asyncio.`**`iscoroutine`**(*obj*)

*obj*가 코루틴 객체면 `True`를 반환합니다.

Added in version 3.4.

Task 객체

class asyncio.**Task**(*coro*, **, loop=None, name=None, context=None, eager_start=False*)

파이썬 코루틴을 실행하는 퓨처류 객체입니다. 스레드 안전하지 않습니다.

태스크는 이벤트 루프에서 코루틴을 실행하는 데 사용됩니다. 만약 코루틴이 Future를 기다리고 있다면, 태스크는 코루틴의 실행을 일시 중지하고 Future의 완료를 기다립니다. 퓨처가 완료되면, 감싸진 코루틴의 실행이 다시 시작됩니다.

이벤트 루프는 협업 스케줄링을 사용합니다: 이벤트 루프는 한 번에 하나의 Task를 실행합니다. Task가 Future의 완료를 기다리는 동안, 이벤트 루프는 다른 태스크, 콜백을 실행하거나 IO 연산을 수행합니다.

테스크를 만들려면 고수준 `asyncio.create_task()` 함수를 사용하거나, 저수준 `loop.create_task()` 나 `ensure_future()` 함수를 사용하십시오. 태스크의 인스턴스를 직접 만드는 것은 권장되지 않습니다.

실행 중인 Task를 취소하려면 `cancel()` 메서드를 사용하십시오. 이를 호출하면 태스크가 감싼 코루틴으로 `CancelledError` 예외를 던집니다. 코루틴이 취소 중에 Future 객체를 기다리고 있으면, Future 객체가 취소됩니다.

`cancelled()`는 태스크가 취소되었는지 확인하는 데 사용할 수 있습니다. 이 메서드는 감싼 코루틴이 `CancelledError` 예외를 억제하지 않고 실제로 취소되었으면 `True`를 반환합니다.

`asyncio.Task`는 `Future.set_result()`와 `Future.set_exception()`을 제외한 모든 API를 `Future`에서 상속받습니다.

An optional keyword-only *context* argument allows specifying a custom `contextvars.Context` for the *coro* to run in. If no *context* is provided, the Task copies the current context and later runs its coroutine in the copied context.

An optional keyword-only *eager_start* argument allows eagerly starting the execution of the `asyncio.Task` at task creation time. If set to `True` and the event loop is running, the task will start executing the coroutine immediately, until the first time the coroutine blocks. If the coroutine returns or raises without blocking, the task will be finished eagerly and will skip scheduling to the event loop.

버전 3.7에서 변경: `contextvars` 모듈에 대한 지원이 추가되었습니다.

버전 3.8에서 변경: *name* 매개 변수가 추가되었습니다.

버전 3.10부터 폐지됨: Deprecation warning is emitted if *loop* is not specified and there is no running event loop.

버전 3.11에서 변경: *context* 매개 변수가 추가되었습니다.

버전 3.12에서 변경: *eager_start* 매개 변수가 추가되었습니다.

done()

Task가 완료(*done*)되었으면 `True`를 반환합니다.

감싼 코루틴이 값을 반환하거나 예외를 일으키거나, Task가 취소되면 Task는 완료(*done*)됩니다.

result()

Task의 결과를 반환합니다.

Task가 완료(*done*)되었으면 감싼 코루틴의 결과가 반환됩니다 (또는 코루틴이 예외를 발생시켰으면 해당 예외가 다시 발생합니다).

태스크가 취소(*cancelled*)되었으면, 이 메서드는 `CancelledError` 예외를 발생시킵니다.

태스크 결과를 아직 사용할 수 없으면, 이 메서드는 `InvalidStateError` 예외를 발생시킵니다.

exception()

Task의 예외를 반환합니다.

감싼 코루틴이 예외를 발생시키면, 그 예외가 반환됩니다. 감싼 코루틴이 정상적으로 반환되면, 이 메서드는 `None`을 반환합니다.

태스크가 취소(*cancelled*)되었으면, 이 메서드는 `CancelledError` 예외를 발생시킵니다.

태스크가 아직 완료(*done*)되지 않았으면, 이 메서드는 `InvalidStateError` 예외를 발생시킵니다.

add_done_callback(*callback*, *, *context=None*)

태스크가 완료(*done*)될 때 실행할 콜백을 추가합니다.

이 메서드는 저수준 콜백 기반 코드에서만 사용해야 합니다.

자세한 내용은 `Future.add_done_callback()` 설명서를 참조하십시오.

remove_done_callback(*callback*)

콜백 목록에서 *callback*을 제거합니다.

이 메서드는 저수준 콜백 기반 코드에서만 사용해야 합니다.

자세한 내용은 `Future.remove_done_callback()` 설명서를 참조하십시오.

get_stack(*, *limit=None*)

이 Task의 스택 프레임 리스트를 돌려줍니다.

감싼 코루틴이 완료되지 않았으면, 일시 정지된 곳의 스택을 반환합니다. 코루틴이 성공적으로 완료되었거나 취소되었으면 빈 리스트가 반환됩니다. 코루틴이 예외로 종료되었으면, 이것은 트레이스백 프레임의 리스트를 반환합니다.

프레임은 항상 가장 오래된 것부터 순서대로 정렬됩니다.

일시 정지된 코루틴에서는 하나의 스택 프레임만 반환됩니다.

선택적 *limit* 인자는 반환할 최대 프레임 수를 설정합니다; 기본적으로 사용 가능한 모든 프레임이 반환됩니다. 반환되는 리스트의 순서는 스택과 트레이스백 중 어느 것이 반환되는지에 따라 다릅니다: 스택은 최신 프레임이 반환되지만, 트레이스백은 가장 오래된 프레임이 반환됩니다. (이는 traceback 모듈의 동작과 일치합니다.)

print_stack(*, *limit=None*, *file=None*)

이 Task의 스택이나 트레이스백을 인쇄합니다.

이것은 `get_stack()`으로 얻은 프레임에 대해 traceback 모듈과 유사한 출력을 생성합니다.

limit 인자는 `get_stack()`에 직접 전달됩니다.

file 인자는 출력이 기록되는 I/O 스트림입니다; 기본적으로 출력은 `sys.stdout`에 기록됩니다.

get_coro()

`Task`로 싸인 코루틴 객체를 반환합니다.

> **참고**
>
> This will return `None` for Tasks which have already completed eagerly. See the *Eager Task Factory*.

Added in version 3.8.

버전 3.12에서 변경: Newly added eager task execution means result may be `None`.

get_context()

Return the `contextvars.Context` object associated with the task.

Added in version 3.12.

get_name()

Task의 이름을 반환합니다.

Task에 명시적으로 이름이 지정되지 않으면, 기본 asyncio Task 구현은 인스턴스화 중에 기본 이름을 생성합니다.

Added in version 3.8.

set_name(*value*)

Task의 이름을 설정합니다.

value 인자는 모든 객체가 될 수 있으며, 문자열로 변환됩니다.

기본 Task 구현에서, 이름은 태스크 객체의 `repr()` 출력에 표시됩니다.

Added in version 3.8.

cancel(*msg=None*)

Task 취소를 요청합니다.

If the Task is already *done* or *cancelled*, return `False`, otherwise, return `True`.

이 메서드는 이벤트 루프의 다음 사이클에서 감싼 코루틴으로 `CancelledError` 예외를 던져넣도록 합니다.

그러면 코루틴은 `try` … … `except CancelledError` … `finally` 블록으로 정리하거나 예외를 억제하여 요청을 거부할 수 있습니다. 따라서, `Future.cancel()`와 달리 `Task.cancel()`은 Task가 취소됨을 보장하지는 않습니다. 하지만 취소를 완전히 억제하는 것은 일반적이지 않고, 그렇게 하지 말도록 적극적으로 권합니다. 그럼에도 불구하고 코루틴이 취소를 억제하기로 결정한 경우, 예외를 잡는 것 외에 `Task.uncancel()` 호출도 해야 합니다.

버전 3.9에서 변경: *msg* 매개 변수가 추가되었습니다.

버전 3.11에서 변경: The `msg` parameter is propagated from cancelled task to its awaiter. 다음 예는 코루틴이 취소 요청을 가로채는 방법을 보여줍니다:

```
async def cancel_me():
    print('cancel_me(): before sleep')

    try:
        # 1시간 동안 기다립니다
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('cancel_me(): cancel sleep')
        raise
    finally:
        print('cancel_me(): after sleep')

async def main():
    # "cancel_me" Task를 만듭니다
    task = asyncio.create_task(cancel_me())

    # 1초 동안 기다립니다
    await asyncio.sleep(1)

    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("main(): cancel_me is cancelled now")

asyncio.run(main())

# 예상 출력:
#
#     cancel_me(): before sleep
#     cancel_me(): cancel sleep
#     cancel_me(): after sleep
#     main(): cancel_me is cancelled now
```

cancelled()

Task가 취소(*cancelled*)되었으면 `True`를 반환합니다.

Task는 `cancel()`로 취소가 요청되고 감싼 코루틴이 자신에게 전달된 `CancelledError` 예외를 확산할 때 최소(*cancelled*)됩니다.

uncancel()

Decrement the count of cancellation requests to this Task.

Returns the remaining number of cancellation requests.

Note that once execution of a cancelled task completed, further calls to `uncancel()` are ineffective.

Added in version 3.11.

This method is used by asyncio' s internals and isn' t expected to be used by end-user code. In particular, if a Task gets successfully uncancelled, this allows for elements of structured concurrency like 태스크 그룹 and `asyncio.timeout()` to continue running, isolating cancellation to the respective structured block. For example:

```
async def make_request_with_timeout():
    try:
        async with asyncio.timeout(1):
            # 시간 제한의 영향을 받는 구조화된 블록:
            await make_request()
            await make_another_request()
    except TimeoutError:
        log("There was a timeout")
    # 시간 제한의 영향을 받지 않는 바깥 코드:
    await unrelated_code()
```

While the block with `make_request()` and `make_another_request()` might get cancelled due to the timeout, `unrelated_code()` should continue running even in case of the timeout. This is implemented with `uncancel()`. `TaskGroup` context managers use `uncancel()` in a similar fashion.

If end-user code is, for some reason, suppressing cancellation by catching `CancelledError`, it needs to call this method to remove the cancellation state.

When this method decrements the cancellation count to zero, the method checks if a previous `cancel()` call had arranged for `CancelledError` to be thrown into the task. If it hasn' t been thrown yet, that arrangement will be rescinded (by resetting the internal `_must_cancel` flag).

버전 3.13에서 변경: Changed to rescind pending cancellation requests upon reaching zero.

cancelling()

Return the number of pending cancellation requests to this Task, i.e., the number of calls to `cancel()` less the number of `uncancel()` calls.

Note that if this number is greater than zero but the Task is still executing, `cancelled()` will still return `False`. This is because this number can be lowered by calling `uncancel()`, which can lead to the task not being cancelled after all if the cancellation requests go down to zero.

This method is used by asyncio' s internals and isn' t expected to be used by end-user code. See `uncancel()` for more details.

Added in version 3.11.

19.1.3 스트림

소스 코드: Lib/asyncio/streams.py

스트림은 네트워크 연결로 작업하기 위해, async/await에서 사용할 수 있는 고수준 프리미티브입니다. 스트림은 콜백이나 저수준 프로토콜과 트랜스포트를 사용하지 않고 데이터를 송수신할 수 있게 합니다.

다음은 asyncio 스트림을 사용하여 작성된 TCP 메아리 클라이언트의 예입니다:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

아래의 예제 절도 참조하십시오.

스트림 함수

다음 최상위 asyncio 함수를 사용하여 스트림을 만들고 작업할 수 있습니다.:

async asyncio.**open_connection**(*host=None, port=None, *, limit=None, ssl=None, family=0, proto=0, flags=0, sock=None, local_addr=None, server_hostname=None, ssl_handshake_timeout=None, ssl_shutdown_timeout=None, happy_eyeballs_delay=None, interleave=None*)

네트워크 연결을 만들고 `(reader, writer)` 객체 쌍을 반환합니다.

반환된 *reader* 와 *writer* 객체는 `StreamReader` 와 `StreamWriter` 클래스의 인스턴스입니다.

*limit*는 반환된 `StreamReader` 인스턴스가 사용하는 버퍼 크기 한계를 결정합니다. 기본적으로 *limit* 는 64KiB로 설정됩니다.

나머지 인자는 `loop.create_connection()`로 직접 전달됩니다.

> **참고**
>
> The *sock* argument transfers ownership of the socket to the `StreamWriter` created. To close the socket, call its `close()` method.

버전 3.7에서 변경: *ssl_handshake_timeout* 매개 변수를 추가했습니다.

버전 3.8에서 변경: Added the *happy_eyeballs_delay* and *interleave* parameters.

버전 3.10에서 변경: Removed the *loop* parameter.

버전 3.11에서 변경: *ssl_shutdown_timeout* 매개 변수를 추가했습니다.

async asyncio.**start_server**(*client_connected_cb, host=None, port=None, *, limit=None, family=socket.AF_UNSPEC, flags=socket.AI_PASSIVE, sock=None, backlog=100, ssl=None, reuse_address=None, reuse_port=None, keep_alive=None, ssl_handshake_timeout=None, ssl_shutdown_timeout=None, start_serving=True*)

소켓 서버를 시작합니다.

새 클라이언트 연결이 만들어질 때마다 *client_connected_cb* 콜백이 호출됩니다. 이 콜백은 두 개의 인자로 `(reader, writer)` 쌍을 받는데, `StreamReader` 와 `StreamWriter` 클래스의 인스턴스입니다.

*client_connected_cb*는 일반 콜러블이나 코루틴 함수 일 수 있습니다; 코루틴 함수면, 자동으로 `Task`로 예약됩니다.

*limit*는 반환된 `StreamReader` 인스턴스가 사용하는 버퍼 크기 한계를 결정합니다. 기본적으로 *limit*는 64KiB로 설정됩니다.

나머지 인자는 `loop.create_server()`로 직접 전달됩니다.

> **참고**
>
> The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server' s `close()` method.

버전 3.7에서 변경: *ssl_handshake_timeout* 와 *start_serving* 매개 변수를 추가했습니다.

버전 3.10에서 변경: Removed the *loop* parameter.

버전 3.11에서 변경: *ssl_shutdown_timeout* 매개 변수를 추가했습니다.

버전 3.13에서 변경: *keep_alive* 매개 변수를 추가했습니다.

유닉스 소켓

async asyncio.**open_unix_connection**(*path=None, *, limit=None, ssl=None, sock=None, server_hostname=None, ssl_handshake_timeout=None, ssl_shutdown_timeout=None*)

유닉스 소켓 연결을 만들고 `(reader, writer)` 쌍을 반환합니다.

`open_connection()`과 비슷하지만, 유닉스 소켓에서 작동합니다.

`loop.create_unix_connection()`의 설명서도 참조하십시오.

> **참고**
>
> The *sock* argument transfers ownership of the socket to the `StreamWriter` created. To close the socket, call its `close()` method.

가용성: Unix.

버전 3.7에서 변경: *ssl_handshake_timeout* 매개 변수를 추가했습니다. *path* 매개 변수는 이제 경로류 객체가 될 수 있습니다.

버전 3.10에서 변경: Removed the *loop* parameter.

버전 3.11에서 변경: *ssl_shutdown_timeout* 매개 변수를 추가했습니다.

async asyncio.**start_unix_server**(*client_connected_cb, path=None, *, limit=None, sock=None, backlog=100, ssl=None, ssl_handshake_timeout=None, ssl_shutdown_timeout=None, start_serving=True, cleanup_socket=True*)

유닉스 소켓 서버를 시작합니다.

`start_server()`와 비슷하지만, 유닉스 소켓에서 작동합니다.

If *cleanup_socket* is true then the Unix socket will automatically be removed from the filesystem when the server is closed, unless the socket has been replaced after the server has been created.

`loop.create_unix_server()`의 설명서도 참조하십시오.

> **참고**
>
> The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server' s `close()` method.

가용성: Unix.

버전 3.7에서 변경: *ssl_handshake_timeout* 와 *start_serving* 매개 변수를 추가했습니다. *path* 매개 변수는 이제 경로류 객체가 될 수 있습니다.

버전 3.10에서 변경: Removed the *loop* parameter.

버전 3.11에서 변경: *ssl_shutdown_timeout* 매개 변수를 추가했습니다.

버전 3.13에서 변경: Added the *cleanup_socket* parameter.

StreamReader

class asyncio.**StreamReader**

IO 스트림에서 데이터를 읽는 API를 제공하는 판독기(reader) 객체를 나타냅니다. 비동기 이터러블로서, 객체는 `async for` 문을 지원합니다.

StreamReader 객체를 직접 인스턴스로 만드는 것은 권장되지 않습니다. 대신 `open_connection()` 과 `start_server()` 를 사용하십시오.

feed_eof()

Acknowledge the EOF.

async read(*n=-1*)

Read up to *n* bytes from the stream.

n 이 제공되지 않거나 `-1` 로 설정되면, EOF까지 읽은 다음, 모든 읽어들인 `bytes` 를 반환합니다. EOF를 수신했고 내부 버퍼가 비어 있으면, 빈 `bytes` 객체를 반환합니다.

If *n* is `0`, return an empty `bytes` object immediately.

If *n* is positive, return at most *n* available `bytes` as soon as at least 1 byte is available in the internal buffer. If EOF is received before any byte is read, return an empty `bytes` object.

async readline()

한 줄을 읽습니다. 여기서 "줄" 은 `\n` 로 끝나는 바이트의 시퀀스입니다.

EOF를 수신했고, `\n` 를 찾을 수 없으면, 이 메서드는 부분적으로 읽은 데이터를 반환합니다.

EOF를 수신했고, 내부 버퍼가 비어 있으면 빈 `bytes` 객체를 반환합니다.

async readexactly(*n*)

정확히 *n* 바이트를 읽습니다.

n 바이트를 읽기 전에 EOF에 도달하면, `IncompleteReadError` 를 일으킵니다. 부분적으로 읽은 데이터를 가져오려면 `IncompleteReadError.partial` 어트리뷰트를 사용하십시오.

async readuntil(*separator=b'\n'*)

separator 가 발견될 때까지 스트림에서 데이터를 읽습니다.

성공하면, 데이터와 separator가 내부 버퍼에서 제거됩니다 (소비됩니다). 반환된 데이터에는 끝에 separator가 포함됩니다.

읽은 데이터의 양이 구성된 스트림 제한을 초과하면 `LimitOverrunError` 예외가 발생하고, 데이터는 내부 버퍼에 그대로 남아 있으며 다시 읽을 수 있습니다.

완전한 separator가 발견되기 전에 EOF에 도달하면 `IncompleteReadError` 예외가 발생하고, 내부 버퍼가 재설정됩니다. `IncompleteReadError.partial` 어트리뷰트에는 separator 일부가 포함될 수 있습니다.

The *separator* may also be a tuple of separators. In this case the return value will be the shortest possible that has any separator as the suffix. For the purposes of `LimitOverrunError`, the shortest possible separator is considered to be the one that matched.

Added in version 3.5.2.

버전 3.13에서 변경: *separator* 매개 변수는 이제 구분자들의 `tuple`일 수 있습니다.

at_eof()

버퍼가 비어 있고 `feed_eof()`가 호출되었으면 `True`를 반환합니다.

StreamWriter

class asyncio.**StreamWriter**

IO 스트림에 데이터를 쓰는 API를 제공하는 기록기(writer) 객체를 나타냅니다.

StreamWriter 객체를 직접 인스턴스로 만드는 것은 권장되지 않습니다. 대신 `open_connection()`과 `start_server()`를 사용하십시오.

write(*data*)

이 메서드는 하부 소켓에 *data*를 즉시 기록하려고 시도합니다. 실패하면, data는 보낼 수 있을 때까지 내부 쓰기 버퍼에 계류됩니다.

이 메서드는 `drain()` 메서드와 함께 사용해야 합니다:

```
stream.write(data)
await stream.drain()
```

> **참고**
>
> The *data* buffer should be a C contiguous one-dimensional *bytes-like object*.

writelines(*data*)

이 메서드는 하부 소켓에 바이트열의 리스트(또는 임의의 이터러블)를 즉시 기록합니다. 실패하면, data는 보낼 수 있을 때까지 내부 쓰기 버퍼에 계류됩니다.

이 메서드는 `drain()` 메서드와 함께 사용해야 합니다:

```
stream.writelines(lines)
await stream.drain()
```

close()

이 메서드는 스트림과 하부 소켓을 닫습니다.

이 메서드는 필수는 아니지만 `wait_closed()` 메서드와 함께 사용해야 합니다:

```
stream.close()
await stream.wait_closed()
```

can_write_eof()

하부 트랜스포트가 `write_eof()` 메서드를 지원하면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

write_eof()

버퍼링 된 쓰기 데이터가 플러시 된 후에 스트림의 쓰기 끝을 닫습니다.

transport

하부 asyncio 트랜스포트를 돌려줍니다.

get_extra_info(*name*, *default=None*)

선택적 트랜스포트 정보에 액세스합니다; 자세한 내용은 `BaseTransport.get_extra_info()`를 참조하십시오.

async drain()

스트림에 기록을 다시 시작하는 것이 적절할 때까지 기다립니다. 예:

```
writer.write(data)
await writer.drain()
```

이것은 하부 IO 쓰기 버퍼와 상호 작용하는 흐름 제어 메서드입니다. 버퍼의 크기가 높은 수위에 도달하면, 버퍼 크기가 낮은 수위까지 내려가서 쓰기가 다시 시작될 수 있을 때까지 *drain*()은 블록합니다. 기다릴 것이 없으면, `drain()`은 즉시 반환합니다.

async start_tls(*sslcontext*, **, server_hostname=None, ssl_handshake_timeout=None, ssl_shutdown_timeout=None*)

Upgrade an existing stream-based connection to TLS.

Parameters:

- *sslcontext*: a configured instance of `SSLContext`.
- *server_hostname*: sets or overrides the host name that the target server' s certificate will be matched against.
- *ssl_handshake_timeout* is the time in seconds to wait for the TLS handshake to complete before aborting the connection. `60.0` seconds if `None` (default).
- *ssl_shutdown_timeout* is the time in seconds to wait for the SSL shutdown to complete before aborting the connection. `30.0` seconds if `None` (default).

Added in version 3.11.

버전 3.12에서 변경: *ssl_shutdown_timeout* 매개 변수를 추가했습니다.

is_closing()

스트림이 닫혔거나 닫히고 있으면 `True`를 반환합니다.

Added in version 3.7.

async wait_closed()

스트림이 닫힐 때까지 기다립니다.

가령 프로그램 종료 전에 모든 데이터가 플러시되었음을 확인하기위해, 하부 연결이 닫힐 때까지 기다리려면 `close()` 뒤에 호출해야 합니다.

Added in version 3.7.

예제

스트림을 사용하는 TCP 메아리 클라이언트

`asyncio.open_connection()` 함수를 사용하는 TCP 메아리 클라이언트:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

더 보기

TCP 메아리 클라이언트 프로토콜 예제는 저수준 `loop.create_connection()` 메서드를 사용합니다.

스트림을 사용하는 TCP 메아리 서버

`asyncio.start_server()` 함수를 사용하는 TCP 메아리 서버:

```
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')

    print(f"Received {message!r} from {addr!r}")

    print(f"Send: {message!r}")
    writer.write(data)
    await writer.drain()

    print("Close the connection")
    writer.close()
    await writer.wait_closed()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addrs = ', '.join(str(sock.getsockname()) for sock in server.sockets)
    print(f'Serving on {addrs}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

더 보기

TCP 메아리 서버 프로토콜 예제는 `loop.create_server()` 메서드를 사용합니다.

HTTP 헤더 가져오기

명령 줄로 전달된 URL의 HTTP 헤더를 조회하는 간단한 예제:

```
import asyncio
import urllib.parse
import sys

async def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(
            url.hostname, 80)

    query = (
        f"HEAD {url.path or '/'} HTTP/1.0\r\n"
        f"Host: {url.hostname}\r\n"
        f"\r\n"
    )

    writer.write(query.encode('latin-1'))
    while True:
        line = await reader.readline()
        if not line:
            break

        line = line.decode('latin1').rstrip()
        if line:
            print(f'HTTP header> {line}')

    # 바디를 무시하고, 소켓을 닫습니다
    writer.close()
    await writer.wait_closed()

url = sys.argv[1]
asyncio.run(print_http_headers(url))
```

사용법:

```
python example.py http://example.com/path/page.html
```

또는 HTTPS를 사용하면:

```
python example.py https://example.com/path/page.html
```

스트림을 사용하여 데이터를 기다리는 열린 소켓 등록

open_connection() 함수를 사용하여 소켓이 데이터를 수신할 때까지 기다리는 코루틴:

```
import asyncio
import socket

async def wait_for_data():
    # 저수준 API에 액세스하기 위해 현재 이벤트 루프에 대한 참조를 가져옵니다.
    loop = asyncio.get_running_loop()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    # 연결된 소켓 쌍을 만듭니다.
    rsock, wsock = socket.socketpair()

    # 데이터를 기다리는 열린 소켓을 등록합니다.
    reader, writer = await asyncio.open_connection(sock=rsock)

    # 네트워크로부터의 데이터 수신을 시뮬레이션합니다
    loop.call_soon(wsock.send, 'abc'.encode())

    # 데이터를 기다립니다
    data = await reader.read(100)

    # 데이터를 받았습니다, 할 일을 마쳤습니다: 소켓을 닫습니다.
    print("Received:", data.decode())
    writer.close()
    await writer.wait_closed()

    # 두 번째 소켓을 닫습니다
    wsock.close()

asyncio.run(wait_for_data())
```

더 보기

프로토콜을 사용하여 데이터를 기다리는 열린 소켓 등록 예제는 저수준 프로토콜과 `loop.create_connection()` 메서드를 사용합니다.

파일 기술자에서 읽기 이벤트를 관찰하기 예제는 저수준 `loop.add_reader()` 메서드를 사용하여 파일 기술자를 관찰합니다.

19.1.4 동기화 프리미티브

소스 코드: Lib/asyncio/locks.py

---

asyncio 동기화 프리미티브는 `threading` 모듈의 것과 유사하도록 설계되었습니다만 두 가지 중요한 주의 사항이 있습니다:

- asyncio 프리미티브는 스레드 안전하지 않으므로, OS 스레드 동기화(이를 위해서는 `threading`을 사용하십시오)에 사용하면 안 됩니다.
- 이러한 동기화 프리미티브의 메서드는 *timeout* 인자를 받아들이지 않습니다; `asyncio.wait_for()` 함수를 사용하여 시간제한이 있는 연산을 수행하십시오.

asyncio에는 다음과 같은 기본 동기화 프리미티브가 있습니다:

- `Lock`
- `Event`
- `Condition`
- `Semaphore`
- `BoundedSemaphore`
- `Barrier`

Lock

class asyncio.**Lock**

asyncio 태스크를 위한 뮤텍스 록을 구현합니다. 스레드 안전하지 않습니다.

asyncio 록은 공유 자원에 대한 독점 액세스를 보장하는 데 사용될 수 있습니다.

Lock을 사용하는 가장 좋은 방법은 `async with` 문입니다:

```
lock = asyncio.Lock()

# ... 나중에
async with lock:
    # 공유 상태를 액세스합니다
```

이는 다음과 동등합니다:

```
lock = asyncio.Lock()

# ... 나중에
await lock.acquire()
try:
    # 공유 상태를 액세스합니다
finally:
    lock.release()
```

버전 3.10에서 변경: *loop* 매개 변수를 제거했습니다.

async acquire()

록을 얻습니다.

이 메서드는 록이 풀림(*unlocked*)이 될 때까지 기다리고, 잠김(*locked*)으로 설정한 다음 `True`를 반환합니다.

잠금이 해제되기를 기다리는 `acquire()`에서 둘 이상의 코루틴 블록 될 때, 결국 한 개의 코루틴만 진행됩니다.

록을 얻는 것은 공평(*fair*)합니다: 진행할 코루틴은 록을 기다리기 시작한 첫 번째 코루틴이 됩니다.

release()

록을 반납합니다.

록이 잠김(*locked*)이면 풀림(*unlocked*)으로 재설정하고 돌아옵니다.

록이 풀림(*unlocked*)이면 `RuntimeError`가 발생합니다.

locked()

록이 잠김(*locked*)이면 `True`를 반환합니다.

Event

class asyncio.**Event**

이벤트 객체. 스레드 안전하지 않습니다.

asyncio 이벤트는 어떤 이벤트가 발생했음을 여러 asyncio 태스크에 알리는 데 사용할 수 있습니다.

Event 객체는 `set()` 메서드로 참으로 설정하고 `clear()` 메서드로 거짓으로 재설정할 수 있는 내부 플래그를 관리합니다. `wait()` 메서드는 플래그가 참으로 설정될 때까지 블록합니다. 플래그는 초기에 거짓으로 설정됩니다.

버전 3.10에서 변경: *loop* 매개 변수를 제거했습니다. 예:

```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
    # Event 객체를 만듭니다.
    event = asyncio.Event()

    # 'event'가 설정될 때까지 대기 할 Task를 만듭니다.
    waiter_task = asyncio.create_task(waiter(event))

    # 1초 동안 잠잔 후에 event를 설정합니다.
    await asyncio.sleep(1)
    event.set()

    # waiter 태스크가 끝날 때까지 기다립니다.
    await waiter_task

asyncio.run(main())
```

async wait()

이벤트가 설정될 때까지 기다립니다.

이벤트가 설정되었으면 `True`를 즉시 반환합니다. 그렇지 않으면 다른 태스크가 *set()*을 호출할 때까지 블록합니다.

set()

이벤트를 설정합니다.

이벤트가 설정되기를 기다리는 모든 태스크는 즉시 깨어납니다.

clear()

이벤트를 지웁니다 (재설정).

Subsequent tasks awaiting on *wait()* will now block until the *set()* method is called again.

is_set()

이벤트가 설정되면 `True`를 반환합니다.

Condition

class asyncio.**Condition**(*lock=None*)

Condition 객체. 스레드 안전하지 않습니다.

asyncio 조건 프리미티브는 태스크가 어떤 이벤트가 발생하기를 기다린 다음 공유 자원에 독점적으로 액세스하는데 사용할 수 있습니다.

본질에서, Condition 객체는 *Event*와 *Lock*의 기능을 결합합니다. 여러 개의 Condition 객체가 하나의 Lock을 공유할 수 있으므로, 공유 자원의 특정 상태에 관심이 있는 다른 태스크 간에 공유 자원에 대한 독점적 액세스를 조정할 수 있습니다.

선택적 *lock* 인자는 *Lock* 객체나 `None` 이어야 합니다. 후자의 경우 새로운 Lock 객체가 자동으로 만들어집니다.

버전 3.10에서 변경: *loop* 매개 변수를 제거했습니다.

Condition을 사용하는 가장 좋은 방법은 `async with` 문입니다:

```
cond = asyncio.Condition()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# ... 나중에
async with cond:
    await cond.wait()
```

이는 다음과 동등합니다:

```
cond = asyncio.Condition()

# ... 나중에
await cond.acquire()
try:
    await cond.wait()
finally:
    cond.release()
```

async acquire()

하부 록을 얻습니다.

이 메서드는 하부 록이 풀림(*unlocked*)이 될 때까지 대기하고, 잠김(*locked*)으로 설정한 다음 `True`를 반환합니다.

notify(*n=1*)

이 조건을 기다리는 *n* 태스크(기본적으로 1개)를 깨웁니다. 대기 중인 태스크가 *n* 개 보다 작으면 모두 깨어납니다.

이 메서드를 호출하기 전에 록을 얻어야 하고, 호출 직후에 반납해야 합니다. 풀린(*unlocked*) 록으로 호출하면 `RuntimeError` 에러가 발생합니다.

locked()

하부 록을 얻었으면 `True`를 돌려줍니다.

notify_all()

이 조건에 대기 중인 모든 태스크를 깨웁니다.

이 메서드는 `notify()`처럼 작동하지만, 대기 중인 모든 태스크를 깨웁니다.

이 메서드를 호출하기 전에 록을 얻어야 하고, 호출 직후에 반납해야 합니다. 풀린(*unlocked*) 록으로 호출하면 `RuntimeError` 에러가 발생합니다.

release()

하부 록을 반납합니다.

풀린 록으로 호출하면, `RuntimeError`가 발생합니다.

async wait()

알릴 때까지 기다립니다.

이 메서드가 호출될 때 호출하는 태스크가 록을 얻지 않았으면 `RuntimeError`가 발생합니다.

이 메서드는 하부 잠금을 반납한 다음, `notify()` 나 `notify_all()` 호출 때문에 깨어날 때까지 블록합니다. 일단 깨어나면, Condition은 록을 다시 얻고, 이 메서드는 `True`를 돌려줍니다.

Note that a task *may* return from this call spuriously, which is why the caller should always re-check the state and be prepared to `wait()` again. For this reason, you may prefer to use `wait_for()` instead.

async wait_for(*predicate*)

predicate가 참이 될 때까지 기다립니다.

predicate는 논릿값으로 해석될 결과를 돌려주는 콜러블이어야 합니다. 메서드는 predicate가 참으로 평가될 때까지 반복적으로 `wait()`합니다. 최종값이 반환 값입니다.

Semaphore

class asyncio.**Semaphore**(*value=1*)

Semaphore 객체. 스레드 안전하지 않습니다.

세마포어는 각 *acquire()* 호출로 감소하고, 각 *release()* 호출로 증가하는 내부 카운터를 관리합니다. 카운터는 절대로 0 밑으로 내려갈 수 없습니다; *acquire()* 가 0을 만나면, *release()* 를 호출할 때까지 기다리면서 블록합니다.

선택적 *value* 인자는 내부 카운터의 초깃값을 제공합니다 (기본적으로 1). 지정된 값이 0보다 작으면 *ValueError*가 발생합니다.

버전 3.10에서 변경: *loop* 매개 변수를 제거했습니다.

Semaphore를 사용하는 가장 좋은 방법은 `async with` 문입니다:

```
sem = asyncio.Semaphore(10)

# ... 나중에
async with sem:
    # 공유 자원으로 작업합니다
```

이는 다음과 동등합니다:

```
sem = asyncio.Semaphore(10)

# ... 나중에
await sem.acquire()
try:
    # 공유 자원으로 작업합니다
finally:
    sem.release()
```

async acquire()

세마포어를 얻습니다.

내부 카운터가 0보다 크면, 1 감소시키고 `True`를 즉시 반환합니다. 0이면, *release()* 가 호출될 때까지 기다린 다음 `True`를 반환합니다.

locked()

세마포어를 즉시 얻을 수 없으면 `True`를 반환합니다.

release()

세마포어를 반납하고 내부 카운터를 1 증가시킵니다. 세마포어를 얻기 위해 대기하는 태스크를 깨울 수 있습니다.

*BoundedSemaphore*와 달리, *Semaphore*는 `acquire()` 호출보다 더 많은 `release()` 호출을 허용합니다.

BoundedSemaphore

class asyncio.**BoundedSemaphore**(*value=1*)

제한된 세마포어 객체. 스레드 안전하지 않습니다.

제한된 세마포어는 초기 *value* 위로 내부 카운터를 증가시키면 *release()* 에서 *ValueError*를 발생시키는 *Semaphore* 버전입니다.

버전 3.10에서 변경: *loop* 매개 변수를 제거했습니다.

Barrier

class asyncio.**Barrier**(*parties*)

장벽 객체. 스레드 안전하지 않습니다.

A barrier is a simple synchronization primitive that allows to block until *parties* number of tasks are waiting on it. Tasks can wait on the `wait()` method and would be blocked until the specified number of tasks end up waiting on `wait()`. At that point all of the waiting tasks would unblock simultaneously.

`async with` can be used as an alternative to awaiting on `wait()`.

The barrier can be reused any number of times.

예:

```
async def example_barrier():
   # 3자 간 장벽
   b = asyncio.Barrier(3)

   # 기다리는 태스크를 2개 만듭니다
   asyncio.create_task(b.wait())
   asyncio.create_task(b.wait())

   await asyncio.sleep(0)
   print(b)

   # 세 번째 .wait() 호출은 장벽을 통과합니다
   await b.wait()
   print(b)
   print("barrier passed")

   await asyncio.sleep(0)
   print(b)

asyncio.run(example_barrier())
```

Result of this example is:

```
<asyncio.locks.Barrier object at 0x... [filling, waiters:2/3]>
<asyncio.locks.Barrier object at 0x... [draining, waiters:0/3]>
barrier passed
<asyncio.locks.Barrier object at 0x... [filling, waiters:0/3]>
```

Added in version 3.11.

async wait()

Pass the barrier. When all the tasks party to the barrier have called this function, they are all unblocked simultaneously.

When a waiting or blocked task in the barrier is cancelled, this task exits the barrier which stays in the same state. If the state of the barrier is "filling", the number of waiting task decreases by 1.

The return value is an integer in the range of 0 to `parties-1`, different for each task. This can be used to select a task to do some special housekeeping, e.g.:

```
...
async with barrier as position:
   if position == 0:
      # 오직 한 태스크만 이것을 인쇄합니다
      print('End of *draining phase*')
```

This method may raise a `BrokenBarrierError` exception if the barrier is broken or reset while a task is waiting. It could raise a `CancelledError` if a task is cancelled.

async reset()

Return the barrier to the default, empty state. Any tasks waiting on it will receive the `BrokenBarrierError` exception.

If a barrier is broken it may be better to just leave it and create a new one.

async abort()

Put the barrier into a broken state. This causes any active or future calls to `wait()` to fail with the `BrokenBarrierError`. Use this for example if one of the tasks needs to abort, to avoid infinite waiting tasks.

parties

The number of tasks required to pass the barrier.

n_waiting

The number of tasks currently waiting in the barrier while filling.

broken

A boolean that is `True` if the barrier is in the broken state.

exception asyncio.**BrokenBarrierError**

This exception, a subclass of `RuntimeError`, is raised when the `Barrier` object is reset or broken.

버전 3.9에서 변경: `await lock` 이나 `yield from lock` 및/또는 `with` 문(`with await lock`, `with (yield from lock)`)을 사용하여 록을 얻는 것은 제거되었습니다. 대신 `async with lock`을 사용하십시오.

19.1.5 서브 프로세스

소스 코드: Lib/asyncio/subprocess.py, Lib/asyncio/base_subprocess.py

---

이 절에서는 서브 프로세스를 만들고 관리하기 위한 고수준 async/await asyncio API에 관해 설명합니다.

다음은 asyncio가 셸 명령을 실행하고 결과를 얻는 방법의 예입니다:

```
import asyncio

async def run(cmd):
    proc = await asyncio.create_subprocess_shell(
        cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE)

    stdout, stderr = await proc.communicate()

    print(f'[{cmd!r} exited with {proc.returncode}]')
    if stdout:
        print(f'[stdout]\n{stdout.decode()}')
    if stderr:
        print(f'[stderr]\n{stderr.decode()}')

asyncio.run(run('ls /zzz'))
```

는 다음과 같이 인쇄할 것입니다:

```
['ls /zzz' exited with 1]
[stderr]
ls: /zzz: No such file or directory
```

모든 asyncio 서브 프로세스 함수는 비동기이고, asyncio가 이러한 함수로 작업 할 수 있는 많은 도구를 제공하기 때문에, 여러 서브 프로세스를 병렬로 실행하고 감시하기가 쉽습니다. 여러 명령을 동시에 실행하도록 위 예제를 수정하는 것은 아주 간단합니다:

```
async def main():
    await asyncio.gather(
        run('ls /zzz'),
        run('sleep 1; echo "hello"'))

asyncio.run(main())
```

예제 하위 절도 참조하십시오.

서브 프로세스 만들기

async asyncio.**create_subprocess_exec**(*program, *args, stdin=None, stdout=None, stderr=None, limit=None, **kwds*)

서브 프로세스를 만듭니다.

limit 인자는 `stdout` 과 `stderr`에 대한 `StreamReader` 래퍼의 버퍼 한계를 설정합니다 (`subprocess.PIPE`가 *stdout* 및 *stderr* 인자에 전달되었을 때).

`Process` 인스턴스를 반환합니다.

다른 매개 변수에 관해서는 `loop.subprocess_exec()`의 설명서를 참조하십시오.

버전 3.10에서 변경: *loop* 매개 변수를 제거했습니다.

async asyncio.**create_subprocess_shell**(*cmd, stdin=None, stdout=None, stderr=None, limit=None, **kwds*)

cmd 셸 명령을 실행합니다.

limit 인자는 `stdout` 과 `stderr`에 대한 `StreamReader` 래퍼의 버퍼 한계를 설정합니다 (`subprocess.PIPE`가 *stdout* 및 *stderr* 인자에 전달되었을 때).

`Process` 인스턴스를 반환합니다.

다른 매개 변수에 관해서는 `loop.subprocess_shell()`의 설명서를 참조하십시오.

> **중요**
>
> 셸 주입 취약점을 피하고자 모든 공백과 특수 문자를 적절하게 따옴표로 감싸는 것은 응용 프로그램의 책임입니다. `shlex.quote()` 함수는 셸 명령을 구성하는 데 사용될 문자열의 공백 문자와 특수 셸 문자를 올바르게 이스케이프 하는 데 사용할 수 있습니다.

버전 3.10에서 변경: *loop* 매개 변수를 제거했습니다.

> **참고**
>
> `ProactorEventLoop`를 쓰면 윈도우에서 서브 프로세스를 사용할 수 있습니다. 자세한 내용은 윈도우에서의 서브 프로세스 지원을 참조하십시오.

> **더 보기**
>
> 또한, asyncio에는 서브 프로세스와 함께 작동하는 다음과 같은 저수준 API가 있습니다: 서브 프로세스 트랜스포트 와 서브 프로세스 프로토콜 뿐만 아니라 `loop.subprocess_exec()`, `loop.subprocess_shell()`, `loop.connect_read_pipe()`, `loop.connect_write_pipe()`.

상수

`asyncio.subprocess.`**`PIPE`**

stdin, *stdout* 또는 *stderr* 매개 변수로 전달될 수 있습니다.

*PIPE*가 *stdin* 인자로 전달되면, `Process.stdin` 어트리뷰트는 `StreamWriter` 인스턴스를 가리킵니다.

*PIPE*가 *stdout* 이나 *stderr* 인자로 전달되면, `Process.stdout` 과 `Process.stderr` 어트리뷰트는 `StreamReader` 인스턴스를 가리 킵니다.

`asyncio.subprocess.`**`STDOUT`**

stderr 인자로 사용할 수 있는 특수 값이며, 표준 에러를 표준 출력으로 리디렉션해야 함을 나타냅니다.

`asyncio.subprocess.`**`DEVNULL`**

프로세스 생성 함수의 *stdin*, *stdout* 또는 *stderr* 인자로 사용할 수 있는 특수 값입니다. 특수 파일 `os.devnull`이 해당 서브 프로세스 스트림에 사용됨을 나타냅니다.

서브 프로세스와 상호 작용하기

`create_subprocess_exec()` 와 `create_subprocess_shell()` 함수는 모두 *Process* 클래스의 인스턴스를 반환합니다. *Process*는 서브 프로세스와 통신하고 완료를 관찰할 수 있는 고수준 래퍼입니다.

`class` `asyncio.subprocess.`**`Process`**

`create_subprocess_exec()` 와 `create_subprocess_shell()` 함수로 만들어진 OS 프로세스를 감싸는 객체.

이 클래스는 `subprocess.Popen` 클래스와 비슷한 API를 갖도록 설계되었지만, 주목할만한 차이점이 있습니다:

- Popen과 달리, Process 인스턴스에는 `poll()` 메서드와 동등한 것이 없습니다;
- `communicate()` 와 `wait()` 메서드에는 *timeout* 매개 변수가 없습니다: `wait_for()` 함수를 사용하십시오;
- `Process.wait()` 메서드는 비동기이지만, `subprocess.Popen.wait()` 메서드는 블로킹 비지 루프(blocking busy loop)로 구현됩니다;
- *universal_newlines* 매개 변수는 지원되지 않습니다.

이 클래스는 스레드 안전하지 않습니다.

서브 프로세스와 스레드 절도 참조하십시오.

`async wait``()`

자식 프로세스가 종료할 때까지 기다립니다.

`returncode` 어트리뷰트를 설정하고 반환합니다.

> **참고**
>
> 이 메서드는 `stdout=PIPE` 나 `stderr=PIPE`를 사용하고 자식 프로세스가 너무 많은 출력을 만들면 교착 상태가 될 수 있습니다. 자식 프로세스는 OS 파이프 버퍼가 더 많은 데이터를 받아들이도록 기다리면서 블록 됩니다. 이 조건을 피하고자, 파이프를 사용할 때는 `communicate()` 메서드를 사용하십시오.

async communicate(*input=None*)

프로세스와 상호 작용합니다:

1. 데이터를 *stdin*으로 보냅니다 (*input*이 `None`이 아니면);
2. *stdin*을 닫습니다;
3. EOF에 도달할 때까지 *stdout* 과 *stderr*에서 데이터를 읽습니다;
4. 프로세스가 종료할 때까지 기다립니다.

선택적 *input* 인자는 자식 프로세스로 전송될 데이터(`bytes` 객체)입니다.

튜플 `(stdout_data, stderr_data)`를 반환합니다.

*input*을 *stdin*에 쓸 때 `BrokenPipeError` 나 `ConnectionResetError` 예외가 발생하면, 예외를 무시합니다. 이 조건은 모든 데이터가 *stdin*에 기록되기 전에 프로세스가 종료할 때 발생합니다.

프로세스의 '*stdin*으로 데이터를 보내려면, 프로세스를 `stdin=PIPE`로 만들어야 합니다. 마찬가지로, 결과 튜플에서 `None` 이외의 것을 얻으려면, `stdout=PIPE` 와/나 `stderr=PIPE` 인자를 사용하여 프로세스를 만들어야 합니다.

데이터가 메모리에 버퍼링 되므로, 데이터 크기가 크거나 무제한이면 이 메서드를 사용하지 마십시오.

버전 3.12에서 변경: `input=None`일 때도 *stdin*이 닫힙니다.

send_signal(*signal*)

시그널 *signal*를 자식 프로세스로 보냅니다.

> **참고**
>
> 윈도우에서, `SIGTERM`은 `terminate()`의 별칭입니다. `CTRL_C_EVENT` 와 `CTRL_BREAK_EVENT`는 `CREATE_NEW_PROCESS_GROUP`을 포함하는 *creationflags* 매개 변수로 시작된 프로세스로 전송될 수 있습니다.

terminate()

자식 프로세스를 중지합니다.

POSIX 시스템에서 이 메서드는 `SIGTERM`를 자식 프로세스로 보냅니다.

윈도우에서는 Win32 API 함수 `TerminateProcess()`가 호출되어 자식 프로세스를 중지합니다.

kill()

자식 프로세스를 죽입니다.

POSIX 시스템에서 이 메서드는 `SIGKILL`를 자식 프로세스로 보냅니다.

윈도우에서 이 메서드는 `terminate()`의 별칭입니다.

stdin

표준 입력 스트림(`StreamWriter`) 또는 프로세스가 `stdin=None`으로 만들어졌으면 `None`.

stdout

표준 출력 스트림(`StreamReader`) 또는 프로세스가 `stdout=None`으로 만들어졌으면 `None`.

stderr

표준 에러 스트림(`StreamReader`) 또는 프로세스가 `stderr=None`으로 만들어졌으면 `None`.

> **경고**
>
> `process.stdin.write()`, `await process.stdout.read()` 또는 `await process.stderr.read()` 대신 `communicate()` 메서드를 사용하십시오. 이렇게 하면 스트림이 읽기나 쓰기를 일시 중지하고 자식 프로세스를 블록하는 것으로 인한 교착 상태가 발생하지 않습니다.

pid

프로세스 식별 번호 (PID).

create_subprocess_shell() 함수로 만들어진 프로세스의 경우, 이 어트리뷰트는 생성된 셸의 PID입니다.

returncode

프로세스가 종료할 때의 반환 코드.

`None` 값은 프로세스가 아직 종료하지 않았음을 나타냅니다.

음수 값 `-N`은 자식이 시그널 `N`으로 종료되었음을 나타냅니다 (POSIX만 해당).

서브 프로세스와 스레드

표준 asyncio 이벤트 루프는 기본적으로 다른 스레드에서 서브 프로세스를 실행하는 것을 지원합니다.

윈도우에서 서브 프로세스는 *ProactorEventLoop*(기본값)에서만 제공되며, *SelectorEventLoop*에는 서브 프로세스 지원이 없습니다.

대체 이벤트 루프 구현에는 나름의 제한 사항이 있을 수 있습니다; 해당 설명서를 참조하십시오.

더 보기

asyncio의 동시성과 다중 스레드 절.

예제

Process 클래스를 사용하여 서브 프로세스를 제어하고 *StreamReader* 클래스를 사용하여 표준 출력을 읽는 예제.

서브 프로세스는 *create_subprocess_exec()* 함수로 만듭니다:

```
import asyncio
import sys

async def get_date():
    code = 'import datetime; print(datetime.datetime.now())'

    # 서브 프로세스를 만듭니다; 표준 출력을 파이프로 리디렉트합니다.
    proc = await asyncio.create_subprocess_exec(
        sys.executable, '-c', code,
        stdout=asyncio.subprocess.PIPE)

    # 한 줄의 출력을 읽습니다.
    data = await proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # 서브 프로세스 종료를 기다립니다.
    await proc.wait()
    return line

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

저수준 API를 사용하여 작성된 같은 예제도 참조하십시오.

19.1.6 큐

소스 코드: Lib/asyncio/queues.py

---

asyncio 큐는 `queue` 모듈의 클래스와 유사하도록 설계되었습니다. asyncio 큐는 스레드 안전하지 않지만, async/await 코드에서 사용되도록 설계되었습니다.

asyncio 큐의 메서드에는 *timeout* 매개 변수가 없습니다; 시간제한이 있는 큐 연산을 하려면 `asyncio.wait_for()` 함수를 사용하십시오.

아래의 예제 절도 참조하십시오.

Queue

class asyncio.**Queue**(*maxsize=0*)

선입 선출 (FIFO) 큐.

*maxsize*가 0보다 작거나 같으면 큐 크기는 무한합니다. `0`보다 큰 정수면, 큐가 *maxsize*에 도달했을 때 `get()`이 항목을 제거할 때까지 `await put()`이 블록합니다.

표준 라이브러리의 스레드를 쓰는 `queue`와는 달리, 큐의 크기는 항상 알려져 있으며 `qsize()` 메서드를 호출하여 얻을 수 있습니다.

버전 3.10에서 변경: *loop* 매개 변수를 제거했습니다.

이 클래스는 스레드 안전하지 않습니다.

maxsize

큐에 허용되는 항목 수.

empty()

큐가 비어 있으면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

full()

큐에 `maxsize` 항목이 있으면 `True`를 반환합니다.

큐가 `maxsize=0` (기본값)으로 초기화되었으면, `full()`은 절대 `True`를 반환하지 않습니다.

async get()

큐에서 항목을 제거하고 반환합니다. 큐가 비어 있으면, 항목이 들어올 때까지 기다립니다.

Raises `QueueShutDown` if the queue has been shut down and is empty, or if the queue has been shut down immediately.

get_nowait()

항목을 즉시 사용할 수 있으면 항목을 반환하고, 그렇지 않으면 `QueueEmpty`를 발생시킵니다.

async join()

큐의 모든 항목을 수신하여 처리할 때까지 블록합니다.

완료되지 않은 작업 수는 항목이 큐에 추가될 때마다 증가합니다. 이 수는 소비자 코루틴이 항목을 수신했고 그 항목에 관한 작업이 모두 완료되었음을 나타내는 `task_done()`를 호출할 때마다 감소합니다. 완료되지 않은 작업 수가 0으로 떨어지면 `join()`가 블록 해제됩니다.

async put(*item*)

큐에 항목을 넣습니다. 큐가 가득 차면, 항목을 추가할 빈자리가 생길 때까지 기다립니다.

Raises `QueueShutDown` if the queue has been shut down.

put_nowait(*item*)

블록하지 않고 항목을 큐에 넣습니다.

자리가 즉시 나지 않으면, `QueueFull`를 일으킵니다.

qsize()

큐에 있는 항목 수를 돌려줍니다.

shutdown(*immediate=False*)

Shut down the queue, making `get()` and `put()` raise `QueueShutDown`.

By default, `get()` on a shut down queue will only raise once the queue is empty. Set *immediate* to true to make `get()` raise immediately instead.

All blocked callers of `put()` and `get()` will be unblocked. If *immediate* is true, a task will be marked as done for each remaining item in the queue, which may unblock callers of `join()`.

Added in version 3.13.

task_done()

이전에 큐에 넣은 작업 항목이 완료되었음을 나타냅니다.

큐 소비자가 사용합니다. 작업 항목을 꺼내는 데 사용된 `get()` 마다, 뒤따르는 `task_done()` 호출은 작업 항목에 관한 처리가 완료되었음을 큐에 알려줍니다.

`join()`이 현재 블록 중이면, 모든 항목이 처리될 때 다시 시작됩니다 (큐에 `put()`한 모든 항목에 대해 `task_done()` 호출이 수신되었음을 뜻합니다).

`shutdown(immediate=True)` calls `task_done()` for each remaining item in the queue.

큐에 넣은 항목보다 더 많이 호출되면 `ValueError`를 발생시킵니다.

우선순위 큐

class asyncio.**PriorityQueue**

`Queue`의 변형; 우선순위 순서로 항목을 꺼냅니다 (가장 낮은 우선순위가 처음입니다).

엔트리는 일반적으로 `(priority_number, data)` 형식의 튜플입니다.

LIFO 큐

class asyncio.**LifoQueue**

가장 최근에 추가된 항목을 먼저 꺼내는 `Queue`의 변형 (후입 선출).

예외

exception asyncio.**QueueEmpty**

이 예외는 `get_nowait()` 메서드가 빈 큐에 호출될 때 발생합니다.

exception asyncio.**QueueFull**

`put_nowait()` 메서드가 *maxsize*에 도달한 큐에 호출될 때 발생하는 예외입니다.

exception asyncio.**QueueShutDown**

`put()` 이나 `get()` 메서드가 종료된 큐에 호출될 때 발생하는 예외입니다.

Added in version 3.13.

예제

큐를 사용하여 여러 동시 태스크로 작업 부하를 분산시킬 수 있습니다:

```
import asyncio
import random
import time


async def worker(name, queue):
    while True:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        # 큐에서 "작업 항목"을 가져옵니다.
        sleep_for = await queue.get()

        # "sleep_for" 초 동안 잡니다.
        await asyncio.sleep(sleep_for)

        # 큐에 "작업 항목"이 처리되었음을 알립니다.
        queue.task_done()

        print(f'{name} has slept for {sleep_for:.2f} seconds')


async def main():
    # "작업 부하"를 저장하는 데 사용할 큐를 만듭니다.
    queue = asyncio.Queue()

    # 무작위 대기 시간을 만들어서 큐에 넣습니다.
    total_sleep_time = 0
    for _ in range(20):
        sleep_for = random.uniform(0.05, 1.0)
        total_sleep_time += sleep_for
        queue.put_nowait(sleep_for)

    # 큐를 동시에 처리할 세 개의 worker 태스크를 만듭니다.
    tasks = []
    for i in range(3):
        task = asyncio.create_task(worker(f'worker-{i}', queue))
        tasks.append(task)

    # 큐가 완전히 처리될 때까지 기다립니다.
    started_at = time.monotonic()
    await queue.join()
    total_slept_for = time.monotonic() - started_at

    # worker 태스크를 취소합니다.
    for task in tasks:
        task.cancel()
    # 모든 worker 태스크가 취소될 때까지 기다립니다.
    await asyncio.gather(*tasks, return_exceptions=True)

    print('====')
    print(f'3 workers slept in parallel for {total_slept_for:.2f} seconds')
    print(f'total expected sleep time: {total_sleep_time:.2f} seconds')


asyncio.run(main())
```

19.1.7 예외

소스 코드: Lib/asyncio/exceptions.py

---

exception asyncio.**TimeoutError**

*TimeoutError*의 폐지된 별칭입니다. 작업이 주어진 마감 시간을 초과했을 때 발생합니다.

버전 3.11에서 변경: 이 클래스를 *TimeoutError*의 별칭으로 만들었습니다.

exception asyncio.**CancelledError**

작업이 취소되었습니다.

이 예외는 asyncio 태스크가 취소될 때 사용자 정의 작업을 수행하기 위해 잡을 수 있습니다. 거의 모든 상황에서 예외를 다시 일으켜야 합니다.

버전 3.8에서 변경: `CancelledError`는 이제 `Exception`가 아니라 `BaseException`의 서브 클래스입니다.

exception asyncio.**InvalidStateError**

`Task` 나 `Future`의 내부 상태가 잘못되었습니다.

이미 결괏값이 설정된 *Future* 객체에 대해 결괏값을 설정하는 것과 같은 상황에서 발생할 수 있습니다.

exception asyncio.**SendfileNotAvailableError**

주어진 소켓이나 파일 유형에서는 "sendfile" 시스템 호출을 사용할 수 없습니다.

`RuntimeError`의 서브 클래스입니다.

exception asyncio.**IncompleteReadError**

요청한 읽기 작업이 완전히 완료되지 않았습니다.

*asyncio 스트림 API*가 일으킵니다.

이 예외는 `EOFError`의 서브 클래스입니다.

expected

기대하는 바이트의 총수 (`int`).

partial

스트림이 끝나기 전에 읽은 `bytes` 문자열.

exception asyncio.**LimitOverrunError**

구분 기호를 찾는 동안 버퍼 크기 제한에 도달했습니다.

*asyncio 스트림 API*가 일으킵니다.

consumed

소비된 바이트의 총수.

19.1.8 Call Graph Introspection

Source code: Lib/asyncio/graph.py

---

asyncio has powerful runtime call graph introspection utilities to trace the entire call graph of a running *coroutine* or *task*, or a suspended *future*. These utilities and the underlying machinery can be used from within a Python program or by external profilers and debuggers.

Added in version 3.14.

asyncio.**print_call_graph**(*future=None*, */*, ***, *file=None*, *depth=1*, *limit=None*)

Print the async call graph for the current task or the provided `Task` or `Future`.

This function prints entries starting from the top frame and going down towards the invocation point.

The function receives an optional *future* argument. If not passed, the current running task will be used.

If the function is called on *the current task*, the optional keyword-only *depth* argument can be used to skip the specified number of frames from top of the stack.

If the optional keyword-only *limit* argument is provided, each call stack in the resulting graph is truncated to include at most `abs(limit)` entries. If *limit* is positive, the entries left are the closest to the invocation point. If *limit* is negative, the topmost entries are left. If *limit* is omitted or `None`, all entries are present. If *limit* is `0`, the call stack is not printed at all, only "awaited by" information is printed.

If *file* is omitted or `None`, the function will print to `sys.stdout`.

Example:

The following Python code:

```
import asyncio

async def test():
    asyncio.print_call_graph()

async def main():
    async with asyncio.TaskGroup() as g:
        g.create_task(test(), name='test')

asyncio.run(main())
```

will print:

```
* Task(name='test', id=0x1039f0fe0)
+ Call stack:
|   File 't2.py', line 4, in async test()
+ Awaited by:
  * Task(name='Task-1', id=0x103a5e060)
    + Call stack:
    |   File 'taskgroups.py', line 107, in async TaskGroup.__aexit__()
    |   File 't2.py', line 7, in async main()
```

`asyncio.`**`format_call_graph`**(*future=None*, */*, ***, *depth=1*, *limit=None*)

Like `print_call_graph()`, but returns a string. If *future* is `None` and there' s no current task, the function returns an empty string.

`asyncio.`**`capture_call_graph`**(*future=None*, */*, ***, *depth=1*, *limit=None*)

Capture the async call graph for the current task or the provided `Task` or `Future`.

The function receives an optional *future* argument. If not passed, the current running task will be used. If there' s no current task, the function returns `None`.

If the function is called on *the current task*, the optional keyword-only *depth* argument can be used to skip the specified number of frames from top of the stack.

Returns a `FutureCallGraph` data class object:

- `FutureCallGraph(future, call_stack, awaited_by)`

  Where *future* is a reference to a `Future` or a `Task` (or their subclasses.)

  `call_stack` is a tuple of `FrameCallGraphEntry` objects.

  `awaited_by` is a tuple of `FutureCallGraph` objects.

- `FrameCallGraphEntry(frame)`

  Where *frame* is a frame object of a regular Python function in the call stack.

Low level utility functions

To introspect an async call graph asyncio requires cooperation from control flow structures, such as `shield()` or `TaskGroup`. Any time an intermediate `Future` object with low-level APIs like `Future.add_done_callback()` is involved, the following two functions should be used to inform asyncio about how exactly such intermediate future objects are connected with the tasks they wrap or control.

asyncio.**future_add_to_awaited_by**(*future*, *waiter*, /)

Record that *future* is awaited on by *waiter*.

Both *future* and *waiter* must be instances of `Future` or `Task` or their subclasses, otherwise the call would have no effect.

A call to `future_add_to_awaited_by()` must be followed by an eventual call to the `future_discard_from_awaited_by()` function with the same arguments.

asyncio.**future_discard_from_awaited_by**(*future*, *waiter*, /)

Record that *future* is no longer awaited on by *waiter*.

Both *future* and *waiter* must be instances of `Future` or `Task` or their subclasses, otherwise the call would have no effect.

19.1.9 이벤트 루프

소스 코드: Lib/asyncio/events.py, Lib/asyncio/base_events.py

머리말

이벤트 루프는 모든 asyncio 응용 프로그램의 핵심입니다. 이벤트 루프는 비동기 태스크 및 콜백을 실행하고 네트워크 IO 연산을 수행하며 자식 프로세스를 실행합니다.

응용 프로그램 개발자는 일반적으로 `asyncio.run()`과 같은 고수준의 asyncio 함수를 사용해야 하며, 루프 객체를 참조하거나 메서드를 호출할 필요가 거의 없습니다. 이 절은 주로 이벤트 루프 동작을 세부적으로 제어해야 하는 저수준 코드, 라이브러리 및 프레임워크의 작성자를 대상으로 합니다.

이벤트 루프 얻기

다음 저수준 함수를 사용하여 이벤트 루프를 가져오거나 설정하거나 만들 수 있습니다.:

asyncio.**get_running_loop**()

현재 OS 스레드에서 실행 중인 이벤트 루프를 반환합니다.

실행 중인 이벤트 루프가 없으면 `RuntimeError`를 발생시킵니다.

이 함수는 코루틴이나 콜백에서만 호출할 수 있습니다.

Added in version 3.7.

asyncio.**get_event_loop**()

현재의 이벤트 루프를 가져옵니다.

When called from a coroutine or a callback (e.g. scheduled with call_soon or similar API), this function will always return the running event loop.

If there is no running event loop set, the function will return the result of the `get_event_loop_policy().get_event_loop()` call.

이 함수는 (특히 사용자 정의 이벤트 루프 정책을 사용할 때) 다소 복잡한 동작을 하므로, 코루틴과 콜백에서 `get_event_loop()`보다 `get_running_loop()` 함수를 사용하는 것이 좋습니다.

위에서 언급된 바와 같이, 이 저수준 함수를 사용하여 수동으로 이벤트 루프를 만들고 닫는 대신, 고수준 `asyncio.run()` 함수를 사용하는 것을 고려하십시오.

버전 3.14에서 변경: Raises a `RuntimeError` if there is no current event loop.

> **참고**
>
> The `asyncio` policy system is deprecated and will be removed in Python 3.16; from there on, this function will return the current running event loop if present else it will return the loop set by `set_event_loop()`.

asyncio.**set_event_loop**(*loop*)

*loop*를 현재 OS 스레드의 현재 이벤트 루프로 설정합니다.

asyncio.**new_event_loop**()

새 이벤트 루프 객체를 만들고 반환합니다.

get_event_loop(), *set_event_loop()* 및 *new_event_loop()* 함수의 동작은 사용자 정의 이벤트 루프 정책 설정에 의해 변경될 수 있음에 유의하십시오.

목차

이 설명서 페이지는 다음과 같은 절로 구성됩니다:

- 이벤트 루프 메서드 절은 이벤트 루프 API의 레퍼런스 설명서입니다.
- 콜백 핸들 절은 *loop.call_soon()* 및 *loop.call_later()*와 같은 예약 메서드에서 반환된 *Handle* 및 *TimerHandle* 인스턴스를 설명합니다.
- 서버 객체 절은 *loop.create_server()*와 같은 이벤트 루프 메서드에서 반환되는 형을 설명합니다.
- 이벤트 루프 구현 절은 *SelectorEventLoop* 및 *ProactorEventLoop* 클래스를 설명합니다.
- 예제 절에서는 일부 이벤트 루프 API로 작업하는 방법을 보여줍니다.

이벤트 루프 메서드

이벤트 루프에는 다음과 같은 **저수준** API가 있습니다:

- 루프 실행 및 중지
- 콜백 예약하기
- 지연된 콜백 예약
- 퓨처와 태스크 만들기
- 네트워크 연결 열기
- 네트워크 서버 만들기
- 파일 전송
- *TLS* 업그레이드
- 파일 기술자 관찰하기
- 소켓 객체로 직접 작업하기
- *DNS*
- 파이프로 작업하기
- 유닉스 시그널
- 스레드 또는 프로세스 풀에서 코드를 실행하기
- 에러 처리 *API*
- 디버그 모드 활성화
- 자식 프로세스 실행하기

루프 실행 및 중지

loop.**run_until_complete**(*future*)

future(*Future*의 인스턴스)가 완료할 때까지 실행합니다.

인자가 코루틴 객체 면, *asyncio.Task*로 실행되도록 묵시적으로 예약 됩니다.

퓨처의 결과를 반환하거나 퓨처의 예외를 일으킵니다.

loop.**run_forever**()

*stop()*가 호출될 때까지 이벤트 루프를 실행합니다.

run_forever() 가 호출되기 전에 *stop()* 이 호출되었으면, 루프는 시간제한 0으로 I/O 셀렉터를 한 번 폴링하고, I/O 이벤트에 따라 예약된 모든 콜백(과 이미 예약된 것들)을 실행한 다음 종료합니다.

만약 *stop()* 이 *run_forever()* 가 실행 중일 때 호출되면, 루프는 현재 걸려있는 콜백들을 실행한 다음 종료합니다. 콜백에 의해 예약되는 새 콜백은 이 경우 실행되지 않습니다; 대신 그것들은 다음에 *run_forever()*나 *run_until_complete()*가 호출될 때 실행됩니다.

loop.**stop**()

이벤트 루프를 중지합니다.

loop.**is_running**()

이벤트 루프가 현재 실행 중이면 `True` 를 반환합니다.

loop.**is_closed**()

이벤트 루프가 닫혔으면 `True` 를 반환합니다.

loop.**close**()

이벤트 루프를 닫습니다.

이 함수를 호출할 때 루프는 반드시 실행 중이지 않아야 합니다. 계류 중인 모든 콜백을 버립니다.

이 메서드는 모든 큐를 비우고 실행기를 종료하지만, 실행기가 완료할 때까지 기다리지 않습니다.

이 메서드는 멱등적(itempotent)이고 되돌릴 수 없습니다. 이벤트 루프가 닫힌 후에 다른 메서드를 호출해서는 안 됩니다.

async loop.**shutdown_asyncgens**()

현재 열려있는 비동기 제너레이터 객체를 모두 `aclose()` 호출로 닫도록 예약 합니다. 이 메서드를 호출한 후에는, 새 비동기 생성기가 이터레이트 되면 이벤트 루프에서 경고를 보냅니다. 예약된 모든 비동기 제너레이터를 신뢰성 있게 종료하는 데 사용해야 합니다.

*asyncio.run()*가 사용될 때 이 함수를 호출할 필요는 없다는 점에 유의하세요.

예:

```
try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

Added in version 3.6.

async loop.**shutdown_default_executor**(*timeout=None*)

기본 실행기의 닫힘을 예약하고 *ThreadPoolExecutor*의 모든 스레드가 조인(join)될 때까지 기다립니다. 이 메서드가 호출된 후, *loop.run_in_executor()*로 기본 실행기를 사용하면 *RuntimeError*가 발생합니다.

The *timeout* parameter specifies the amount of time (in *float* seconds) the executor will be given to finish joining. With the default, `None`, the executor is allowed an unlimited amount of time.

If the *timeout* is reached, a *RuntimeWarning* is emitted and the default executor is terminated without waiting for its threads to finish joining.

> **참고**
>
> Do not call this method when using `asyncio.run()`, as the latter handles default executor shutdown automatically.

Added in version 3.9.

버전 3.12에서 변경: *timeout* 매개 변수가 추가되었습니다.

콜백 예약하기

`loop.call_soon(callback, *args, context=None)`

이벤트 루프의 다음 이터레이션 때 *args* 인자로 호출할 *callback* 콜백을 예약합니다.

`asyncio.Handle` 인스턴스를 반환하는데, 나중에 콜백을 취소하는 데 사용할 수 있습니다.

콜백은 등록된 순서대로 호출됩니다. 각 콜백은 정확히 한 번 호출됩니다.

선택적인 키워드 전용 *context* 인자는 *callback* 을 실행할 사용자 정의 `contextvars.Context` 를 지정합니다. *context* 가 제공되지 않을 때, 콜백은 현재 컨텍스트를 사용합니다.

`call_soon_threadsafe()`와 달리, 이 메서드는 스레드 안전하지 않습니다.

`loop.call_soon_threadsafe(callback, *args, context=None)`

스레드 안전한 `call_soon()` 변형입니다. `call_soon()`은 스레드 안전하지 않기 때문에, 다른 스레드에서 콜백을 예약할 때, 반드시 이 함수를 사용해야 합니다.

This function is safe to be called from a reentrant context or signal handler, however, it is not safe or fruitful to use the returned handle in such contexts.

Raises `RuntimeError` if called on a loop that' s been closed. This can happen on a secondary thread when the main application is shutting down.

설명서의 동시성과 다중 스레딩 절을 참고하십시오.

버전 3.7에서 변경: *context* 키워드 전용 매개 변수가 추가되었습니다. 자세한 정보는 **PEP 567**을 보십시오.

> **참고**
>
> 대부분 `asyncio` 예약 함수는 키워드 인자 전달을 허용하지 않습니다. 그렇게 하려면 `functools.partial()`을 사용하십시오:
>
> ```
> # "print("Hello", flush=True)" 를 예약합니다
> loop.call_soon(
>  functools.partial(print, "Hello", flush=True))
> ```
>
> asyncio는 디버그 및 오류 메시지에서 partial 객체를 더욱 잘 표시할 수 있으므로, partial 객체를 사용하는 것이 람다를 사용하는 것보다 편리합니다.

지연된 콜백 예약

이벤트 루프는 콜백 함수가 미래의 어떤 시점에서 호출되도록 예약하는 메커니즘을 제공합니다. 이벤트 루프는 단조 시계를 사용하여 시간을 추적합니다.

`loop.call_later(delay, callback, *args, context=None)`

지정된 *delay* 초 (int 또는 float) 뒤에 *callback* 이 호출되도록 예약합니다.

`asyncio.TimerHandle` 의 인스턴스가 반환되는데, 콜백을 취소하는 데 사용할 수 있습니다.

callback 은 정확히 한번 호출됩니다. 두 콜백이 정확히 같은 시간에 예약되면, 어떤 것이 먼저 호출되는지는 정의되지 않습니다.

선택적 위치 *args* 는 호출 될 때 콜백에 전달됩니다. 콜백을 키워드 인자로 호출하고 싶으면 `functools.partial()` 를 사용하십시오.

선택적인 키워드 전용 *context* 인자는 *callback* 을 실행할 사용자 정의 `contextvars.Context` 를 지정할 수 있게 합니다. *context* 가 제공되지 않을 때는 현재 컨텍스트가 사용됩니다.

> **참고**
>
> For performance, callbacks scheduled with `loop.call_later()` may run up to one clock-resolution early (see `time.get_clock_info('monotonic').resolution`).

버전 3.7에서 변경: *context* 키워드 전용 매개 변수가 추가되었습니다. 자세한 정보는 **PEP 567**을 보십시오.

버전 3.8에서 변경: 파이썬 3.7 및 이전 버전에서 기본 이벤트 루프 구현을 사용할 때, *delay*는 하루를 초과할 수 없었습니다. 이 문제는 파이썬 3.8에서 수정되었습니다.

`loop.`**`call_at`**(*when*, *callback*, **args*, *context=None*)

지정된 절대 타임스탬프 *when*(int 또는 float)에 *callback* 이 호출되도록 예약합니다. `loop.time()` 과 같은 시간 참조를 사용하십시오.

이 메서드의 동작은 `call_later()`와 같습니다.

`asyncio.TimerHandle` 의 인스턴스가 반환되는데, 콜백을 취소하는 데 사용할 수 있습니다.

> **참고**
>
> For performance, callbacks scheduled with `loop.call_at()` may run up to one clock-resolution early (see `time.get_clock_info('monotonic').resolution`).

버전 3.7에서 변경: *context* 키워드 전용 매개 변수가 추가되었습니다. 자세한 정보는 **PEP 567**을 보십시오.

버전 3.8에서 변경: 파이썬 3.7 및 이전 버전에서 기본 이벤트 루프 구현을 사용할 때, *when*와 현재 시각의 차이는 하루를 초과할 수 없었습니다. 이 문제는 파이썬 3.8에서 수정되었습니다.

`loop.`**`time`**()

이벤트 루프의 내부 단조 시계에 따라, `float` 값으로 현재 시각을 반환합니다.

> **참고**
>
> 버전 3.8에서 변경: 파이썬 3.7 및 이전 버전에서 제한 시간(상대적인 *delay* 나 절대적인 *when*)은 1 일을 초과하지 않아야 했습니다. 이 문제는 파이썬 3.8에서 수정되었습니다.

> **더 보기**
>
> `asyncio.sleep()` 함수.

퓨처와 태스크 만들기

`loop.`**`create_future`**()

이벤트 루프에 연결된 `asyncio.Future` 객체를 만듭니다.

이것이 asyncio에서 퓨처를 만드는 데 선호되는 방법입니다. 이렇게 하면 제삼자 이벤트 루프가 Future 객체의 다른 구현(더 나은 성능이나 계측(instrumentation))을 제공할 수 있습니다

Added in version 3.5.2.

loop.**create_task**(*coro*, *, *name=None*, *context=None*, *eager_start=None*, **kwargs)

코루틴 *coro*의 실행을 예약합니다. `Task` 객체를 반환합니다.

제삼자 이벤트 루프는 상호 운용성을 위해 자신만의 `Task` 의 서브 클래스를 사용할 수 있습니다. 이 경우, 결과 형은 `Task` 의 서브 클래스입니다.

The full function signature is largely the same as that of the `Task` constructor (or factory) - all of the keyword arguments to this function are passed through to that interface.

name 인자가 제공되고 `None` 이 아니면, `Task.set_name()` 을 사용하여 태스크의 이름으로 설정됩니다.

선택적인 키워드 전용 *context* 인자는 *coro* 를 실행할 사용자 정의 `contextvars.Context` 를 지정할 수 있게 합니다. *context* 가 제공되지 않을 때는 현재 컨텍스트 사본이 만들어집니다.

An optional keyword-only *eager_start* argument allows specifying if the task should execute eagerly during the call to create_task, or be scheduled later. If *eager_start* is not passed the mode set by `loop.set_task_factory()` will be used.

버전 3.8에서 변경: *name* 매개 변수가 추가되었습니다.

버전 3.11에서 변경: *context* 매개 변수가 추가되었습니다.

버전 3.13.3에서 변경: Added `kwargs` which passes on arbitrary extra parameters, including `name` and `context`.

버전 3.13.4에서 변경: Rolled back the change that passes on *name* and *context* (if it is None), while still passing on other arbitrary keyword arguments (to avoid breaking backwards compatibility with 3.13.3).

버전 3.14에서 변경: All *kwargs* are now passed on. The *eager_start* parameter works with eager task factories.

loop.**set_task_factory**(*factory*)

`loop.create_task()` 에 의해 사용되는 태스크 팩토리를 설정합니다.

factory 가 `None` 이면 기본 태스크 팩토리가 설정됩니다. 그렇지 않으면, *factory* 는 반드시 콜러블이어야 하고, `(loop, coro, **kwargs)` 과 일치하는 서명을 가져야 합니다. 여기서 *loop* 는 활성 이벤트 루프에 대한 참조가 되고, *coro* 는 코루틴 객체가 됩니다. 콜러블은 모든 *kwargs* 를 전달해야 하며, `asyncio.Task` 호환 객체를 반환해야 합니다.

버전 3.13.3에서 변경: Required that all *kwargs* are passed on to `asyncio.Task`.

버전 3.13.4에서 변경: *name* is no longer passed to task factories. *context* is no longer passed to task factories if it is `None`.

버전 3.14에서 변경: *name* and *context* are now unconditionally passed on to task factories again.

loop.**get_task_factory**()

태스크 팩토리를 반환하거나, 기본값이 사용 중이면 `None` 을 반환합니다.

네트워크 연결 열기

async loop.**create_connection**(*protocol_factory*, *host=None*, *port=None*, *, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *happy_eyeballs_delay=None*, *interleave=None*, *all_errors=False*)

주어진 *host* 와 *port*로 지정된 주소로의 스트리밍 트랜스포트 연결을 엽니다.

소켓 패밀리는 *host*(또는 지정된 경우 *family*)에 따라 `AF_INET` 또는 `AF_INET6`일 수 있습니다.

소켓 유형은 `SOCK_STREAM`이 됩니다.

protocol_factory 는 반드시 *asyncio* 프로토콜 구현을 반환하는 콜러블이어야 합니다.

이 메서드는 백그라운드에서 연결을 맺으려고 시도합니다. 성공하면, `(transport, protocol)` 쌍을 반환합니다.

하부 연산의 시간순 개요는 다음과 같습니다:

1. 연결이 맺어지고, 이를 위한 트랜스포트(*transport*) 가 만들어집니다.
2. *protocol_factory* 가 인자 없이 호출되고, 프로토콜(*protocol*) 인스턴스를 반환할 것으로 기대됩니다.
3. 프로토콜 인스턴스는 `connection_made()` 메서드를 호출함으로써 트랜스포트와 연결됩니다.
4. 성공하면 `(transport, protocol)` 튜플이 반환됩니다.

만들어진 트랜스포트는 구현 의존적인 양방향 스트림입니다.

다른 인자들:

- *ssl*: 주어지고 거짓이 아니면, SSL/TLS 트랜스포트가 만들어집니다 (기본적으로는 평범한 TCP 트랜스포트가 만들어집니다). *ssl* 이 `ssl.SSLContext` 객체면, 트랜스포트를 만들 때 이 컨텍스트가 사용됩니다; *ssl* 이 `True` 면, `ssl.create_default_context()` 가 반환하는 기본 컨텍스트가 사용됩니다.

  더 보기

  SSL/TLS 보안 고려 사항

- *server_hostname*는 대상 서버의 인증서가 일치될 호스트 이름을 설정하거나 대체합니다. *ssl*이 `None`이 아닐 때만 전달되어야 합니다. 기본적으로 *host* 인자의 값이 사용됩니다. *host* 가 비어 있으면, 기본값이 없고 *server_hostname* 값을 전달해야 합니다. *server_hostname* 이 빈 문자열이면, 호스트 이름 일치가 비활성화됩니다 (이것은 심각한 보안 위험으로, 잠재적인 중간자 공격을 허용하게 됩니다).
- *family*, *proto*, *flags* 는 *host* 결정을 위해 getaddrinfo() 에 전달할 선택적 주소 패밀리, 프로토콜, 플래그입니다. 주어지면, 이것들은 모두 해당하는 `socket` 모듈 상수에 대응하는 정수여야 합니다.
- 주어지면, *happy_eyeballs_delay*는 이 연결에 대해 Happy Eyeballs를 활성화합니다. 다음 시도를 병렬로 시작하기 전에, 연결 시도가 완료되기를 기다리는 시간(초)을 나타내는 부동 소수점 숫자여야 합니다. 이것은 **RFC 8305**에 정의된 "연결 시도 지연(Connection Attempt Delay)" 입니다. RFC에서 권장하는 적절한 기본값은 `0.25`(250밀리 초)입니다.
- *interleave*는 호스트 이름이 여러 IP 주소로 해석될 때 주소 재정렬을 제어합니다. `0`이거나 지정되지 않으면, 재정렬이 수행되지 않고, 주소는 `getaddrinfo()`에 의해 반환된 순서대로 시도됩니다. 양의 정수가 지정되면, 주소는 주소 패밀리에 의해 인터리브 되고, 주어진 정수는 **RFC 8305**에 정의된 대로 "첫 번째 주소 패밀리 수(First Address Family Count)" 로 해석됩니다. 기본값은 *happy_eyeballs_delay*가 지정되지 않으면 `0`이고, 지정되면 `1`입니다.
- *sock* 이 주어지면, 트랜스포트가 사용할, 기존의 이미 연결된 `socket.socket` 객체여야 합니다. *sock* 이 주어지면, *host*, *port*, *family*, *proto*, *flags*, *happy_eyeballs_delay*, *interleave*, *local_addr* 를 지정해서는 안 됩니다.

  참고

  The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport' s `close()` method.

- *local_addr* 이 주어지면, 소켓을 로컬에 바인드 하는데 사용되는 `(local_host, local_port)` 튜플이어야 합니다. *local_host* 와 *local_port* 는 *host* 및 *port* 와 유사하게 `getaddrinfo()` 를 사용하여 조회됩니다.
- *ssl_handshake_timeout* 은 (TLS 연결의 경우) 연결을 중단하기 전에 TLS 핸드 셰이크가 완료될 때까지 대기하는 시간(초)입니다. `None` (기본값) 이면 `60.0` 초가 사용됩니다.
- *ssl_shutdown_timeout* 은 연결을 중단하기 전에 SSL 종료가 완료될 때까지 대기하는 시간(초)입니다. `None` (기본값) 이면 `30.0` 초가 사용됩니다.
- *all_errors* determines what exceptions are raised when a connection cannot be created. By default, only a single `Exception` is raised: the first exception if there is only one or all errors have same message, or a

single `OSError` with the error messages combined. When `all_errors` is `True`, an `ExceptionGroup` will be raised containing all exceptions (even if there is only one).

버전 3.5에서 변경: `ProactorEventLoop`에 SSL/TLS에 대한 지원이 추가되었습니다.

버전 3.6에서 변경: 소켓 옵션 *socket.TCP_NODELAY*는 기본적으로 모든 TCP 연결에 대해 설정됩니다.

버전 3.7에서 변경: *ssl_handshake_timeout* 매개 변수를 추가했습니다.

버전 3.8에서 변경: *happy_eyeballs_delay*와 *interleave* 매개 변수가 추가되었습니다.

Happy Eyeballs 알고리즘: 듀얼 스택 호스트의 성공. 서버의 IPv4 경로와 프로토콜이 작동하지만, 서버의 IPv6 경로와 프로토콜은 작동하지 않을 때, 이중 스택 클라이언트 응용 프로그램은 IPv4 전용 클라이언트와 비교하여 상당한 연결 지연이 발생합니다. 이중 스택 클라이언트의 사용자 환경이 더 나빠지기 때문에 바람직하지 않습니다. 이 문서는 이런 사용자가 볼 수 있는 지연을 줄이고 알고리즘에 대한 요구 사항을 지정하고 알고리즘을 제공합니다.

자세한 정보: https://datatracker.ietf.org/doc/html/rfc6555

버전 3.11에서 변경: *ssl_shutdown_timeout* 매개 변수를 추가했습니다.

버전 3.12에서 변경: *all_errors* was added.

더 보기

`open_connection()` 함수는 고수준 대안 API입니다. async/await 코드에서 직접 사용할 수 있는 (`StreamReader`, `StreamWriter`) 쌍을 반환합니다.

async `loop.`**`create_datagram_endpoint`**(*protocol_factory, local_addr=None, remote_addr=None, *, family=0, proto=0, flags=0, reuse_port=None, allow_broadcast=None, sock=None*)

데이터 그램 연결을 만듭니다.

소켓 패밀리는 *host*(또는 주어지면 *family*)에 따라 `AF_INET`, `AF_INET6` 또는 `AF_UNIX`일 수 있습니다.

소켓 유형은 `SOCK_DGRAM`이 됩니다.

protocol_factory 는 반드시 프로토콜 구현을 반환하는 콜러블이어야 합니다.

성공하면 `(transport, protocol)` 튜플이 반환됩니다.

다른 인자들:

- *local_addr* 이 주어지면, 소켓을 로컬에 바인드 하는 데 사용되는 `(local_host, local_port)` 튜플입니다. *local_host* 와 *local_port* 는 `getaddrinfo()`를 사용하여 조회됩니다.

  참고

  On Windows, when using the proactor event loop with `local_addr=None`, an `OSError` with `errno.WSAEINVAL` will be raised when running it.

- *remote_addr* 이 주어지면, 소켓을 원격 주소에 연결하는 데 사용되는 `(remote_host, remote_port)` 튜플입니다. *remote_host* 와 *remote_port* 는 `getaddrinfo()`를 사용하여 조회됩니다.
- *family, proto, flags* 는 *host* 결정을 위해 `getaddrinfo()` 에 전달할 선택적 주소 패밀리, 프로토콜, 플래그입니다. 주어지면, 이것들은 모두 해당하는 `socket` 모듈 상수에 대응하는 정수여야 합니다.
- *reuse_port* 는 모두 만들 때 이 플래그를 설정하는 한, 이 말단이 다른 기존 말단이 바인드 된 것과 같은 포트에 바인드 되도록 허용하도록 커널에 알려줍니다. 이 옵션은 윈도우나 일부 유닉스에서는 지원되지 않습니다. *socket.SO_REUSEPORT* 상수가 정의되어 있지 않으면, 이 기능은 지원되지 않는 것입니다.

- *allow_broadcast* 는 이 말단이 브로드캐스트 주소로 메시지를 보낼 수 있도록 커널에 알립니다.
- *sock* 은 트랜스포트가 사용할 소켓 객체로, 기존의 이미 연결된 `socket.socket` 객체를 사용하기 위해 선택적으로 지정할 수 있습니다. 지정되면 *local_addr* 과 *remote_addr* 를 생략해야 합니다 (반드시 `None` 이어야 합니다).

> **참고**
>
> The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport' s `close()` method.

UDP 메아리 클라이언트 프로토콜 과 *UDP* 메아리 서버 프로토콜 예제를 참고하세요.

버전 3.4.4에서 변경: *family*, *proto*, *flags*, *reuse_address*, *reuse_port*, *allow_broadcast*, *sock* 매개 변수가 추가되었습니다.

버전 3.8에서 변경: 윈도우에 대한 지원이 추가되었습니다.

버전 3.8.1에서 변경: *reuse_address* 매개 변수는 더는 지원되지 않습니다, *socket.SO_REUSEADDR*를 사용하면 UDP에서 심각한 보안 문제가 발생하기 때문입니다. `reuse_address=True`를 명시적으로 전달하면 예외가 발생합니다.

UID가 다른 여러 프로세스가 `SO_REUSEADDR`를 사용하여 소켓을 같은 UDP 소켓 주소에 할당하면, 들어오는 패킷이 소켓 간에 무작위로 분산될 수 있습니다.

지원되는 플랫폼의 경우, *reuse_port*를 유사한 기능의 대체품으로 사용할 수 있습니다. *reuse_port*에서는 *socket.SO_REUSEPORT*가 대신 사용되는데, 다른 UID를 가진 프로세스가 같은 소켓 주소에 소켓을 할당하지 못하게 구체적으로 막습니다.

버전 3.11에서 변경: The *reuse_address* parameter, disabled since Python 3.8.1, 3.7.6 and 3.6.10, has been entirely removed.

async `loop.`**`create_unix_connection`**(*protocol_factory*, *path=None*, *, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

유닉스 연결을 만듭니다.

소켓 패밀리는 `AF_UNIX`가 됩니다; 소켓 유형은 `SOCK_STREAM`이 됩니다.

성공하면 `(transport, protocol)` 튜플이 반환됩니다.

path 는 유닉스 도메인 소켓의 이름이며, *sock* 매개 변수가 지정되지 않으면 필수입니다. 추상 유닉스 소켓, `str`, `bytes`, `Path` 경로가 지원됩니다.

이 메서드의 인자에 관한 정보는 `loop.create_connection()` 메서드의 설명서를 참조하십시오.

가용성: Unix.

버전 3.7에서 변경: *ssl_handshake_timeout* 매개 변수를 추가했습니다. *path* 매개 변수는 이제 경로류 객체 가 될 수 있습니다.

버전 3.11에서 변경: *ssl_shutdown_timeout* 매개 변수를 추가했습니다.

네트워크 서버 만들기

async `loop.`**`create_server`**(*protocol_factory*, *host=None*, *port=None*, *, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *keep_alive=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*, *start_serving=True*)

host 주소의 *port* 에서 리스닝하는 TCP 서버(소켓 유형 `SOCK_STREAM`)를 만듭니다.

`Server` 객체를 반환합니다.

인자:

- *protocol_factory* 는 반드시 프로토콜 구현을 반환하는 콜러블이어야 합니다.
- *host* 매개 변수는 서버가 리스닝할 위치를 결정하는 여러 형으로 설정할 수 있습니다.:
  - *host*가 문자열이면, TCP 서버는 *host*로 지정된 단일 네트워크 인터페이스에 바인딩 됩니다.
  - *host*가 문자열의 시퀀스면, TCP 서버는 시퀀스로 지정된 모든 네트워크 인터페이스에 바인딩 됩니다.
  - *host*가 빈 문자열이거나 `None`이면, 모든 인터페이스가 사용되는 것으로 가정하고, 여러 소켓의 리스트가 반환됩니다 (대체로 IPv4 하나와 IPv6 하나).
- The *port* parameter can be set to specify which port the server should listen on. If `0` or `None` (the default), a random unused port will be selected (note that if *host* resolves to multiple network interfaces, a different random port will be selected for each interface).
- *family* 는 `socket.AF_INET` 또는 `AF_INET6` 중 하나로 설정되어, 소켓이 IPv4 또는 IPv6을 사용하게 할 수 있습니다. 설정되지 않으면, *family* 는 호스트 이름에 의해 결정됩니다(기본값 `AF_UNSPEC`).
- *flags* 은 `getaddrinfo()`를 위한 비트 마스크입니다.
- *sock* 은 기존 소켓 객체를 사용하기 위해 선택적으로 지정할 수 있습니다. 지정되면, *host* 및 *port* 는 지정할 수 없습니다.

> **참고**
>
> The *sock* argument transfers ownership of the socket to the server created. To close the socket, call the server' s `close()` method.

- *backlog* 는 `listen()` 으로 전달되는 최대 대기 연결 수 입니다 (기본값은 100).
- *ssl* 을 `SSLContext` 인스턴스로 설정하면, 들어오는 연결에 TLS를 사용합니다.
- *reuse_address* 는, 일반적인 시간제한이 만료될 때까지 기다리지 않고, `TIME_WAIT` 상태의 로컬 소켓을 재사용하도록 커널에 알려줍니다. 지정하지 않으면 유닉스에서 자동으로 `True` 로 설정됩니다.
- *reuse_port* 는 모두 만들 때 이 플래그를 설정하는 한, 이 말단이 다른 기존 말단이 바인드 된 것과 같은 포트에 바인드 되도록 허용하도록 커널에 알려줍니다. 이 옵션은 윈도우에서 지원되지 않습니다.
- *keep_alive* set to `True` keeps connections active by enabling the periodic transmission of messages.

버전 3.13에서 변경: *keep_alive* 매개 변수가 추가되었습니다.

- *ssl_handshake_timeout* 은 (TLS 서버의 경우) 연결을 중단하기 전에 TLS 핸드 셰이크가 완료될 때까지 대기하는 시간(초)입니다. `None` (기본값) 이면 `60.0` 초가 사용됩니다.
- *ssl_shutdown_timeout* 은 연결을 중단하기 전에 SSL 종료가 완료될 때까지 대기하는 시간(초)입니다. `None` (기본값) 이면 `30.0` 초가 사용됩니다.
- *start_serving* 을 `True` (기본값) 로 설정하면, 생성된 서버가 즉시 연결을 받아들입니다. `False` 로 설정되면, 사용자는 서버가 연결을 받기 시작하도록 `Server.start_serving()` 이나 `Server.serve_forever()`를 await 해야 합니다.

버전 3.5에서 변경: `ProactorEventLoop`에 SSL/TLS에 대한 지원이 추가되었습니다.

버전 3.5.1에서 변경: *host* 매개 변수는 문자열의 시퀀스가 될 수 있습니다.

버전 3.6에서 변경: *ssl_handshake_timeout* 과 *start_serving* 매개 변수를 추가했습니다. 소켓 옵션 *socket.TCP_NODELAY*는 기본적으로 모든 TCP 연결에 대해 설정됩니다.

버전 3.11에서 변경: *ssl_shutdown_timeout* 매개 변수를 추가했습니다.

> **더 보기**
>
> `start_server()` 함수는 async/await 코드에서 사용할 수 있는 `StreamReader` 및 `StreamWriter` 쌍을 반환하는 고수준의 대체 API입니다.

async `loop.`**`create_unix_server`**(*protocol_factory, path=None, *, sock=None, backlog=100, ssl=None, ssl_handshake_timeout=None, ssl_shutdown_timeout=None, start_serving=True, cleanup_socket=True*)

`loop.create_server()`와 유사하지만, 소켓 패밀리 `AF_UNIX` 용입니다.

path 는 유닉스 도메인 소켓의 이름이며, *sock* 매개 변수가 제공되지 않으면 필수입니다. 추상 유닉스 소켓, `str`, `bytes`, `Path` 경로가 지원됩니다.

If *cleanup_socket* is true then the Unix socket will automatically be removed from the filesystem when the server is closed, unless the socket has been replaced after the server has been created.

이 메서드의 인자에 대한 정보는 `loop.create_server()` 메서드의 설명서를 참조하십시오.

가용성: Unix.

버전 3.7에서 변경: *ssl_handshake_timeout* 과 *start_serving* 매개 변수를 추가했습니다. *path* 매개 변수는 이제 `Path` 객체일 수 있습니다.

버전 3.11에서 변경: *ssl_shutdown_timeout* 매개 변수를 추가했습니다.

버전 3.13에서 변경: *cleanup_socket* 매개 변수가 추가되었습니다.

async `loop.`**`connect_accepted_socket`**(*protocol_factory, sock, *, ssl=None, ssl_handshake_timeout=None, ssl_shutdown_timeout=None*)

이미 받아들인 연결을 트랜스포트/프로토콜 쌍으로 래핑합니다.

이 메서드는 asyncio 밖에서 연결을 받아들이지만, 그 연결을 처리하는데 asyncio 를 사용하는 서버에서 사용됩니다.

매개 변수:

- *protocol_factory* 는 반드시 프로토콜 구현을 반환하는 콜러블이어야 합니다.
- *sock* 은 `socket.accept` 가 반환한 기존 소켓 객체입니다.

> **참고**
>
> The *sock* argument transfers ownership of the socket to the transport created. To close the socket, call the transport' s `close()` method.

- *ssl* 을 `SSLContext` 로 설정하면, 들어오는 연결에 SSL을 사용합니다.
- *ssl_handshake_timeout* 은 (SSL 연결의 경우) 연결을 중단하기 전에 SSL 핸드 셰이크가 완료될 때까지 대기하는 시간(초)입니다. `None` (기본값) 이면 `60.0` 초가 사용됩니다.
- *ssl_shutdown_timeout* 은 연결을 중단하기 전에 SSL 종료가 완료될 때까지 대기하는 시간(초)입니다. `None` (기본값) 이면 `30.0` 초가 사용됩니다.

`(transport, protocol)` 쌍을 반환합니다.

Added in version 3.5.3.

버전 3.7에서 변경: *ssl_handshake_timeout* 매개 변수를 추가했습니다.

버전 3.11에서 변경: *ssl_shutdown_timeout* 매개 변수를 추가했습니다.

파일 전송

async `loop.`**sendfile**(*transport*, *file*, *offset=0*, *count=None*, *, *fallback=True*)

file 을 *transport* 로 보냅니다. 전송된 총 바이트 수를 반환합니다.

이 메서드는 가능한 경우 고성능 `os.sendfile()` 을 사용합니다.

file 는 바이너리 모드로 열린 일반 파일 객체여야 합니다.

offset 은 파일 읽기 시작할 위치를 알려줍니다. *count* 를 제공하면, EOF에 도달할 때까지 파일을 보내는 대신, 전송할 총 바이트 수를 지정합니다. 파일의 위치가 갱신됩니다, 이 메서드가 에러를 일으킬 때조차. 그리고, `file.tell()` 는 실제 전송된 바이트 수를 얻는 데 사용될 수 있습니다.

fallback 을 `True` 로 설정하면, 플랫폼이 sendfile 시스템 호출을 지원하지 않을 때 (가령 유닉스에서 SSL 소켓을 사용하거나 윈도우인 경우), asyncio 가 파일을 수동으로 읽고 보내도록 합니다.

시스템이 *sendfile* 시스템 호출을 지원하지 않고 *fallback* 이 `False` 면 `SendfileNotAvailableError` 를 발생시킵니다.

Added in version 3.7.

TLS 업그레이드

async `loop.`**start_tls**(*transport*, *protocol*, *sslcontext*, *, *server_side=False*, *server_hostname=None*, *ssl_handshake_timeout=None*, *ssl_shutdown_timeout=None*)

기존 트랜스포트 기반 연결을 TLS로 업그레이드합니다.

Create a TLS coder/decoder instance and insert it between the *transport* and the *protocol*. The coder/decoder implements both *transport*-facing protocol and *protocol*-facing transport.

Return the created two-interface instance. After *await*, the *protocol* must stop using the original *transport* and communicate with the returned object only because the coder caches *protocol*-side data and sporadically exchanges extra TLS session packets with *transport*.

In some situations (e.g. when the passed transport is already closing) this may return `None`.

매개 변수:

- `create_server()` 와 `create_connection()` 같은 메서드가 반환하는 *transport* 와 *protocol* 인스턴스.
- *sslcontext*: 구성된 `SSLContext` 의 인스턴스.
- (`create_server()` 에 의해 생성된 것과 같은) 서버 측 연결이 업그레이드될 때 *server_side* 에 `True` 를 전달합니다.
- *server_hostname*: 대상 서버의 인증서가 일치될 호스트 이름을 설정하거나 대체합니다.
- *ssl_handshake_timeout* 은 (TLS 연결의 경우) 연결을 중단하기 전에 TLS 핸드 셰이크가 완료될 때까지 대기하는 시간(초)입니다. `None` (기본값) 이면 `60.0` 초가 사용됩니다.
- *ssl_shutdown_timeout* 은 연결을 중단하기 전에 SSL 종료가 완료될 때까지 대기하는 시간(초)입니다. `None` (기본값) 이면 `30.0` 초가 사용됩니다.

Added in version 3.7.

버전 3.11에서 변경: *ssl_shutdown_timeout* 매개 변수를 추가했습니다.

파일 기술자 관찰하기

`loop.`**add_reader**(*fd*, *callback*, **args*)

fd 파일 기술자가 읽기 가능한지 관찰하기 시작하고, 일단 *fd* 가 읽기 가능해지면 지정한 인자로 *callback* 을 호출합니다.

Any preexisting callback registered for *fd* is cancelled and replaced by *callback*.

loop.**remove_reader**(*fd*)

fd 파일 기술자가 읽기 가능한지 관찰하는 것을 중단합니다. *fd*가 이전에 읽기에 대해 관찰 중이었으면 `True`를 반환합니다.

loop.**add_writer**(*fd*, *callback*, **args*)

fd 파일 기술자가 쓰기 가능한지 관찰하기 시작하고, 일단 *fd*가 쓰기 가능해지면 지정한 인자로 *callback* 을 호출합니다.

Any preexisting callback registered for *fd* is cancelled and replaced by *callback*.

callback 에 키워드 인자를 전달하려면 `functools.partial()`를 사용하십시오.

loop.**remove_writer**(*fd*)

fd 파일 기술자가 쓰기 가능한지 관찰하는 것을 중단합니다. *fd*가 이전에 쓰기에 대해 관찰 중이었으면 `True`를 반환합니다.

이 메서드의 일부 제한 사항은 플랫폼 지원 절을 참조하십시오.

소켓 객체로 직접 작업하기

일반적으로 `loop.create_connection()` 및 `loop.create_server()`와 같은 트랜스포트 기반 API를 사용하는 프로토콜 구현은 소켓을 직접 사용하는 구현보다 빠릅니다. 그러나, 성능이 결정적이지 않고 `socket` 객체로 직접 작업하는 것이 더 편리한 사용 사례가 있습니다.

async loop.**sock_recv**(*sock*, *nbytes*)

sock 에서 최대 *nbytes* 를 수신합니다. `socket.recv()` 의 비동기 버전.

수신한 데이터를 바이트열 객체로 반환합니다.

sock 은 반드시 비 블로킹 소켓이어야 합니다.

버전 3.7에서 변경: 이 메서드가 항상 코루틴 메서드라고 설명되어왔지만, 파이썬 3.7 이전에는 `Future`를 반환했습니다. 파이썬 3.7부터, 이것은 `async def` 메서드입니다.

async loop.**sock_recv_into**(*sock*, *buf*)

sock 에서 *buf* 버퍼로 데이터를 수신합니다. 블로킹 `socket.recv_into()` 메서드를 따라 만들어졌습니다.

버퍼에 기록된 바이트 수를 돌려줍니다.

sock 은 반드시 비 블로킹 소켓이어야 합니다.

Added in version 3.7.

async loop.**sock_recvfrom**(*sock*, *bufsize*)

sock 에서 최대 *bufsize* 크기의 데이터그램을 수신합니다. `socket.recvfrom()` 의 비동기 버전.

(수신된 데이터, 원격 주소) 튜플을 반환합니다.

sock 은 반드시 비 블로킹 소켓이어야 합니다.

Added in version 3.11.

async loop.**sock_recvfrom_into**(*sock*, *buf*, *nbytes=0*)

sock 에서 최대 *nbytes* 크기의 데이터그램을 *buf*로 수신합니다. `socket.recvfrom_into()` 의 비동기 버전.

(수신된 바이트 수, 원격 주소) 튜플을 반환합니다.

sock 은 반드시 비 블로킹 소켓이어야 합니다.

Added in version 3.11.

async loop.**sock_sendall**(*sock*, *data*)

data 를 *sock* 소켓으로 보냅니다. socket.sendall() 의 비동기 버전.

이 메서드는 *data* 의 모든 데이터가 송신되거나 에러가 발생할 때까지 소켓으로 계속 송신합니다. 성공하면 None 이 반환됩니다. 에러가 발생하면 예외가 발생합니다. 또한, 연결의 수신 단에서 성공적으로 처리한 (있기는 하다면) 데이터의 크기를 확인하는 방법은 없습니다.

sock 은 반드시 비 블로킹 소켓이어야 합니다.

버전 3.7에서 변경: 이 메서드가 항상 코루틴 메서드라고 설명되어왔지만, 파이썬 3.7 이전에는 Future를 반환했습니다. 파이썬 3.7부터, 이것은 async def 메서드입니다.

async loop.**sock_sendto**(*sock*, *data*, *address*)

sock 에서 데이터그램을 *address* 로 보냅니다. socket.sendto() 의 비동기 버전.

보낸 바이트 수를 돌려줍니다.

sock 은 반드시 비 블로킹 소켓이어야 합니다.

Added in version 3.11.

async loop.**sock_connect**(*sock*, *address*)

sock 을 *address* 에 있는 원격 소켓에 연결합니다.

socket.connect() 의 비동기 버전.

sock 은 반드시 비 블로킹 소켓이어야 합니다.

버전 3.5.2에서 변경: address 는 더는 결정될 필요가 없습니다. sock_connect 는 socket.inet_pton() 을 호출하여 *address* 가 이미 결정되었는지를 검사합니다. 그렇지 않으면, loop.getaddrinfo() 가 *address* 를 결정하는 데 사용됩니다.

> **더 보기**
>
> loop.create_connection() 과 asyncio.open_connection().

async loop.**sock_accept**(*sock*)

연결을 받아들입니다. 블로킹 socket.accept() 메서드를 따라 만들어졌습니다.

소켓은 주소에 바인드 되어 연결을 리스닝해야 합니다. 반환 값은 (conn, address) 쌍인데, *conn* 은 연결로 데이터를 주고받을 수 있는 새 소켓 객체이고, *address* 는 연결의 반대편 끝의 소켓에 바인드된 주소입니다.

sock 은 반드시 비 블로킹 소켓이어야 합니다.

버전 3.7에서 변경: 이 메서드가 항상 코루틴 메서드라고 설명되어왔지만, 파이썬 3.7 이전에는 Future를 반환했습니다. 파이썬 3.7부터, 이것은 async def 메서드입니다.

> **더 보기**
>
> loop.create_server() 와 start_server().

async loop.**sock_sendfile**(*sock*, *file*, *offset=0*, *count=None*, **, fallback=True*)

가능하면 고성능 os.sendfile 을 사용하여 파일을 보냅니다. 전송된 총 바이트 수를 반환합니다.

socket.sendfile() 의 비동기 버전.

sock 은 반드시 비 블로킹 socket.SOCK_STREAM socket 이어야 합니다.

file 는 바이너리 모드로 열린 일반 파일 객체여야 합니다.

offset 은 파일 읽기 시작할 위치를 알려줍니다. *count* 를 제공하면, EOF에 도달할 때까지 파일을 보내는 대신, 전송할 총 바이트 수를 지정합니다. 파일의 위치가 갱신됩니다, 이 메서드가 에러를 일으킬 때조차. 그리고, file.tell() 는 실제 전송된 바이트 수를 얻는 데 사용될 수 있습니다.

fallback 을 `True` 로 설정하면, 플랫폼이 sendfile 시스템 호출을 지원하지 않을 때 (가령 유닉스에서 SSL 소켓을 사용하거나 윈도우인 경우), asyncio 가 파일을 수동으로 읽고 보내도록 합니다.

시스템이 *sendfile* 시스템 호출을 지원하지 않고 *fallback* 이 `False` 면 `SendfileNotAvailableError` 를 발생시킵니다.

sock 은 반드시 비 블로킹 소켓이어야 합니다.

Added in version 3.7.

DNS

async `loop.`**`getaddrinfo`**(*host, port, *, family=0, type=0, proto=0, flags=0*)

`socket.getaddrinfo()` 의 비동기 버전.

async `loop.`**`getnameinfo`**(*sockaddr, flags=0*)

`socket.getnameinfo()` 의 비동기 버전.

> **참고**
>
> Both *getaddrinfo* and *getnameinfo* internally utilize their synchronous versions through the loop' s default thread pool executor. When this executor is saturated, these methods may experience delays, which higher-level networking libraries may report as increased timeouts. To mitigate this, consider using a custom executor for other user tasks, or setting a default executor with a larger number of workers.

버전 3.7에서 변경: *getaddrinfo* 와 *getnameinfo* 메서드는 모두 코루틴 메서드라고 설명되어왔지만, 파이썬 3.7 이전에 실제로는 `asyncio.Future` 객체를 반환했습니다. 파이썬 3.7부터 두 가지 메서드 모두 코루틴입니다.

파이프로 작업하기

async `loop.`**`connect_read_pipe`**(*protocol_factory, pipe*)

이벤트 루프에 *pipe*의 읽기용 끝을 등록합니다.

protocol_factory 는 반드시 *asyncio* 프로토콜 구현을 반환하는 콜러블이어야 합니다.

*pipe*는 파일류 객체입니다.

쌍 `(transport, protocol)` 를 반환합니다. 여기서 *transport*는 `ReadTransport` 인터페이스를 지원하고, *protocol*은 *protocol_factory*에 의해 인스턴스로 만들어진 객체입니다.

`SelectorEventLoop` 이벤트 루프를 사용하면, *pipe* 는 비 블로킹 모드로 설정됩니다.

async `loop.`**`connect_write_pipe`**(*protocol_factory, pipe*)

이벤트 루프에 *pipe*의 쓰기용 끝을 등록합니다.

protocol_factory 는 반드시 *asyncio* 프로토콜 구현을 반환하는 콜러블이어야 합니다.

*pipe*는 파일류 객체입니다.

쌍 `(transport, protocol)` 를 반환합니다. 여기서 *transport*는 `WriteTransport` 인터페이스를 지원하고, *protocol*은 *protocol_factory*에 의해 인스턴스로 만들어진 객체입니다.

`SelectorEventLoop` 이벤트 루프를 사용하면, *pipe* 는 비 블로킹 모드로 설정됩니다.

> **참고**
>
> 윈도우에서 `SelectorEventLoop`는 위의 메서드들을 지원하지 않습니다. 윈도우에서는 대신 `ProactorEventLoop`를 사용하십시오.

더 보기

loop.subprocess_exec() 와 *loop.subprocess_shell()* 메서드.

유닉스 시그널

loop.**add_signal_handler**(*signum*, *callback*, **args*)

*callback*을 *signum* 시그널의 처리기로 설정합니다.

콜백은 다른 대기 중인 콜백과 해당 이벤트 루프의 실행 가능한 코루틴과 함께 *loop*에 의해 호출됩니다. *signal.signal()*을 사용하여 등록된 시그널 처리기와 달리, 이 함수로 등록된 콜백은 이벤트 루프와 상호 작용할 수 있습니다.

시그널 번호가 유효하지 않거나 잡을 수 없으면 *ValueError* 를 발생시킵니다. 처리기를 설정하는데 문제가 있는 경우 *RuntimeError* 를 발생시킵니다.

callback 에 키워드 인자를 전달하려면 *functools.partial()*를 사용하십시오.

*signal.signal()*와 마찬가지로, 이 함수는 메인 스레드에서 호출되어야 합니다.

loop.**remove_signal_handler**(*sig*)

sig 시그널의 처리기를 제거합니다.

시그널 처리기가 제거되면 `True` 를, 주어진 시그널에 처리기가 설정되지 않았으면 `False` 를 반환합니다.

가용성: Unix.

더 보기

signal 모듈.

스레드 또는 프로세스 풀에서 코드를 실행하기

awaitable loop.**run_in_executor**(*executor*, *func*, **args*)

지정된 실행기에서 *func* 가 호출되도록 배치합니다.

The *executor* argument should be an *concurrent.futures.Executor* instance. The default executor is used if *executor* is `None`. The default executor can be set by *loop.set_default_executor()*, otherwise, a *concurrent.futures.ThreadPoolExecutor* will be lazy-initialized and used by *run_in_executor()* if needed.

예:

```
import asyncio
import concurrent.futures

def blocking_io():
    # File operations (such as logging) can block the
    # event loop: run them in a thread pool.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    # CPU-bound operations will block the event loop:
    # in general it is preferable to run them in a
    # process pool.
    return sum(i * i for i in range(10 ** 7))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
async def main():
    loop = asyncio.get_running_loop()

    ## Options:

    # 1. Run in the default loop's executor:
    result = await loop.run_in_executor(
        None, blocking_io)
    print('default thread pool', result)

    # 2. Run in a custom thread pool:
    with concurrent.futures.ThreadPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, blocking_io)
        print('custom thread pool', result)

    # 3. Run in a custom process pool:
    with concurrent.futures.ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom process pool', result)

    # 4. Run in a custom interpreter pool:
    with concurrent.futures.InterpreterPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom interpreter pool', result)

if __name__ == '__main__':
    asyncio.run(main())
```

Note that the entry point guard (`if __name__ == '__main__'`) is required for option 3 due to the peculiarities of `multiprocessing`, which is used by `ProcessPoolExecutor`. See *Safe importing of main module*.

이 메서드는 `asyncio.Future` 객체를 반환합니다.

func 에 키워드 인자를 전달하려면 `functools.partial()`를 사용하십시오.

버전 3.5.3 에서 변경: `loop.run_in_executor()` 는 더는 자신이 만드는 스레드 풀 실행기의 `max_workers` 를 설정하지 않습니다. 대신 스레드 풀 실행기(`ThreadPoolExecutor`)가 스스로 기본값을 설정하도록 합니다.

`loop.`**`set_default_executor`**(*executor*)

Set *executor* as the default executor used by `run_in_executor()`. *executor* must be an instance of `ThreadPoolExecutor`, which includes `InterpreterPoolExecutor`.

버전 3.11 에서 변경: *executor* 는 반드시 `ThreadPoolExecutor`의 인스턴스여야 합니다.

에러 처리 API

이벤트 루프에서 예외를 처리하는 방법을 사용자 정의 할 수 있습니다.

`loop.`**`set_exception_handler`**(*handler*)

handler 를 새 이벤트 루프 예외 처리기로 설정합니다.

handler 가 `None` 이면, 기본 예외 처리기가 설정됩니다. 그렇지 않으면, *handler* 는 반드시 `(loop, context)` 와 일치하는 서명을 가진 콜러블이어야 합니다. 여기서 `loop` 는 활성 이벤트 루프에 대한 참조가 될 것이고, `context` 는 예외에 관한 세부 정보를 담고 있는 `dict` 객체가 됩니다 (context 에 대한 자세한 내용은 `call_exception_handler()` 문서를 참조하십시오).

If the handler is called on behalf of a `Task` or `Handle`, it is run in the `contextvars.Context` of that task or callback handle.

버전 3.12에서 변경: The handler may be called in the `Context` of the task or handle where the exception originated.

`loop.get_exception_handler()`

현재 예외 처리기를 반환하거나, 사용자 정의 예외 처리기가 설정되지 않았으면 `None` 을 반환합니다.

Added in version 3.5.2.

`loop.default_exception_handler(context)`

기본 예외 처리기.

예외가 발생하고 예외 처리기가 설정되지 않았을 때 호출됩니다. 기본 동작으로 위임하려는 사용자 정의 예외 처리기가 호출할 수 있습니다.

context 매개 변수는 `call_exception_handler()` 에서와 같은 의미입니다.

`loop.call_exception_handler(context)`

현재 이벤트 루프 예외 처리기를 호출합니다.

context 는 다음 키를 포함하는 `dict` 객체입니다 (새 키가 미래의 파이썬 버전에서 추가될 수 있습니다):

- 'message' : 에러 메시지;
- 'exception' (선택적): 예외 객체;
- 'future' (선택적): `asyncio.Future` 인스턴스;
- 'task' (선택적): `asyncio.Task` 인스턴스;
- 'handle' (선택적): `asyncio.Handle` 인스턴스;
- 'protocol' (선택적): 프로토콜 인스턴스;
- 'transport' (선택적): 트랜스포트 인스턴스;
- 'socket' (선택적): `socket.socket` 인스턴스;
- 'source_traceback' (optional): Traceback of the source;
- 'handle_traceback' (optional): Traceback of the handle;
- **'asyncgen' (optional): Asynchronous generator that caused**
  예외.

> **참고**
>
> 이 메서드는 서브 클래스 된 이벤트 루프에서 재정의되지 않아야 합니다. 사용자 정의 예외 처리를 위해서는 `set_exception_handler()` 메서드를 사용하십시오.

디버그 모드 활성화

`loop.get_debug()`

이벤트 루프의 디버그 모드(`bool`)를 가져옵니다.

기본값은 환경 변수 `PYTHONASYNCIODEBUG` 가 비어 있지 않은 문자열로 설정되면 `True` 이고, 그렇지 않으면 `False` 입니다.

`loop.set_debug(enabled: bool)`

이벤트 루프의 디버그 모드를 설정합니다.

버전 3.7에서 변경: 이제 새로운 파이썬 개발 모드를 사용하여 디버그 모드를 활성화할 수도 있습니다.

loop.**slow_callback_duration**

This attribute can be used to set the minimum execution duration in seconds that is considered "slow". When debug mode is enabled, "slow" callbacks are logged.

Default value is 100 milliseconds.

> **더 보기**
>
> *asyncio*의 디버그 모드.

자식 프로세스 실행하기

이 하위 절에서 설명하는 메서드는 저수준입니다. 일반적인 async/await 코드에서는 대신 고수준의 `asyncio.create_subprocess_shell()` 및 `asyncio.create_subprocess_exec()` 편리 함수를 사용하는 것을 고려하십시오.

> **참고**
>
> 윈도우에서, 기본 이벤트 루프 `ProactorEventLoop`는 자식 프로세스를 지원합니다, 반면에 `SelectorEventLoop`는 지원하지 않습니다. 자세한 내용은 윈도우에서의 자식 프로세스 지원을 참조하십시오.

async loop.**subprocess_exec**(*protocol_factory, *args, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE, **kwargs*)

*args*로 지정된 하나 이상의 문자열 인자로 서브 프로세스를 만듭니다.

*args*는 반드시 다음과 같은 것으로 표현되는 문자열의 목록이어야 합니다:

- `str`;
- 또는 파일 시스템 인코딩으로로 인코딩된 `bytes`.

첫 번째 문자열은 프로그램 실행 파일을 지정하고, 나머지 문자열은 인자를 지정합니다. 함께, 문자열 인자들은 프로그램의 `argv`를 구성합니다.

이것은 `shell=False`와 문자열의 목록을 첫 번째 인자로 호출된 표준 라이브러리 `subprocess.Popen` 클래스와 유사합니다. 그러나 `Popen`이 문자열 목록인 단일 인자를 받아들이지만, *subprocess_exec*는 여러 문자열 인자를 받아들입니다.

*protocol_factory*는 반드시 `asyncio.SubprocessProtocol` 클래스의 서브 클래스를 반환하는 콜러블이어야 합니다.

다른 매개 변수:

- *stdin*은 다음 중 하나일 수 있습니다:
  - a file-like object
  - an existing file descriptor (a positive integer), for example those created with `os.pipe()`
  - `subprocess.PIPE` 상수 (기본값), 새 파이프를 만들고 연결합니다,
  - 서브 프로세스가 이 프로세스의 파일 기술자를 상속하게 하는 값 `None`
  - 특수 `os.devnull` 파일이 사용될 것임을 나타내는 `subprocess.DEVNULL` 상수
- *stdout*은 다음 중 하나일 수 있습니다:
  - a file-like object
  - `subprocess.PIPE` 상수 (기본값), 새 파이프를 만들고 연결합니다,
  - 서브 프로세스가 이 프로세스의 파일 기술자를 상속하게 하는 값 `None`
  - 특수 `os.devnull` 파일이 사용될 것임을 나타내는 `subprocess.DEVNULL` 상수

- *stderr*은 다음 중 하나일 수 있습니다:
  - a file-like object
  - `subprocess.PIPE` 상수 (기본값), 새 파이프를 만들고 연결합니다,
  - 서브 프로세스가 이 프로세스의 파일 기술자를 상속하게 하는 값 `None`
  - 특수 `os.devnull` 파일이 사용될 것임을 나타내는 `subprocess.DEVNULL` 상수
  - `subprocess.STDOUT` 상수, 표준 에러 스트림을 프로세스의 표준 출력 스트림에 연결합니다
- 다른 모든 키워드 인자는 해석 없이 `subprocess.Popen`로 전달됩니다. 다만, *bufsize*, *universal_newlines*, *shell*, *text*, *encoding* 및 *errors*는 예외인데, 이것들은 지정되지 않아야 합니다.

  `asyncio` 서브 프로세스 API는 스트림을 텍스트로 디코딩하는 것을 지원하지 않습니다. `bytes.decode()`는 스트림에서 반환된 바이트열을 텍스트로 변환하는 데 사용될 수 있습니다.

If a file-like object passed as *stdin*, *stdout* or *stderr* represents a pipe, then the other side of this pipe should be registered with `connect_write_pipe()` or `connect_read_pipe()` for use with the event loop.

다른 인자에 관한 설명은 `subprocess.Popen` 클래스의 생성자를 참조하십시오.

`(transport, protocol)` 쌍을 반환합니다. 여기에서 *transport*는 `asyncio.SubprocessTransport` 베이스 클래스를 따르고, *protocol*은 *protocol_factory*에 의해 인스턴스로 만들어진 객체입니다.

async `loop.`**subprocess_shell**(*protocol_factory, cmd, *, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE, **kwargs*)

플랫폼의 "셸" 구문을 사용하는 *cmd*로 자식 프로세스를 만듭니다. *cmd*는 `str`이나 파일 시스템 인코딩으로 인코딩된 `bytes` 일 수 있습니다.

이것은 `shell=True`로 호출된 표준 라이브러리 `subprocess.Popen` 클래스와 유사합니다.

*protocol_factory*는 반드시 `SubprocessProtocol` 클래스의 서브 클래스를 반환하는 콜러블이어야 합니다.

나머지 인자에 관한 자세한 내용은 `subprocess_exec()`를 참조하십시오.

`(transport, protocol)` 쌍을 반환합니다. 여기서 *transport*는 `SubprocessTransport` 베이스 클래스를 따르고, *protocol*은 *protocol_factory*에 의해 인스턴스로 만들어진 객체입니다.

> **참고**
>
> 셸 주입 취약점을 피하고자 모든 공백과 특수 문자를 적절하게 따옴표 처리하는 것은 응용 프로그램의 책임입니다. `shlex.quote()` 함수를 사용하여 셸 명령을 구성하는 데 사용될 문자열에 있는 공백 및 특수 문자를 올바르게 이스케이프 할 수 있습니다.

콜백 핸들

class `asyncio.`**Handle**

`loop.call_soon()`, `loop.call_soon_threadsafe()` 에 의해 반환되는 콜백 래퍼 객체.

get_context()

Return the `contextvars.Context` object associated with the handle.

Added in version 3.12.

cancel()

콜백을 취소합니다. 콜백이 이미 취소되었거나 실행되었다면 이 메서드는 아무 효과가 없습니다.

cancelled()

콜백이 취소되었으면 `True` 을 반환합니다.

Added in version 3.7.

class asyncio.**TimerHandle**

loop.call_later() 및 *loop.call_at()* 에 의해 반환되는 콜백 래퍼 객체.

이 클래스는 *Handle*의 서브 클래스입니다.

when()

예약된 콜백 시간을 *float* 초로 반환합니다.

시간은 절대 타임스탬프입니다. *loop.time()* 과 같은 시간 참조를 사용합니다.

Added in version 3.7.

서버 객체

Server 객체는 *loop.create_server()*, *loop.create_unix_server()*, *start_server()*, *start_unix_server()*로 만듭니다.

Server 클래스의 인스턴스를 직접 만들지 마십시오.

class asyncio.**Server**

Server 객체는 비동기 컨텍스트 관리자입니다. `async with` 문에서 사용될 때, `async with` 문이 완료되면 서버 객체가 닫혀 있고 새 연결을 받아들이지 않는다는 것이 보장됩니다:

```
srv = await loop.create_server(...)

async with srv:
    # 코드

# 이 지점에서, srv 는 닫혔고 더는 새 연결을 받아들이지 않습니다.
```

버전 3.7에서 변경: Server 객체는 파이썬 3.7부터 비동기 컨텍스트 관리자입니다.

버전 3.11에서 변경: This class was exposed publicly as `asyncio.Server` in Python 3.9.11, 3.10.3 and 3.11.

close()

서버를 중지합니다: 리스닝 소켓을 닫고 *sockets* 어트리뷰트를 `None` 으로 설정합니다.

이미 받아들여진 클라이언트 연결을 나타내는 소켓은 열린 채로 있습니다.

서버는 비동기적으로 닫힙니다; 서버가 닫힐 때까지 (그리고 더는 활성 연결이 남아있지 않을 때까지) 대기하려면 *wait_closed()* 코루틴을 사용하십시오.

close_clients()

모든 이미 받아들여진 클라이언트 연결을 닫습니다.

Calls *close()* on all associated transports.

close() should be called before *close_clients()* when closing the server to avoid races with new clients connecting.

Added in version 3.13.

abort_clients()

Close all existing incoming client connections immediately, without waiting for pending operations to complete.

Calls *abort()* on all associated transports.

close() should be called before *abort_clients()* when closing the server to avoid races with new clients connecting.

Added in version 3.13.

get_loop()

서버 객체와 연관된 이벤트 루프를 반환합니다.

Added in version 3.7.

async start_serving()

연결을 받아들이기 시작합니다.

이 메서드는 멱등적이라서, 서버가 이미 시작되었을 때도 호출 할 수 있습니다.

*loop.create_server()*와 *asyncio.start_server()* 의 *start_serving* 키워드 전용 매개 변수는 즉시 연결을 받아들이지 않는 서버 객체를 만들 수 있도록 합니다. 이 경우 `Server.start_serving()`, 또는 *Server.serve_forever()*를 사용하여 Server가 연결을 받아들이기 시작하도록 할 수 있습니다.

Added in version 3.7.

async serve_forever()

코루틴이 취소될 때까지 연결을 받아들이기 시작합니다. `serve_forever` 태스크를 취소하면 서버가 닫힙니다.

이 메서드는 서버가 이미 연결을 받아들이고 있어도 호출 할 수 있습니다. 하나의 *Server* 객체 당 하나의 `serve_forever` 태스크만 존재할 수 있습니다.

예:

```
async def client_connected(reader, writer):
    # reader/writer 스트림으로 클라이언트와
    # 통신합니다. 예를 들어:
    await reader.readline()

async def main(host, port):
    srv = await asyncio.start_server(
        client_connected, host, port)
    await srv.serve_forever()

asyncio.run(main('127.0.0.1', 0))
```

Added in version 3.7.

is_serving()

서버가 새 연결을 받아들이고 있으면 `True` 를 반환합니다.

Added in version 3.7.

async wait_closed()

close() 메서드가 완료되고 모든 활성 연결이 종료될 때까지 기다립니다.

sockets

서버가 리스닝하고 있는 소켓류 객체 `asyncio.trsock.TransportSocket`의 리스트.

버전 3.7에서 변경: 파이썬 3.7 이전에는 `Server.sockets` 가 서버 소켓의 내부 리스트를 직접 반환했습니다. 3.7에서는 그 리스트의 복사본이 반환됩니다.

이벤트 루프 구현

asyncio에는 두 가지 이벤트 루프 구현이 함께 제공됩니다: *SelectorEventLoop* 및 *ProactorEventLoop*.

기본적으로 asyncio는 *EventLoop*를 사용하도록 구성됩니다.

class asyncio.**SelectorEventLoop**

selectors 모듈을 기반으로 하는 *AbstractEventLoop* 의 서브 클래스.

주어진 플랫폼에서 사용할 수 있는 가장 효율적인 *selector*를 사용합니다. 정확한 셀렉터 구현을 수동으로 구성하여 사용할 수도 있습니다.:

```
import asyncio
import selectors

async def main():
    ...

loop_factory = lambda: asyncio.SelectorEventLoop(selectors.SelectSelector())
asyncio.run(main(), loop_factory=loop_factory)
```

가용성: Unix, Windows.

class asyncio.**ProactorEventLoop**

"I/O 완료 포트" (IOCP)를 사용하는 윈도우용 *AbstractEventLoop*의 서브 클래스.

가용성: Windows.

> **더 보기**
>
> I/O 완료 포트에 관한 MSDN 설명서.

class asyncio.**EventLoop**

An alias to the most efficient available subclass of *AbstractEventLoop* for the given platform.

유닉스에서는 *SelectorEventLoop*의, 윈도우에서는 *ProactorEventLoop*의 별칭입니다.

Added in version 3.13.

class asyncio.**AbstractEventLoop**

asyncio 호환 이벤트 루프의 추상 베이스 클래스.

이벤트 루프 메서드 절은 AbstractEventLoop의 다른 구현이 정의해야 하는 모든 메서드를 나열합니다.

예제

이 절의 모든 예는 **의도적으로** *loop.run_forever()* 및 *loop.call_soon()*와 같은 저수준 이벤트 루프 API를 사용하는 방법을 보여줍니다. 현대 asyncio 응용 프로그램은 거의 이런 식으로 작성할 필요가 없습니다; *asyncio.run()*과 같은 고수준 함수를 사용하는 것을 고려하십시오.

call_soon()을 사용하는 Hello World

콜백을 예약하기 위해 *loop.call_soon()* 메서드를 사용하는 예제. 콜백은 "Hello World" 를 표시한 다음 이벤트 루프를 중지합니다:

```
import asyncio

def hello_world(loop):
    """A callback to print 'Hello World' and stop the event loop"""
    print('Hello World')
    loop.stop()

loop = asyncio.new_event_loop()

# hello_world() 호출을 예약합니다
loop.call_soon(hello_world, loop)

# 블로킹 호출이 loop.stop() 에 의해 중단됩니다
try:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    loop.run_forever()
finally:
    loop.close()
```

더 보기

코루틴과 `run()` 함수로 작성된 유사한 *Hello World* 예제.

call_later()로 현재 날짜를 표시합니다.

초마다 현재 날짜를 표시하는 콜백의 예입니다. 콜백은 `loop.call_later()` 메서드를 사용하여 5초 동안 자신을 다시 예약한 다음 이벤트 루프를 중지합니다:

```
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.new_event_loop()

# 첫번째 display_date() 호출을 예약합니다
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# 블로킹 호출이 loop.stop() 에 의해 중단됩니다
try:
    loop.run_forever()
finally:
    loop.close()
```

더 보기

코루틴과 `run()` 함수로 작성된 유사한 현재 날짜 예제.

파일 기술자에서 읽기 이벤트를 관찰하기

`loop.add_reader()` 메서드를 사용하여 파일 기술자가 데이터를 수신할 때까지 기다렸다가 이벤트 루프를 닫습니다:

```
import asyncio
from socket import socketpair

# 연결된 파일 기술자 쌍을 만듭니다
rsock, wsock = socketpair()

loop = asyncio.new_event_loop()

def reader():
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    data = rsock.recv(100)
    print("Received:", data.decode())

    # 할 일을 끝냈습니다: 파일 기술자를 등록 취소합니다
    loop.remove_reader(rsock)

    # 이벤트 루프를 중지합니다
    loop.stop()

# 읽기 이벤트를 위해 파일 기술자를 등록합니다
loop.add_reader(rsock, reader)

# 네트웍으로 부터의 데이터 수신을 흉내냅니다
loop.call_soon(wsock.send, 'abc'.encode())

try:
    # 이벤트 루프를 실행합니다
    loop.run_forever()
finally:
    # 할 일을 끝냈습니다, 소켓과 이벤트 루프를 닫습니다
    rsock.close()
    wsock.close()
    loop.close()
```

더 보기

- 트랜스포트, 프로토콜, *loop.create_connection()* 메서드를 사용한 유사한 예제.
- 고수준의 *asyncio.open_connection()* 함수와 스트림을 사용하는 또 다른 유사한 예제.

SIGINT 및 SIGTERM에 대한 시그널 처리기 설정

(이 `signals` 예제는 유닉스에서만 작동합니다.)

loop.add_signal_handler() 메서드를 사용하여 *SIGINT*와 *SIGTERM* 시그널을 위한 처리기를 등록합니다:

```
import asyncio
import functools
import os
import signal

def ask_exit(signame, loop):
    print("got signal %s: exit" % signame)
    loop.stop()

async def main():
    loop = asyncio.get_running_loop()

    for signame in {'SIGINT', 'SIGTERM'}:
        loop.add_signal_handler(
            getattr(signal, signame),
            functools.partial(ask_exit, signame, loop))

    await asyncio.sleep(3600)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
print("Event loop running for 1 hour, press Ctrl+C to interrupt.")
print(f"pid {os.getpid()}: send SIGINT or SIGTERM to exit.")

asyncio.run(main())
```

19.1.10 퓨처

소스 코드: Lib/asyncio/futures.py, Lib/asyncio/base_futures.py

Future 객체는 **저수준 콜백 기반 코드**와 고수준 async/await 코드 간에 다리를 놓는 데 사용됩니다.

퓨처 함수

asyncio.**isfuture**(*obj*)

*obj*가 다음 중 하나면 `True`를 반환합니다:

- `asyncio.Future`의 인스턴스,
- `asyncio.Task`의 인스턴스,
- `_asyncio_future_blocking` 어트리뷰트를 가진 퓨처류 객체.

Added in version 3.5.

asyncio.**ensure_future**(*obj*, *, *loop=None*)

다음을 반환합니다:

- *obj*가 `Future`, `Task` 또는 퓨처류 객체면, *obj* 인자를 있는 그대로 (`isfuture()`로 검사합니다.)
- *obj*가 코루틴이면, *obj*를 감싸는 `Task` 객체 (`iscoroutine()`로 검사합니다); 이 경우 코루틴은 `ensure_future()`로 예약됩니다.
- *obj*가 어웨이터블이면, *obj*를 기다릴 `Task` 객체 (`inspect.isawaitable()`로 검사합니다.)

*obj*가 이 중 어느 것도 아니면, `TypeError`가 발생합니다.

> **중요**
>
> Save a reference to the result of this function, to avoid a task disappearing mid-execution.
>
> See also the `create_task()` function which is the preferred way for creating new tasks or use `asyncio.TaskGroup` which keeps reference to the task internally.

버전 3.5.1에서 변경: 함수는 모든 어웨이터블 객체를 받아들입니다.

버전 3.10부터 폐지됨: Deprecation warning is emitted if *obj* is not a Future-like object and *loop* is not specified and there is no running event loop.

asyncio.**wrap_future**(*future*, *, *loop=None*)

`concurrent.futures.Future` 객체를 `asyncio.Future` 객체로 감쌉니다.

버전 3.10부터 폐지됨: Deprecation warning is emitted if *future* is not a Future-like object and *loop* is not specified and there is no running event loop.

Future 객체

class asyncio.**Future**(*, *loop=None*)

Future는 비동기 연산의 최종 결과를 나타냅니다. 스레드 안전하지 않습니다.

Future는 어웨이터블 객체입니다. 코루틴은 결과나 예외가 설정되거나 취소될 때까지 Future 객체를 기다릴 수 있습니다. Future는 여러번 어웨이트할 수 있으며 결과는 같습니다.

일반적으로 퓨처는 저수준 콜백 기반 코드(예를 들어, asyncio 트랜스포트를 사용하여 구현된 프로토콜에서)가 고수준 async/await 코드와 상호 운용되도록 하는 데 사용됩니다.

간단한 규칙은 사용자가 만나는 API에서 Future 객체를 절대 노출하지 않는 것이며, Future 객체를 만드는 권장 방법은 `loop.create_future()`를 호출하는 것입니다. 이런 식으로 대체 이벤트 루프 구현이 자신의 최적화된 Future 객체 구현을 주입할 수 있습니다.

버전 3.7에서 변경: `contextvars` 모듈에 대한 지원이 추가되었습니다.

버전 3.10부터 폐지됨: Deprecation warning is emitted if *loop* is not specified and there is no running event loop.

result()

Future의 결과를 반환합니다.

Future가 완료(*done*)했고 `set_result()` 메서드로 결과가 설정되었으면, 결괏값이 반환됩니다.

Future가 완료(*done*)했고 `set_exception()` 메서드로 예외가 설정되었으면, 이 메서드는 예외를 발생시킵니다.

Future가 취소(*cancelled*)되었으면, 이 메서드는 `CancelledError` 예외를 발생시킵니다.

Future의 결과를 아직 사용할 수 없으면, 이 메서드는 `InvalidStateError` 예외를 발생시킵니다.

set_result(*result*)

Future를 완료(*done*)로 표시하고, 그 결과를 설정합니다.

Future가 이미 완료(*done*)했으면, `InvalidStateError` 에러를 발생시킵니다.

set_exception(*exception*)

Future를 완료(*done*)로 표시하고, 예외를 설정합니다.

Future가 이미 완료(*done*)했으면, `InvalidStateError` 에러를 발생시킵니다.

done()

Future가 완료(*done*)했으면 `True`를 반환합니다.

Future는 취소(*cancelled*)되었거나 `set_result()` 나 `set_exception()` 호출로 결과나 예외가 설정되면 완료(*done*)됩니다.

cancelled()

Future가 최소(*cancelled*)되었으면, `True`를 반환합니다.

이 메서드는 대개 결과나 예외를 설정하기 전에 Future가 취소(*cancelled*)되었는지 확인하는 데 사용됩니다:

```
if not fut.cancelled():
    fut.set_result(42)
```

add_done_callback(*callback*, *, *context=None*)

Future가 완료(*done*)될 때 실행할 콜백을 추가합니다.

*callback*는 유일한 인자인 Future 객체로 호출됩니다.

이 메서드가 호출될 때 Future가 이미 완료(*done*)되었으면, 콜백이 `loop.call_soon()`으로 예약됩니다.

선택적 키워드 전용 *context* 인자는 *callback*이 실행될 사용자 정의 `contextvars.Context`를 지정할 수 있도록 합니다. *context*가 제공되지 않으면 현재 컨텍스트가 사용됩니다.

`functools.partial()`을 사용하여 매개 변수를 callback에 전달할 수 있습니다, 예를 들어:

```
# "fut"가 완료되면 'print("Future:", fut)'를 호출합니다.
fut.add_done_callback(
    functools.partial(print, "Future:"))
```

버전 3.7에서 변경: *context* 키워드 전용 매개 변수가 추가되었습니다. 자세한 내용은 **PEP 567**을 참조하십시오.

remove_done_callback(*callback*)

콜백 목록에서 *callback*을 제거합니다.

제거된 콜백 수를 반환합니다. 콜백이 두 번 이상 추가되지 않는 한 일반적으로 1입니다.

cancel(*msg=None*)

Future를 취소하고 콜백을 예약합니다.

Future가 이미 완료(*done*)했거나 취소(*cancelled*)되었으면, `False`를 반환합니다. 그렇지 않으면 Future의 상태를 취소(*cancelled*)로 변경하고, 콜백을 예약한 다음 `True`를 반환합니다.

버전 3.9에서 변경: *msg* 매개 변수를 추가했습니다.

exception()

이 Future에 설정된 예외를 반환합니다.

Future가 완료(*done*)했을 때만 예외(또는 예외가 설정되지 않았으면 `None`)가 반환됩니다.

Future가 취소(*cancelled*)되었으면, 이 메서드는 `CancelledError` 예외를 발생시킵니다.

Future가 아직 완료(*done*)하지 않았으면, 이 메서드는 `InvalidStateError` 예외를 발생시킵니다.

get_loop()

Future 객체가 연결된 이벤트 루프를 반환합니다.

Added in version 3.7.

이 예제는 Future 객체를 만들고, Future에 결과를 설정하는 비동기 Task를 만들고 예약하며, Future가 결과를 얻을 때까지 기다립니다:

```
async def set_after(fut, delay, value):
    # *delay* 초 동안 잠잡니다.
    await asyncio.sleep(delay)

    # *fut* 퓨처의 결과로 *value* 를 설정합니다.
    fut.set_result(value)

async def main():
    # 현재 이벤트 루프를 얻습니다.
    loop = asyncio.get_running_loop()

    # 새로운 Future 객체를 만듭니다.
    fut = loop.create_future()

    # 병렬 태스크로 "set_after()" 코루틴을 실행합니다.
    # 이벤트 루프에 대한 참조를 이미 가지고 있으므로, 여기서는 저수준 "loop.create_
↪task()"
    # API를 사용하고 있습니다.
    # 그렇지 않다면 그저 "asyncio.create_task()" 를 사용할 수 있었습니다.
    loop.create_task(
        set_after(fut, 1, '... world'))

    print('hello ...')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    # *fut* 에 결과가 올 때까지 기다렸다가 (1초) 그것을 인쇄합니다.
    print(await fut)

asyncio.run(main())
```

중요

Future 객체는 `concurrent.futures.Future`를 흉내 내도록 설계되었습니다. 주요 차이점은 다음과 같습니다:

- asyncio 퓨처와는 달리, `concurrent.futures.Future` 인스턴스는 await 할 수 없습니다.
- `asyncio.Future.result()` 와 `asyncio.Future.exception()` 은 *timeout* 인자를 받아들이지 않습니다.
- `asyncio.Future.result()` 와 `asyncio.Future.exception()` 는 Future 가 완료 (*done*) 하지 않았을 때 `InvalidStateError` 예외를 발생시킵니다.
- `asyncio.Future.add_done_callback()` 으로 등록된 콜백은 즉시 호출되지 않습니다. 대신 `loop.call_soon()` 로 예약됩니다.
- asyncio Future 는 `concurrent.futures.wait()` 와 `concurrent.futures.as_completed()` 함수와 호환되지 않습니다.
- `asyncio.Future.cancel()` 은 선택적 `msg` 인자를 받아들이지만, `concurrent.futures.Future.cancel()` 은 받아들이지 않습니다.

19.1.11 트랜스포트와 프로토콜

머리말

트랜스포트와 프로토콜은 `loop.create_connection()` 와 같은 **저수준** 이벤트 루프 API 에서 사용됩니다. 그들은 콜백 기반 프로그래밍 스타일을 사용하고 네트워크 나 IPC 프로토콜 (예를 들어 HTTP) 의 고성능 구현을 가능하게 합니다.

본질에서 트랜스포트와 프로토콜은 라이브러리와 프레임워크에서만 사용되어야 하며 고수준 asyncio 응용 프로그램에서는 사용되지 않아야 합니다.

이 문서 페이지는 트랜스포트와 프로토콜을 모두 다룹니다.

소개

최상위 수준에서, 트랜스포트는 어떻게 (*how*) 바이트를 전송할지를 다루지만, 프로토콜은 어떤 (*which*) 바이트를 전송할지를 결정합니다 (그리고 어느 정도는 언제 (when) 도).

같은 것을 다른 식으로 말하면: 트랜스포트는 소켓 (또는 유사한 I/O 엔드포인트) 의 추상화지만, 프로토콜은 트랜스포트의 관점에서 응용 프로그램의 추상화입니다.

또 다른 시각은 트랜스포트와 프로토콜 인터페이스가 함께 네트워크 I/O 와 프로세스 간 I/O 를 사용하기 위한 추상 인터페이스를 정의한다는 것입니다.

트랜스포트 객체와 프로토콜 객체 간에는 항상 1:1 관계가 있습니다: 프로토콜은 트랜스포트 메서드를 호출하여 데이터를 보내지만, 트랜스포트는 프로토콜 메서드를 호출하여 받은 데이터를 전달합니다.

대부분의 연결 지향 이벤트 루프 메서드 (가령 `loop.create_connection()`) 는 *Transport* 객체로 표현되는 받아들인 연결을 위한 *Protocol* 객체를 만드는 데 사용되는 *protocol_factory* 인자를 받아들입니다. 이러한 메서드는 대개 `(transport, protocol)` 의 튜플을 반환합니다.

목차

이 설명서 페이지는 다음 절로 구성됩니다:

- 트랜스포트 절은 asyncio `BaseTransport`, `ReadTransport`, `WriteTransport`, `Transport`, `DatagramTransport` 및 `SubprocessTransport` 클래스를 설명합니다.
- 프로토콜 절은 asyncio `BaseProtocol`, `Protocol`, `BufferedProtocol`, `DatagramProtocol` 및 `SubprocessProtocol` 클래스를 설명합니다.
- 예제 절은 트랜스포트, 프로토콜 및 저수준 이벤트 루프 API를 사용하는 방법을 보여줍니다.

트랜스포트

소스 코드: Lib/asyncio/transports.py

트랜스포트는 다양한 종류의 통신 채널을 추상화하기 위해 `asyncio`에서 제공하는 클래스입니다.

트랜스포트 객체는 항상 *asyncio* 이벤트 루프에 의해 인스턴스로 만들어집니다.

asyncio는 TCP, UDP, SSL 및 서브 프로세스 파이프를 위한 트랜스포트를 구현합니다. 트랜스포트에서 사용할 수 있는 메서드는 트랜스포트의 종류에 따라 다릅니다.

트랜스포트 클래스는 스레드 안전하지 않습니다.

트랜스포트 계층 구조

class asyncio.**BaseTransport**

모든 트랜스포트의 베이스 클래스. 모든 asyncio 트랜스포트가 공유하는 메서드를 포함합니다.

class asyncio.**WriteTransport**(*BaseTransport*)

쓰기 전용 연결을 위한 베이스 트랜스포트

WriteTransport 클래스의 인스턴스는 `loop.connect_write_pipe()` 이벤트 루프 메서드에서 반환되며 `loop.subprocess_exec()`와 같은 서브 프로세스 관련 메서드에서도 사용됩니다.

class asyncio.**ReadTransport**(*BaseTransport*)

읽기 전용 연결을 위한 베이스 트랜스포트

ReadTransport 클래스의 인스턴스는 `loop.connect_read_pipe()` 이벤트 루프 메서드에서 반환되며 `loop.subprocess_exec()`와 같은 서브 프로세스 관련 메서드에서도 사용됩니다.

class asyncio.**Transport**(*WriteTransport*, *ReadTransport*)

TCP 연결과 같은 양방향 트랜스포트를 나타내는 인터페이스.

사용자는 트랜스포트를 직접 인스턴스로 만들지 않습니다; 유틸리티 함수를 호출하고, 프로토콜 팩토리와 트랜스포트와 프로토콜을 만드는 데 필요한 기타 정보를 전달합니다.

Transport 클래스의 인스턴스는 `loop.create_connection()`, `loop.create_unix_connection()`, `loop.create_server()`, `loop.sendfile()` 등과 같은 이벤트 루프 메서드에서 반환되거나 사용됩니다.

class asyncio.**DatagramTransport**(*BaseTransport*)

데이터 그램 (UDP) 연결을 위한 트랜스포트.

DatagramTransport 클래스의 인스턴스는 `loop.create_datagram_endpoint()` 이벤트 루프 메서드에서 반환됩니다.

class asyncio.**SubprocessTransport**(*BaseTransport*)

부모와 그 자식 OS 프로세스 간의 연결을 나타내는 추상화.

SubprocessTransport 클래스의 인스턴스는 이벤트 루프 메서드 `loop.subprocess_shell()`과 `loop.subprocess_exec()`에서 반환됩니다.

베이스 트랜스포트

BaseTransport.**close**()

트랜스포트를 닫습니다.

트랜스포트에 송신 데이터용 버퍼가 있으면, 버퍼 된 데이터는 비동기적으로 플러시 됩니다. 더는 데이터가 수신되지 않습니다. 버퍼 된 모든 데이터가 플러시 된 후, 프로토콜의 *protocol.connection_lost()* 메서드가 *None*을 인자로 사용하여 호출됩니다. 트랜스포트가 닫힌 후에는 사용해서는 안 됩니다.

BaseTransport.**is_closing**()

트랜스포트가 닫히는 중이거나 닫혔으면 `True`를 반환합니다.

BaseTransport.**get_extra_info**(*name*, *default=None*)

트랜스포트나 그것이 사용하는 하부 자원에 대한 정보를 반환합니다.

*name*은 얻고자 하는 트랜스포트 특정 정보의 조각을 나타내는 문자열입니다.

*default*는 정보가 없거나 트랜스포트가 제삼자 이벤트 루프 구현이나 현재 플랫폼에서 조회를 지원하지 않을 때 반환 할 값입니다.

예를 들어, 다음 코드는 트랜스포트의 하부 소켓 객체를 가져오려고 시도합니다:

```
sock = transport.get_extra_info('socket')
if sock is not None:
    print(sock.getsockopt(...))
```

일부 트랜스포트에서 조회할 수 있는 정보의 범주:

- 소켓:
  - `'peername'`: 소켓이 연결된 원격 주소, *socket.socket.getpeername()*의 결과 (에러 시 `None`)
  - `'socket'`: *socket.socket* 인스턴스
  - `'sockname'`: 소켓 자체 주소, *socket.socket.getsockname()*의 결과
- SSL 소켓:
  - `'compression'`: 문자열로 표현된 사용된 압축 알고리즘, 또는 연결이 압축되지 않았으면 `None`; *ssl.SSLSocket.compression()*의 결과
  - `'cipher'`: 사용되는 암호 체계의 이름, 이의 사용을 정의하는 SSL 프로토콜의 버전 및 사용되는 비밀 비트의 수를 포함하는 3-튜플; *ssl.SSLSocket.cipher()*의 결과
  - `'peercert'`: 피어 인증서; *ssl.SSLSocket.getpeercert()*의 결과
  - `'sslcontext'`: *ssl.SSLContext* 인스턴스
  - `'ssl_object'`: *ssl.SSLObject* 나 *ssl.SSLSocket* 인스턴스
- 파이프:
  - `'pipe'`: 파이프 객체
- 서브 프로세스:
  - `'subprocess'`: *subprocess.Popen* 인스턴스

BaseTransport.**set_protocol**(*protocol*)

새 프로토콜을 설정합니다.

프로토콜의 교환은 두 프로토콜이 교환을 지원한다고 문서화 되었을 때만 수행되어야 합니다.

BaseTransport.**get_protocol**()

현재 프로토콜을 반환합니다.

읽기 전용 트랜스포트

ReadTransport.**is_reading**()

트랜스 포트가 새로운 데이터를 받고 있으면 `True`를 반환합니다.

Added in version 3.7.

ReadTransport.**pause_reading**()

트랜스포트의 수신 끝을 일시 중지합니다. *resume_reading()*이 호출 될 때까지 프로토콜의 *protocol.data_received()* 메서드에 데이터가 전달되지 않습니다.

버전 3.7에서 변경: 이 메서드는 멱등적(idempotent)입니다. 즉, 트랜스포트가 이미 일시 중지되거나 닫혔을 때도 호출할 수 있습니다.

ReadTransport.**resume_reading**()

수신 끝을 재개합니다. 읽을 수 있는 데이터가 있다면 프로토콜의 *protocol.data_received()* 메서드가 다시 한번 호출됩니다.

버전 3.7에서 변경: 이 메서드는 멱등적(idempotent)입니다. 즉, 트랜스포트가 이미 읽고 있을 때도 호출할 수 있습니다.

쓰기 전용 트랜스포트

WriteTransport.**abort**()

계류 중인 작업이 완료될 때까지 기다리지 않고, 즉시 트랜스포트를 닫습니다. 버퍼 된 데이터는 손실됩니다. 더는 데이터가 수신되지 않습니다. 프로토콜의 *protocol.connection_lost()* 메서드는 결국 *None*을 인자로 사용하여 호출됩니다.

WriteTransport.**can_write_eof**()

트랜스포트가 *write_eof()*를 지원하면 *True*를 반환하고, 그렇지 않으면 *False*를 반환합니다.

WriteTransport.**get_write_buffer_size**()

트랜스포트가 사용하는 출력 버퍼의 현재 크기를 반환합니다.

WriteTransport.**get_write_buffer_limits**()

쓰기 흐름 제어를 위한 *high* 와 *low* 수위(watermark)를 가져옵니다. 튜플 `(low, high)`를 반환합니다. 여기서 *low* 와 *high*는 양의 바이트 수입니다.

제한을 설정하려면 *set_write_buffer_limits()*를 사용하십시오.

Added in version 3.4.2.

WriteTransport.**set_write_buffer_limits**(*high=None*, *low=None*)

쓰기 흐름 제어를 위한 *high* 와 *low* 수위(watermark)를 설정합니다.

이 두 값(바이트 수로 측정)은 프로토콜의 *protocol.pause_writing()* 과 *protocol.resume_writing()* 메서드가 언제 호출될지를 제어합니다. 지정하면, low 수위는 high 수위보다 작거나 같아야 합니다. *high* 와 *low*는 음수가 될 수 없습니다.

버퍼 크기가 *high* 값보다 크거나 같아질 때 *pause_writing()*가 호출됩니다. 쓰기가 일시 중지되면, 버퍼 크기가 *low* 값보다 작거나 같아질 때 *resume_writing()*이 호출됩니다.

기본값은 구현에 따라 다릅니다. high 수위만 주어지면, low 수위는 high 수위보다 작거나 같은 구현 특정 기본값이 사용됩니다. *high*를 0으로 설정하면, *low*도 0으로 설정되고, 버퍼가 비어있지 않게 될 때마다 *pause_writing()*가 호출되도록 합니다. *low*를 0으로 설정하면 버퍼가 빌 때만 *resume_writing()*이 호출됩니다. 두 제한 중 하나에 0을 사용하는 것은 I/O와 계산을 동시에 수행할 기회를 줄이기 때문에 일반적으로 최선이 아닙니다.

제한을 가져오려면 *get_write_buffer_limits()*를 사용하십시오.

WriteTransport.**write**(*data*)

어떤 *data* 바이트열을 트랜스포트에 기록합니다.

이 메서드는 블록하지 않습니다; 데이터를 버퍼하고 비동기적으로 전송되도록 배치합니다.

WriteTransport.**writelines**(*list_of_data*)

트랜스포트에 데이터 바이트열 리스트(또는 임의의 이터러블)를 기록합니다. 이것은 이터러블이 산출하는 각 요소에 대해 `write()`를 호출하는 것과 기능 면에서 동등하지만, 더 효율적으로 구현될 수 있습니다.

WriteTransport.**write_eof**()

버퍼 된 모든 데이터를 플러시 한 후 트랜스포트의 쓰기 끝을 닫습니다. 데이터가 여전히 수신될 수 있습니다.

이 메서드는 트랜스포트(예를 들어 SSL)가 반만 닫힌 연결을 지원하지 않으면 `NotImplementedError`를 발생시킬 수 있습니다.

데이터 그램 트랜스포트

DatagramTransport.**sendto**(*data*, *addr=None*)

addr(트랜스포트 종속적인 대상 주소)에 의해 주어진 원격 피어로 *data* 바이트열을 보냅니다. *addr*가 `None`이면, 트랜스포트를 만들 때 지정된 대상 주소로 data가 전송됩니다.

이 메서드는 블록하지 않습니다; 데이터를 버퍼하고 비동기적으로 전송되도록 배치합니다.

버전 3.13에서 변경: This method can be called with an empty bytes object to send a zero-length datagram. The buffer size calculation used for flow control is also updated to account for the datagram header.

DatagramTransport.**abort**()

계류 중인 작업이 완료될 때까지 기다리지 않고, 즉시 트랜스포트를 닫습니다. 버퍼 된 데이터는 손실됩니다. 더는 데이터가 수신되지 않습니다. 프로토콜의 `protocol.connection_lost()` 메서드는 결국 `None`을 인자로 사용하여 호출됩니다.

서브 프로세스 트랜스포트

SubprocessTransport.**get_pid**()

서브 프로세스 ID를 정수로 반환합니다.

SubprocessTransport.**get_pipe_transport**(*fd*)

정수 파일 기술자 *fd*에 대응하는 통신 파이프의 트랜스포트를 돌려줍니다:

- `0`: writable streaming transport of the standard input (*stdin*), or `None` if the subprocess was not created with `stdin=PIPE`
- `1`: readable streaming transport of the standard output (*stdout*), or `None` if the subprocess was not created with `stdout=PIPE`
- `2`: readable streaming transport of the standard error (*stderr*), or `None` if the subprocess was not created with `stderr=PIPE`
- 다른 *fd*: `None`

SubprocessTransport.**get_returncode**()

서브 프로세스 반환 값을 정수로, 혹은 반환되지 않았으면 `None`을 반환합니다. `subprocess.Popen.returncode` 어트리뷰트와 유사합니다.

SubprocessTransport.**kill**()

서브 프로세스를 죽입니다.

POSIX 시스템에서, 이 함수는 SIGKILL을 서브 프로세스로 보냅니다. 윈도우에서, 이 메서드는 `terminate()`의 별칭입니다.

`subprocess.Popen.kill()`도 참조하십시오.

SubprocessTransport.**send_signal**(*signal*)

`subprocess.Popen.send_signal()`와 마찬가지로 *signal* 번호를 서브 프로세스로 보냅니다.

SubprocessTransport.**terminate**()

서브 프로세스를 중지합니다.

POSIX 시스템에서, 이 메서드는 *SIGTERM*을 서브 프로세스로 보냅니다. 윈도에서, 서브 프로세스를 중지하기 위해 윈도우 API 함수 `TerminateProcess()` 가 호출됩니다.

subprocess.Popen.terminate() 도 참조하십시오.

SubprocessTransport.**close**()

kill() 메서드를 호출하여 서브 프로세스를 죽입니다.

서브 프로세스가 아직 반환하지 않았으면, *stdin*, *stdout* 및 *stderr* 파이프의 트랜스포트를 닫습니다.

프로토콜

소스 코드: Lib/asyncio/protocols.py

---

asyncio는 네트워크 프로토콜을 구현하는 데 사용해야 하는 추상 베이스 클래스 집합을 제공합니다. 이러한 클래스는 트랜스포트와 함께 사용해야 합니다.

추상 베이스 프로토콜 클래스의 서브 클래스는 일부 또는 모든 메서드를 구현할 수 있습니다. 이 모든 메서드는 콜백입니다: 이것들은 특정 이벤트에서 트랜스포트에 의해 호출됩니다, 예를 들어 어떤 데이터가 수신될 때. 베이스 프로토콜 메서드는 해당 트랜스포트에 의해 호출되어야 합니다.

베이스 프로토콜

class asyncio.**BaseProtocol**

모든 프로토콜이 공유하는 메서드를 가진 베이스 프로토콜.

class asyncio.**Protocol**(*BaseProtocol*)

스트리밍 프로토콜(TCP, 유닉스 소켓 등)을 구현하기 위한 베이스 클래스.

class asyncio.**BufferedProtocol**(*BaseProtocol*)

수신 버퍼의 수동 제어로 스트리밍 프로토콜을 구현하기 위한 베이스 클래스.

class asyncio.**DatagramProtocol**(*BaseProtocol*)

데이터 그램 (UDP) 프로토콜을 구현하기 위한 베이스 클래스.

class asyncio.**SubprocessProtocol**(*BaseProtocol*)

자식 프로세스와 통신하는 (단방향 파이프) 프로토콜을 구현하기 위한 베이스 클래스.

베이스 프로토콜

모든 asyncio 프로토콜은 베이스 프로토콜 콜백을 구현할 수 있습니다.

연결 콜백

연결 콜백은 모든 프로토콜에서 성공적으로 연결될 때마다 정확히 한 번 호출됩니다. 다른 모든 프로토콜 콜백은 이 두 메서드 사이에서만 호출될 수 있습니다.

BaseProtocol.**connection_made**(*transport*)

연결이 이루어질 때 호출됩니다.

transport 인자는 연결을 나타내는 트랜스포트입니다. 트랜스포트에 대한 참조를 저장하는 것은 프로토콜의 책임입니다.

BaseProtocol.**connection_lost**(*exc*)

연결이 끊어지거나 닫힐 때 호출됩니다.

인자는 예외 객체나 *None*입니다. 후자는 정상적인 EOF가 수신되었거나, 연결의 이쪽에서 연결이 중단(abort)되거나 닫혔다는 것을 의미합니다.

흐름 제어 콜백

흐름 제어 콜백은 프로토콜에 의해 수행되는 쓰기를 일시 중지하거나 다시 시작하도록 트랜스포트에 의해 호출될 수 있습니다.

자세한 내용은 *set_write_buffer_limits()* 메서드 설명서를 참조하십시오.

BaseProtocol.**pause_writing**()

트랜스포트 버퍼가 높은 수위를 넘을 때 호출됩니다.

BaseProtocol.**resume_writing**()

트랜스포트 버퍼가 낮은 수위 아래로 내려갈 때 호출됩니다.

버퍼 크기가 높은 수위와 같으면, *pause_writing()* 이 호출되지 않습니다: 버퍼 크기는 엄격하게 초과해야 합니다.

반대로, *resume_writing()* 은 버퍼 크기가 낮은 수위와 같거나 낮을 때 호출됩니다. 이러한 종료 조건은 수위가 0일 때 예상대로 진행되게 하려면 중요합니다.

스트리밍 프로토콜

loop.create_server(), *loop.create_unix_server()*, *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_accepted_socket()*, *loop.connect_read_pipe()* 및 *loop.connect_write_pipe()* 와 같은 이벤트 메서드는 스트리밍 프로토콜을 반환하는 팩토리를 받아들입니다.

Protocol.**data_received**(*data*)

어떤 데이터가 수신될 때 호출됩니다. *data* 는 수신 데이터가 들어있는 비어 있지 않은 바이트열 객체입니다.

데이터가 버퍼 되고, 조각으로 나뉘고, 재조립되는지는 트랜스포트에 달려있습니다. 일반적으로, 특정 의미에 의존해서는 안 되며, 구문 분석을 일반적이고 유연하게 만들어야 합니다. 그러나, 데이터는 항상 올바른 순서로 수신됩니다.

이 메서드는 연결이 열려있는 동안 임의의 횟수만큼 호출될 수 있습니다.

그러나, *protocol.eof_received()* 는 최대한 한 번만 호출됩니다. 일단 `eof_received()` 가 호출되면, `data_received()` 는 더는 호출되지 않습니다.

Protocol.**eof_received**()

다른 쪽 끝이 더는 데이터를 보내지 않을 것이라는 신호를 보낼 때 호출됩니다 (예를 들어, 다른 쪽 끝도 asyncio를 사용하면, *transport.write_eof()* 를 호출하여).

이 메서드는 거짓 값(`None` 포함)을 반환할 수 있으며, 이때 트랜스포트는 스스로 닫힙니다. 반대로, 이 메서드가 참값을 반환하면, 사용된 프로토콜이 트랜스포트를 닫을지를 결정합니다. 기본 구현이 `None` 을 반환하기 때문에, 묵시적으로 연결을 닫습니다.

SSL을 포함한 일부 트랜스포트는, 반만 닫힌 연결을 지원하지 않습니다. 이때, 이 메서드에서 참을 반환하면 연결이 닫힙니다.

상태 기계:

```
start -> connection_made
    [-> data_received]*
    [-> eof_received]?
-> connection_lost -> end
```

버퍼 된 스트리밍 프로토콜

Added in version 3.7.

버퍼 된 프로토콜은 스트리밍 프로토콜을 지원하는 모든 이벤트 루프 메서드와 함께 사용할 수 있습니다.

`BufferedProtocol` 구현은 수신 버퍼의 명시적 수동 할당과 제어를 허용합니다. 이벤트 루프는 프로토콜에서 제공하는 버퍼를 사용하여 불필요한 데이터 복사를 피할 수 있습니다. 이로 인해 대량의 데이터를

수신하는 프로토콜의 성능이 크게 향상될 수 있습니다. 정교한 프로토콜 구현은 버퍼 할당 수를 크게 줄일 수 있습니다.

다음 콜백은 `BufferedProtocol` 인스턴스에 호출됩니다.:

`BufferedProtocol.`**`get_buffer`**(*sizehint*)

새로운 수신 버퍼를 할당하기 위해서 호출됩니다.

*sizehint*는 반환되는 버퍼에 대해 권장되는 최소 크기입니다. *sizehint*가 제안하는 것보다 더 작거나 큰 버퍼를 반환하는 것이 허용됩니다. -1로 설정하면 버퍼 크기는 임의적일 수 있습니다. 크기가 0인 버퍼를 반환하는 것은 에러입니다.

`get_buffer()`는 버퍼 프로토콜을 구현하는 객체를 반환해야 합니다.

`BufferedProtocol.`**`buffer_updated`**(*nbytes*)

수신된 데이터로 버퍼가 갱신될 때 호출됩니다.

*nbytes*는 버퍼에 기록된 총 바이트 수입니다.

`BufferedProtocol.`**`eof_received`**()

`protocol.eof_received()` 메서드의 설명서를 참조하십시오.

`get_buffer()`는 연결 중에 임의의 횟수만큼 호출될 수 있습니다. 그러나, `protocol.eof_received()`는 최대한 한 번만 호출되며, 호출되면 `get_buffer()`와 `buffer_updated()`는 그 이후로 호출되지 않습니다.

상태 기계:

```
start -> connection_made
    [-> get_buffer
        [-> buffer_updated]?
    ]*
    [-> eof_received]?
-> connection_lost -> end
```

데이터 그램 프로토콜

데이터 그램 프로토콜 인스턴스는 `loop.create_datagram_endpoint()` 메서드에 전달된 프로토콜 팩토리에 의해 만들어져야 합니다.

`DatagramProtocol.`**`datagram_received`**(*data*, *addr*)

데이터 그램이 수신될 때 호출됩니다. *data*는 수신 데이터를 포함하는 바이트열 객체입니다. *addr*는 데이터를 보내는 피어의 주소입니다; 정확한 형식은 트랜스포트에 따라 다릅니다.

`DatagramProtocol.`**`error_received`**(*exc*)

이전의 송신이나 수신 연산이 `OSError`를 일으킬 때 호출됩니다. *exc*는 `OSError` 인스턴스입니다.

이 메서드는 드문 조건에서 호출됩니다, 트랜스포트(예를 들어 UDP)가 데이터 그램을 수신자에게 전달할 수 없음을 감지했을 때입니다. 하지만 대부분 전달할 수 없는 데이터 그램은 조용히 삭제됩니다.

> **참고**
>
> BSD 시스템(macOS, FreeBSD 등)에서는, 너무 많은 패킷을 쓰는 것으로 인한 전송 실패를 감지하는 신뢰성 있는 방법이 없으므로 데이터 그램 프로토콜에 대한 흐름 제어가 지원되지 않습니다.
>
> 소켓은 항상 'ready'로 나타나고 여분의 패킷은 삭제됩니다. `errno`가 `errno.ENOBUFS`로 설정된 `OSError`가 발생할 수도 그렇지 않을 수도 있습니다; 발생하면, `DatagramProtocol.error_received()`에게 보고되지만 그렇지 않으면 무시됩니다.

서브 프로세스 프로토콜

서브 프로세스 프로토콜 인스턴스는 *loop.subprocess_exec()* 와 *loop.subprocess_shell()* 메서드에 전달된 프로토콜 팩토리에 의해 만들어져야 합니다.

SubprocessProtocol.**pipe_data_received**(*fd*, *data*)

자식 프로세스가 stdout 이나 stderr 파이프에 데이터를 쓸 때 호출됩니다.

*fd*는 파이프의 정수 파일 기술자입니다.

*data*는 수신 된 데이터를 포함하는 비어 있지 않은 바이트열 객체입니다.

SubprocessProtocol.**pipe_connection_lost**(*fd*, *exc*)

자식 프로세스와 통신하는 파이프 중 하나가 닫히면 호출됩니다.

*fd*는 닫힌 정수 파일 기술자입니다.

SubprocessProtocol.**process_exited**()

자식 프로세스가 종료할 때 호출됩니다.

It can be called before *pipe_data_received()* and *pipe_connection_lost()* methods.

예제

TCP 메아리 서버

loop.create_server() 메서드를 사용하여 TCP 메아리 서버를 만들고, 받은 데이터를 다시 보내고, 연결을 닫습니다:

```
import asyncio


class EchoServerProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()


async def main():
    # 저수준 API를 사용할 계획이므로
    # 이벤트 루프에 대한 참조를 얻습니다.
    loop = asyncio.get_running_loop()

    server = await loop.create_server(
        EchoServerProtocol,
        '127.0.0.1', 8888)

    async with server:
        await server.serve_forever()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
asyncio.run(main())
```

더 보기

스트림을 사용하는 *TCP* 메아리 서버 예제는 고수준 `asyncio.start_server()` 함수를 사용합니다.

TCP 메아리 클라이언트

`loop.create_connection()` 메서드를 사용하는 TCP 메아리 클라이언트, 데이터를 보내고 연결이 닫힐 때까지 기다립니다:

```
import asyncio


class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
        self.on_con_lost.set_result(True)


async def main():
    # 저수준 API를 사용할 계획이므로
    # 이벤트 루프에 대한 참조를 얻습니다.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = 'Hello World!'

    transport, protocol = await loop.create_connection(
        lambda: EchoClientProtocol(message, on_con_lost),
        '127.0.0.1', 8888)

    # 프로토콜이 연결이 끊어졌음을 알릴 때까지 기다리고
    # 트랜스포트를 닫습니다.
    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

> **더 보기**
>
> 스트림을 사용하는 *TCP* 메아리 클라이언트 예제는 고수준 `asyncio.open_connection()` 함수를 사용합니다.

UDP 메아리 서버

`loop.create_datagram_endpoint()` 메서드를 사용하는 UDP 메아리 서버, 수신된 데이터를 다시 보냅니다:

```
import asyncio


class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)


async def main():
    print("Starting UDP server")

    # 저수준 API를 사용할 계획이므로
    # 이벤트 루프에 대한 참조를 얻습니다.
    loop = asyncio.get_running_loop()

    # 모든 클라이언트 요청을 처리할
    # 하나의 프로토콜 인스턴스가 만들어집니다.
    transport, protocol = await loop.create_datagram_endpoint(
        EchoServerProtocol,
        local_addr=('127.0.0.1', 9999))

    try:
        await asyncio.sleep(3600)  # 1시간 동안 서비스합니다.
    finally:
        transport.close()


asyncio.run(main())
```

UDP 메아리 클라이언트

`loop.create_datagram_endpoint()` 메서드를 사용하는 UDP 메아리 클라이언트, 데이터를 보내고 응답을 받으면 트랜스포트를 닫습니다:

```
import asyncio


class EchoClientProtocol:
    def __init__(self, message, on_con_lost):
        self.message = message
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        self.on_con_lost = on_con_lost
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Connection closed")
        self.on_con_lost.set_result(True)


async def main():
    # 저수준 API를 사용할 계획이므로
    # 이벤트 루프에 대한 참조를 얻습니다.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = "Hello World!"

    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoClientProtocol(message, on_con_lost),
        remote_addr=('127.0.0.1', 9999))

    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

기존 소켓 연결하기

프로토콜과 함께 *loop.create_connection()* 메서드를 사용하여 소켓이 데이터를 수신할 때까지 기다립니다:

```
import asyncio
import socket


class MyProtocol(asyncio.Protocol):

    def __init__(self, on_con_lost):
        self.transport = None
        self.on_con_lost = on_con_lost
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())

        # 작업을 완료했습니다: 트랜스포트를 닫습니다;
        # connection_lost() 가 자동으로 호출됩니다.
        self.transport.close()

    def connection_lost(self, exc):
        # 소켓이 닫혔습니다
        self.on_con_lost.set_result(True)


async def main():
    # 저수준 API를 사용할 계획이므로
    # 이벤트 루프에 대한 참조를 얻습니다.
    loop = asyncio.get_running_loop()
    on_con_lost = loop.create_future()

    # 연결된 소켓 쌍을 만듭니다
    rsock, wsock = socket.socketpair()

    # 데이터를 기다릴 소켓을 등록합니다.
    transport, protocol = await loop.create_connection(
        lambda: MyProtocol(on_con_lost), sock=rsock)

    # 네트워크로부터의 데이터 수신을 흉내 냅니다.
    loop.call_soon(wsock.send, 'abc'.encode())

    try:
        await protocol.on_con_lost
    finally:
        transport.close()
        wsock.close()

asyncio.run(main())
```

더 보기

파일 기술자에서 읽기 이벤트를 관찰하기 예제는 저수준의 *loop.add_reader()* 메서드를 사용하여 FD를 등록합니다.

스트림을 사용하여 데이터를 기다리는 열린 소켓 등록 예제는 코루틴에서 *open_connection()* 함수에 의해 생성된 고수준 스트림을 사용합니다.

loop.subprocess_exec() 와 SubprocessProtocol

서브 프로세스의 출력을 가져오고 서브 프로세스가 끝날 때까지 대기하는 데 사용되는 서브 프로세스 프로토콜의 예.

서브 프로세스는 *loop.subprocess_exec()* 메서드에 의해 만들어집니다:

```
import asyncio
import sys

class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()
        self.pipe_closed = False
        self.exited = False

    def pipe_connection_lost(self, fd, exc):
        self.pipe_closed = True
        self.check_for_exit()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exited = True
        # pipe_connection_lost() 메서드 전에
        # process_exited() 메서드가 호출될 수 있습니다:
        # 두 메서드가 모두 호출될 때까지 기다립니다.
        self.check_for_exit()

    def check_for_exit(self):
        if self.pipe_closed and self.exited:
            self.exit_future.set_result(True)

async def get_date():
    # 저수준 API를 사용할 계획이므로
    # 이벤트 루프에 대한 참조를 얻습니다.
    loop = asyncio.get_running_loop()

    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # DateProtocol이 제어하는 서브 프로세스를 만듭니다;
    # 표준 출력을 파이프로 리디렉트합니다.
    transport, protocol = await loop.subprocess_exec(
        lambda: DateProtocol(exit_future),
        sys.executable, '-c', code,
        stdin=None, stderr=None)

    # 프로토콜의 process_exited() 메서드를 사용하여
    # 서브 프로세스 종료를 기다립니다.
    await exit_future

    # stdout 파이프를 닫습니다.
    transport.close()

    # 프로토콜의 pipe_data_received() 메서드로
    # 수집된 출력을 읽습니다.
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

고수준 API를 사용하여 작성된 같은 예제도 참조하십시오.

19.1.12 정책

> **경고**
>
> Policies are deprecated and will be removed in Python 3.16. Users are encouraged to use the `asyncio.run()` function or the `asyncio.Runner` with *loop_factory* to use the desired loop implementation.

An event loop policy is a global object used to get and set the current *event loop*, as well as create new event loops. The default policy can be *replaced* with *built-in alternatives* to use different event loop implementations, or substituted by a *custom policy* that can override these behaviors.

The *policy object* gets and sets a separate event loop per *context*. This is per-thread by default, though custom policies could define *context* differently.

사용자 정의 이벤트 루프 정책은 `get_event_loop()`, `set_event_loop()` 및 `new_event_loop()`의 동작을 제어할 수 있습니다.

정책 객체는 `AbstractEventLoopPolicy` 추상 베이스 클래스에 정의된 API를 구현해야 합니다.

정책을 얻고 설정하기

다음 함수는 현재 프로세스의 정책을 가져오고 설정하는 데 사용할 수 있습니다:

`asyncio.`**`get_event_loop_policy`**`()`

현재의 프로세스 전반의 정책을 돌려줍니다.

버전 3.14부터 폐지됨: The `get_event_loop_policy()` function is deprecated and will be removed in Python 3.16.

`asyncio.`**`set_event_loop_policy`**`(`*policy*`)`

현재 프로세스 전반의 정책을 *policy*로 설정합니다.

*policy*를 `None`으로 설정하면, 기본 정책이 복원됩니다.

버전 3.14부터 폐지됨: The `set_event_loop_policy()` function is deprecated and will be removed in Python 3.16.

정책 객체

추상 이벤트 루프 정책 베이스 클래스는 다음과 같이 정의됩니다:

`class` `asyncio.`**`AbstractEventLoopPolicy`**

asyncio 정책의 추상 베이스 클래스.

`get_event_loop``()`

현재 컨텍스트의 이벤트 루프를 가져옵니다.

`AbstractEventLoop` 인터페이스를 구현하는 이벤트 루프 객체를 반환합니다.

이 메서드는 절대 `None`을 반환해서는 안 됩니다.

버전 3.6에서 변경.

`set_event_loop``(`*loop*`)`

현재 컨텍스트에 대한 이벤트 루프를 *loop*로 설정합니다.

`new_event_loop``()`

새 이벤트 루프 객체를 만들고 반환합니다.

이 메서드는 절대 `None`을 반환해서는 안 됩니다.

버전 3.14부터 폐지됨: The *AbstractEventLoopPolicy* class is deprecated and will be removed in Python 3.16.

asyncio에는 다음과 같은 내장 정책이 제공됩니다:

class asyncio.**DefaultEventLoopPolicy**

기본 asyncio 정책. 유닉스에서는 *SelectorEventLoop*를, 윈도우에서는 *ProactorEventLoop*를 사용합니다.

수동으로 기본 정책을 설치할 필요는 없습니다. asyncio는 기본 정책을 자동으로 사용하도록 구성됩니다.

버전 3.8에서 변경: 윈도우에서, 이제 기본적으로 *ProactorEventLoop*가 사용됩니다.

버전 3.14에서 변경: The *get_event_loop()* method of the default asyncio policy now raises a *RuntimeError* if there is no set event loop.

버전 3.14부터 폐지됨: The *DefaultEventLoopPolicy* class is deprecated and will be removed in Python 3.16.

class asyncio.**WindowsSelectorEventLoopPolicy**

SelectorEventLoop 이벤트 루프 구현을 사용하는 대안 이벤트 루프 정책.

가용성: Windows.

버전 3.14부터 폐지됨: The *WindowsSelectorEventLoopPolicy* class is deprecated and will be removed in Python 3.16.

class asyncio.**WindowsProactorEventLoopPolicy**

ProactorEventLoop 이벤트 루프 구현을 사용하는 대안 이벤트 루프 정책.

가용성: Windows.

버전 3.14부터 폐지됨: The *WindowsProactorEventLoopPolicy* class is deprecated and will be removed in Python 3.16.

사용자 정의 정책

새로운 이벤트 루프 정책을 구현하려면, *DefaultEventLoopPolicy*의 서브 클래스를 만들고 사용자 정의 동작이 필요한 메서드를 재정의하는 것이 좋습니다, 예를 들어:

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
        """이벤트 루프를 가져옵니다.

        None이거나 EventLoop의 인스턴스일 수 있습니다.
        """
        loop = super().get_event_loop()
        # loop로 무언가를 합니다 ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

19.1.13 플랫폼 지원

asyncio 모듈은 이식성이 있도록 설계되었지만, 플랫폼의 하부 아키텍처와 기능으로 인해 일부 플랫폼에는 미묘한 차이점과 제약이 있습니다.

모든 플랫폼

- `loop.add_reader()`와 `loop.add_writer()`는 파일 I/O를 감시하는데 사용할 수 없습니다.

윈도우

소스 코드: Lib/asyncio/proactor_events.py, Lib/asyncio/windows_events.py, Lib/asyncio/windows_utils.py

---

버전 3.8에서 변경: 윈도우에서, `ProactorEventLoop`는 이제 기본 이벤트 루프입니다.

윈도우의 모든 이벤트 루프는 다음 메서드를 지원하지 않습니다:

- `loop.create_unix_connection()` 와 `loop.create_unix_server()`는 지원되지 않습니다. `socket.AF_UNIX` 소켓 패밀리는 유닉스에만 적용됩니다.
- `loop.add_signal_handler()`와 `loop.remove_signal_handler()`는 지원되지 않습니다.

`SelectorEventLoop`에는 다음과 같은 제약이 있습니다:

- `SelectSelector`는 소켓 이벤트를 기다리는 데 사용됩니다: 소켓을 지원하며 512개의 소켓으로 제한됩니다.
- `loop.add_reader()`와 `loop.add_writer()`는 소켓 핸들만 받아들입니다 (예를 들어, 파이프 파일 기술자는 지원되지 않습니다).
- 파이프는 지원되지 않으므로, `loop.connect_read_pipe()`와 `loop.connect_write_pipe()` 메서드는 구현되지 않습니다.
- 서브 프로세스는 지원되지 않습니다, 즉, `loop.subprocess_exec()` 와 `loop.subprocess_shell()` 메서드가 구현되지 않습니다.

`ProactorEventLoop`에는 다음과 같은 제약이 있습니다:

- `loop.add_reader()`와 `loop.add_writer()` 메서드는 지원되지 않습니다.

윈도우에서 단조 시계의 해상도는 대개 15.6 밀리초 근처입니다. 최상의 해상도는 0.5 밀리초입니다. 해상도는 하드웨어(HPET이 사용 가능한지)와 윈도우 구성에 따라 다릅니다.

윈도우에서의 서브 프로세스 지원

윈도우에서, 기본 이벤트 루프 `ProactorEventLoop`는 서브 프로세스를 지원하지만, `SelectorEventLoop`는 그렇지 않습니다.

macOS

최신 macOS 버전은 완전하게 지원됩니다.

macOS <= 10.8

macOS 10.6, 10.7 및 10.8에서, 기본 이벤트 루프는 `selectors.KqueueSelector`를 사용하는데, 이 버전에서는 문자 장치를 지원하지 않습니다. 이러한 이전 버전의 macOS에서 문자 장치를 지원하려면, `SelectorEventLoop`가 `SelectSelector` 나 `PollSelector`를 사용하도록 수동으로 구성할 수 있습니다. 예:

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

19.1.14 Extending

The main direction for `asyncio` extending is writing custom *event loop* classes. Asyncio has helpers that could be used to simplify this task.

> **참고**
>
> Third-parties should reuse existing asyncio code with caution, a new Python version is free to break backward compatibility in *internal* part of API.

Writing a Custom Event Loop

`asyncio.AbstractEventLoop` declares very many methods. Implementing all them from scratch is a tedious job.

A loop can get many common methods implementation for free by inheriting from `asyncio.BaseEventLoop`.

In turn, the successor should implement a bunch of *private* methods declared but not implemented in `asyncio.BaseEventLoop`.

For example, `loop.create_connection()` checks arguments, resolves DNS addresses, and calls `loop._make_socket_transport()` that should be implemented by inherited class. The `_make_socket_transport()` method is not documented and is considered as an *internal* API.

Future and Task private constructors

`asyncio.Future` and `asyncio.Task` should be never created directly, please use corresponding `loop.create_future()` and `loop.create_task()`, or `asyncio.create_task()` factories instead.

However, third-party *event loops* may *reuse* built-in future and task implementations for the sake of getting a complex and highly optimized code for free.

For this purpose the following, *private* constructors are listed:

`Future.__init__`(*, *loop=None*)

Create a built-in future instance.

loop is an optional event loop instance.

`Task.__init__`(*coro*, *, *loop=None*, *name=None*, *context=None*)

Create a built-in task instance.

loop is an optional event loop instance. The rest of arguments are described in `loop.create_task()` description.

버전 3.11에서 변경: *context* argument is added.

Task lifetime support

A third party task implementation should call the following functions to keep a task visible by `asyncio.all_tasks()` and `asyncio.current_task()`:

`asyncio._register_task`(*task*)

Register a new *task* as managed by *asyncio*.

Call the function from a task constructor.

`asyncio._unregister_task`(*task*)

Unregister a *task* from *asyncio* internal structures.

The function should be called when a task is about to finish.

asyncio.**_enter_task**(*loop*, *task*)

Switch the current task to the *task* argument.

Call the function just before executing a portion of embedded *coroutine* (`coroutine.send()` or `coroutine.throw()`).

asyncio.**_leave_task**(*loop*, *task*)

Switch the current task back from *task* to `None`.

Call the function just after `coroutine.send()` or `coroutine.throw()` execution.

19.1.15 고수준 API 색인

이 페이지에는 모든 고수준의 async/await 활성화된 asyncio API가 나열됩니다.

태스크

asyncio 프로그램을 실행하고, 태스크를 만들고, 시간제한 있게 여러 가지를 기다리는 유틸리티.

run()	이벤트 루프를 만들고, 코루틴을 실행하고, 루프를 닫습니다.
Runner	A context manager that simplifies multiple async function calls.
Task	Task 객체.
TaskGroup	A context manager that holds a group of tasks. Provides a convenient and reliable way to wait for all tasks in the group to finish.
create_task()	asyncio Task를 시작한 후 반환합니다.
current_task()	현재 Task를 돌려줍니다.
all_tasks()	이벤트 루프의 아직 종료하지 않은 모든 태스크를 반환합니다.
`await` *sleep()*	몇 초 동안 잠잡니다.
`await` *gather()*	여러 가지를 동시에 예약하고 기다립니다.
`await` *wait_for()*	시간제한 있게 실행합니다.
`await` *shield()*	취소로부터 보호합니다.
`await` *wait()*	완료를 감시합니다.
timeout()	Run with a timeout. Useful in cases when `wait_for` is not suitable.
to_thread()	별도의 OS 스레드에서 함수를 비동기적으로 실행합니다.
run_coroutine_threadsafe()	다른 OS 스레드에서 코루틴을 예약합니다.
`for in` *as_completed()*	`for` 루프로 완료를 감시합니다.

예제

- 여러 가지를 병렬로 실행하기 위해 *asyncio.gather*() 사용하기.
- 시간제한을 주기 위해 *asyncio.wait_for*() 사용하기.
- 취소.
- *asyncio.sleep*() 사용하기.
- 주 태스크 설명서 페이지를 참조하십시오.

큐

큐는 여러 asyncio 태스크 간에 작업을 분산하고, 연결 풀과 pub/sub 패턴을 구현하는 데 사용해야 합니다.

Queue	FIFO 큐.
PriorityQueue	우선순위 큐.
LifoQueue	LIFO 큐.

예제

- 여러 태스크로 작업부하를 분산하는데 *asyncio.Queue* 사용하기.
- 큐 설명서 페이지도 참조하십시오.

서브 프로세스

서브 프로세스를 생성하고 셸 명령을 실행하는 유틸리티.

`await` *create_subprocess_exec()*	서브 프로세스를 만듭니다.
`await` *create_subprocess_shell()*	셸 명령을 실행합니다.

예제

- 셸 명령 실행하기.
- 서브 프로세스 *API* 설명서도 참조하십시오.

스트림

네트워크 IO로 작업하는 고수준 API

`await` *open_connection()*	TCP 연결을 만듭니다.
`await` *open_unix_connection()*	유닉스 소켓 연결을 만듭니다.
`await` *start_server()*	TCP 서버를 시작합니다.
`await` *start_unix_server()*	유닉스 소켓 서버를 시작합니다.
StreamReader	네트워크 데이터를 수신하는 고수준 async/await 객체.
StreamWriter	네트워크 데이터를 보내는 고수준 async/await 객체.

예제

- 예제 *TCP* 클라이언트.
- 스트림 *API* 설명서도 참조하십시오.

동기화

태스크에 쓸 수 있는 threading과 유사한 동기화 프리미티브.

Lock	뮤텍스 록.
Event	이벤트 객체.
Condition	조건 객체.
Semaphore	세마포어.
BoundedSemaphore	제한된 세마포어.
Barrier	장벽 객체.

예제

- *asyncio.Event* 사용하기.
- *asyncio.Barrier* 사용하기.
- asyncio 동기화 프리미티브의 설명서도 참조하십시오.

예외

`asyncio.CancelledError`	Task가 취소될 때 발생합니다. `Task.cancel()`도 참조하십시오.
`asyncio.BrokenBarrierError`	Barrier가 깨질 때 발생합니다. `Barrier.wait()`도 참조하십시오.

예제

- 취소 요청 시에 코드를 실행하기 위해 *CancelledError* 처리하기.
- *asyncio* 전용 예외의 전체 목록도 참조하십시오.

19.1.16 저수준 API 색인

이 페이지는 모든 저수준 asyncio API를 나열합니다.

이벤트 루프 얻기

`asyncio.get_running_loop()`	실행 중인 이벤트 루프를 가져오는 데 **선호되는** 함수.
`asyncio.get_event_loop()`	이벤트 루프 인스턴스를 가져옵니다 (실행 중이거나 현재 정책을 통해 현재).
`asyncio.set_event_loop()`	현재 정책을 통해 현재 이벤트 루프를 설정합니다.
`asyncio.new_event_loop()`	새 이벤트 루프를 만듭니다.

예제

- *asyncio.get_running_loop*() 사용하기.

이벤트 루프 메서드

이벤트 루프 메서드에 관한 주 설명서 절도 참조하십시오.

수명주기

`loop.run_until_complete()`	완료할 때까지 퓨처/태스크/어웨이터블을 실행합니다.
`loop.run_forever()`	이벤트 루프를 영원히 실행합니다.
`loop.stop()`	이벤트 루프를 중지합니다.
`loop.close()`	이벤트 루프를 닫습니다.
`loop.is_running()`	이벤트 루프가 실행 중이면 `True`를 반환합니다.
`loop.is_closed()`	이벤트 루프가 닫혔으면 `True`를 반환합니다.
`await loop.shutdown_asyncgens()`	비동기 제너레이터를 닫습니다.

디버깅

loop.set_debug()	디버그 모드를 활성화 또는 비활성화합니다.
loop.get_debug()	현재의 디버그 모드를 얻습니다.

콜백 예약하기

loop.call_soon()	콜백을 곧 호출합니다.
loop.call_soon_threadsafe()	스레드 안전한 *loop.call_soon()*의 변형입니다.
loop.call_later()	주어진 시간 후에 콜백을 호출합니다.
loop.call_at()	주어진 시간 에 콜백을 호출합니다.

Thread/Interpreter/Process Pool

`await` *loop.run_in_executor()*	*concurrent.futures* 실행기에서 CPU-병목이나 다른 블로킹 함수를 실행합니다.
loop.set_default_executor()	*loop.run_in_executor()*의 기본 실행기를 설정합니다.

태스크와 퓨처

loop.create_future()	*Future* 객체를 만듭니다.
loop.create_task()	코루틴을 *Task*로 예약합니다.
loop.set_task_factory()	*loop.create_task()*가 태스크를 만드는데 사용하는 팩토리를 설정합니다.
loop.get_task_factory()	*loop.create_task()*가 태스크를 만드는데 사용하는 팩토리를 얻습니다.

DNS

`await` *loop.getaddrinfo()*	*socket.getaddrinfo()*의 비동기 버전.
`await` *loop.getnameinfo()*	*socket.getnameinfo()*의 비동기 버전.

네트워킹과 IPC

`await` *loop.create_connection()*	TCP 연결을 엽니다.
`await` *loop.create_server()*	TCP 서버를 만듭니다.
`await` *loop.create_unix_connection()*	유닉스 소켓 연결을 엽니다.
`await` *loop.create_unix_server()*	Unix 소켓 서버를 만듭니다.
`await` *loop.connect_accepted_socket()*	*socket*을 `(transport, protocol)` 쌍으로 감쌉니다.
`await` *loop.create_datagram_endpoint()*	데이터 그램 (UDP) 연결을 엽니다.
`await` *loop.sendfile()*	트랜스포트를 통해 파일을 보냅니다.
`await` *loop.start_tls()*	기존 연결을 TLS로 업그레이드합니다.
`await` *loop.connect_read_pipe()*	파이프의 읽기 끝을 `(transport, protocol)` 쌍으로 감쌉니다.
`await` *loop.connect_write_pipe()*	파이프의 쓰기 끝을 `(transport, protocol)` 쌍으로 감쌉니다.

소켓

`await loop.sock_recv()`	`socket`에서 데이터를 수신합니다.
`await loop.sock_recv_into()`	`socket`에서 데이터를 버퍼로 수신합니다.
`await loop.sock_recvfrom()`	`socket`에서 데이터 그램을 수신합니다.
`await loop.sock_recvfrom_into()`	`socket`에서 데이터 그램을 버퍼로 수신합니다.
`await loop.sock_sendall()`	데이터를 `socket`으로 보냅니다.
`await loop.sock_sendto()`	데이터 그램을 `socket`을 통해 주어진 주소로 보냅니다.
`await loop.sock_connect()`	`socket`을 연결합니다.
`await loop.sock_accept()`	`socket` 연결을 수락합니다.
`await loop.sock_sendfile()`	`socket`를 통해 파일을 보냅니다.
`loop.add_reader()`	파일 기술자가 읽기 가능한지 관찰하기 시작합니다.
`loop.remove_reader()`	파일 기술자가 읽기 가능한지 관찰하는 것을 중단합니다.
`loop.add_writer()`	파일 기술자가 쓰기 가능한지 관찰하기 시작합니다.
`loop.remove_writer()`	파일 기술자가 쓰기 가능한지 관찰하는 것을 중단합니다.

유닉스 시그널

`loop.add_signal_handler()`	`signal`에 대한 처리기를 추가합니다.
`loop.remove_signal_handler()`	`signal`에 대한 처리기를 제거합니다.

서브 프로세스

`loop.subprocess_exec()`	서브 프로세스를 스폰합니다.
`loop.subprocess_shell()`	셸 명령으로 서브 프로세스를 스폰합니다.

에러 처리

`loop.call_exception_handler()`	예외 처리기를 호출합니다.
`loop.set_exception_handler()`	새로운 예외 처리기를 설정합니다.
`loop.get_exception_handler()`	현재 예외 처리기를 가져옵니다.
`loop.default_exception_handler()`	기본 예외 처리기 구현.

예제

- *asyncio.new_event_loop*() 와 *loop.run_forever*() 사용하기.
- *loop.call_later*() 사용하기.
- `loop.create_connection()` 을 사용하여 메아리 클라이언트 구현하기.
- `loop.create_connection()` 을 사용하여 소켓 연결하기.
- *add_reader*()를 사용하여 *FD*에서 읽기 이벤트 관찰하기.
- *loop.add_signal_handler*() 사용하기.
- *loop.subprocess_exec*() 사용하기.

트랜스포트

모든 트랜스포트는 다음과 같은 메서드를 구현합니다:

transport.close()	트랜스포트를 닫습니다.
transport.is_closing()	트랜스포트가 닫히고 있거나 닫혔으면 `True`를 반환합니다.
transport.get_extra_info()	트랜스포트에 대한 정보를 요청합니다.
transport.set_protocol()	새 프로토콜을 설정합니다.
transport.get_protocol()	현재 프로토콜을 돌려줍니다.

데이터를 받을 수 있는 트랜스포트 (TCP 및 유닉스 연결, 파이프 등). *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_read_pipe()* 등의 메서드에서 반환됩니다:

트랜스포트 읽기

transport.is_reading()	트랜스포트가 수신 중이면 `True`를 반환합니다.
transport.pause_reading()	수신을 일시 정지합니다.
transport.resume_reading()	수신을 재개합니다.

데이터를 전송할 수 있는 트랜스포트 (TCP와 유닉스 연결, 파이프 등). *loop.create_connection()*, *loop.create_unix_connection()*, *loop.connect_write_pipe()* 등의 메서드에서 반환됩니다:

트랜스포트 쓰기

transport.write()	데이터를 트랜스포트에 씁니다.
transport.writelines()	버퍼들을 트랜스포트에 씁니다.
transport.can_write_eof()	트랜스포트가 EOF 전송을 지원하면 *True*를 반환합니다.
transport.write_eof()	버퍼 된 데이터를 플러시 한 후 닫고 EOF를 보냅니다.
transport.abort()	즉시 트랜스포트를 닫습니다.
transport.get_write_buffer_size()	출력 버퍼의 현재 크기를 돌려줍니다.
transport.get_write_buffer_limits()	쓰기 흐름 제어를 위한 높은 수위와 낮은 수위를 반환합니다.
transport.set_write_buffer_limits()	쓰기 흐름 제어를 위한 새로운 높은 수위와 낮은 수위를 설정합니다.

*loop.create_datagram_endpoint()*에서 반환된 트랜스포트:

데이터 그램 트랜스포트

transport.sendto()	데이터를 원격 피어로 보냅니다.
transport.abort()	즉시 트랜스포트를 닫습니다.

서브 프로세스에 대한 저수준 트랜스포트 추상화. *loop.subprocess_exec()* 와 *loop.subprocess_shell()*가 반환합니다:

서브 프로세스 트랜스포트

transport.get_pid()	서브 프로세스의 프로세스 ID를 돌려줍니다.
transport.get_pipe_transport()	요청한 통신 파이프 (*stdin*, *stdout* 또는 *stderr*) 에 대한 트랜스포트를 반환합니다.
transport.get_returncode()	서브 프로세스 반환 코드를 돌려줍니다.
transport.kill()	서브 프로세스를 죽입니다.
transport.send_signal()	서브 프로세스에 시그널을 보냅니다.
transport.terminate()	서브 프로세스를 중지합니다.
transport.close()	서브 프로세스를 죽이고 모든 파이프를 닫습니다.

프로토콜

프로토콜 클래스는 다음 **콜백 메서드를** 구현할 수 있습니다:

callback *connection_made()*	연결이 이루어질 때 호출됩니다.
callback *connection_lost()*	연결이 끊어지거나 닫힐 때 호출됩니다.
callback *pause_writing()*	트랜스포트 버퍼가 높은 수위를 초과할 때 호출됩니다.
callback *resume_writing()*	트랜스포트 버퍼가 낮은 수위 아래로 내려갈 때 호출됩니다.

스트리밍 프로토콜 (TCP, 유닉스 소켓, 파이프)

callback *data_received()*	어떤 데이터가 수신될 때 호출됩니다.
callback *eof_received()*	EOF가 수신될 때 호출됩니다.

버퍼 된 스트리밍 프로토콜

callback *get_buffer()*	새로운 수신 버퍼를 할당하기 위해서 호출됩니다.
callback *buffer_updated()*	수신된 데이터로 버퍼가 갱신될 때 호출됩니다.
callback *eof_received()*	EOF가 수신될 때 호출됩니다.

데이터 그램 프로토콜

callback *datagram_received()*	데이터 그램이 수신될 때 호출됩니다.
callback *error_received()*	이전의 송신이나 수신 연산이 *OSError*를 일으킬 때 호출됩니다.

서브 프로세스 프로토콜

callback *pipe_data_received()*	자식 프로세스가 *stdout* 이나 *stderr* 파이프에 데이터를 쓸 때 호출됩니다.
callback *pipe_connection_lost()*	자식 프로세스와 통신하는 파이프 중 하나가 닫힐 때 호출됩니다.
callback *process_exited()*	자식 프로세스가 종료할 때 호출됩니다. *pipe_data_received()* 와 *pipe_connection_lost()* 메서드 전에 호출될 수 있습니다.

이벤트 루프 정책

정책은 *asyncio.get_event_loop()*와 같은 함수의 동작을 변경하는 저수준 메커니즘입니다. 자세한 내용은 주 정책 절을 참조하십시오.

정책 액세스하기

asyncio.get_event_loop_policy()	현재 프로세스 전반의 정책을 돌려줍니다.
asyncio.set_event_loop_policy()	새로운 프로세스 전반의 정책을 설정합니다.
AbstractEventLoopPolicy	정책 객체의 베이스 클래스.

19.1.17 asyncio로 개발하기

비동기 프로그래밍은 고전적인 "순차적" 프로그래밍과 다릅니다.

이 페이지는 흔한 실수와 함정을 나열하고, 이를 피하는 방법을 설명합니다.

디버그 모드

기본적으로 asyncio는 프로덕션 모드로 실행됩니다. 개발을 쉽게 하려고 asyncio에는 디버그 모드를 제공합니다.

여러 가지 방법으로 asyncio 디버그 모드를 활성화할 수 있습니다:

- `PYTHONASYNCIODEBUG` 환경 변수를 `1`로 설정.
- 파이썬 개발 모드 사용.
- `debug=True`를 *asyncio.run()*로 전달.
- *loop.set_debug()*를 호출.

디버그 모드를 활성화하는 것 외에도, 다음을 고려하십시오:

- *asyncio* 로거의 로그 수준을 *logging.DEBUG*로 설정, 예를 들어 응용 프로그램 시작 시 다음 코드 조각을 실행할 수 있습니다:

```
logging.basicConfig(level=logging.DEBUG)
```

- *ResourceWarning* 경고를 표시하도록 *warnings* 모듈을 구성. 이렇게 하는 한 가지 방법은 `-W default` 명령 줄 옵션을 사용하는 것입니다.

디버그 모드가 활성화되면:

- 많은 스레드 안전하지 않은 asyncio API(*loop.call_soon()*과 *loop.call_at()* 메서드와 같은)가 잘못된 스레드에서 호출될 때 예외를 발생시킵니다.
- I/O 선택기의 실행 시간은 I/O 연산 수행에 너무 오래 걸리면 로그 됩니다.
- 100 밀리 초 보다 오래 걸리는 콜백이 로그 됩니다. *loop.slow_callback_duration* 어트리뷰트는 "느린" 것으로 간주할 최소 실행 시간(초)을 설정하는 데 사용될 수 있습니다.

동시성과 다중 스레드

이벤트 루프는 스레드(일반적으로 주 스레드)에서 실행되며 그 스레드에서 모든 콜백과 태스크를 실행합니다. 태스크가 이벤트 루프에서 실행되는 동안, 다른 태스크는 같은 스레드에서 실행될 수 없습니다. 태스크가 `await` 표현식을 실행하면, 실행 중인 태스크가 일시 중지되고 이벤트 루프는 다음 태스크를 실행합니다.

다른 OS 스레드에서 콜백을 예약하려면, *loop.call_soon_threadsafe()* 메서드를 사용해야 합니다. 예:

```
loop.call_soon_threadsafe(callback, *args)
```

거의 모든 asyncio 객체는 스레드 안전하지 않습니다. 태스크나 콜백 외부에서 작동하는 코드가 없으면 일반적으로 문제가 되지 않습니다. 그러한 코드가 저수준 asyncio API를 호출해야 하면, `loop.call_soon_threadsafe()` 메서드를 사용해야 합니다, 예를 들어:

```
loop.call_soon_threadsafe(fut.cancel)
```

다른 OS 스레드에서 코루틴 객체를 예약하려면, `run_coroutine_threadsafe()` 함수를 사용해야 합니다. 결과에 액세스할 수 있도록 `concurrent.futures.Future`를 반환합니다:

```
async def coro_func():
    return await asyncio.sleep(1, 42)

# 나중에 다른 OS 스레드에서:

future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
# 결과를 기다립니다:
result = future.result()
```

시그널을 처리하려면, 이벤트 루프를 메인 스레드에서 실행해야 합니다.

The `loop.run_in_executor()` method can be used with a `concurrent.futures.ThreadPoolExecutor` or `InterpreterPoolExecutor` to execute blocking code in a different OS thread without blocking the OS thread that the event loop runs in.

현재 다른 프로세스(가령 `multiprocessing`으로 시작된 프로세스)에서 직접 코루틴이나 콜백을 예약할 방법은 없습니다. 이벤트 루프 메서드 섹션은 이벤트 루프를 블록하지 않고 파이프를 읽고 파일 기술자를 감시할 수 있는 API를 나열합니다. 또한 asyncio의 서브 프로세스 API는 프로세스를 시작하고 이벤트 루프에서 프로세스와 통신하는 방법을 제공합니다. 마지막으로, 앞서 언급한 `loop.run_in_executor()` 메서드를 `concurrent.futures.ProcessPoolExecutor`와 함께 사용하여 다른 프로세스에서 코드를 실행할 수도 있습니다.

블로킹 코드 실행하기

블로킹 (CPU 병목) 코드는 직접 호출하면 안 됩니다. 예를 들어, 함수가 CPU 집약적인 계산을 1초 동안 수행하면, 모든 동시 asyncio 태스크와 IO 연산이 1초 지연됩니다.

An executor can be used to run a task in a different thread, including in a different interpreter, or even in a different process to avoid blocking the OS thread with the event loop. See the `loop.run_in_executor()` method for more details.

로깅

asyncio는 `logging` 모듈을 사용하고, 모든 로깅은 `"asyncio"` 로거를 통해 수행됩니다.

기본 로그 수준은 `logging.INFO`며, 쉽게 조정할 수 있습니다:

```
logging.getLogger("asyncio").setLevel(logging.WARNING)
```

Network logging can block the event loop. It is recommended to use a separate thread for handling logs or use non-blocking IO. For example, see blocking-handlers.

await 하지 않은 코루틴 감지

코루틴 함수가 호출되었지만 기다리지 않을 때(예를 들어, `await coro()` 대신 `coro()`)나 코루틴이 `asyncio.create_task()`로 예약되지 않으면 asyncio가 `RuntimeWarning`을 방출합니다:

```
import asyncio

async def test():
    print("never scheduled")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
async def main():
    test()

asyncio.run(main())
```

출력:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
  test()
```

디버그 모드에서의 출력:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
Coroutine created at (most recent call last)
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

  < .. >

  File "../t.py", line 7, in main
    test()
  test()
```

일반적인 수정법은 코루틴을 await 하거나 *asyncio.create_task()* 함수를 호출하는 것입니다:

```
async def main():
    await test()
```

전달되지 않은 예외 감지

Future.set_exception() 가 호출되었지만, Future 객체가 await 되지 않으면, 예외는 절대로 사용자 코드로 전파되지 않습니다. 이럴 때, Future 객체가 가비지 수집될 때 asyncio가 로그 메시지를 출력합니다.

처리되지 않은 예외의 예:

```
import asyncio

async def bug():
    raise Exception("not consumed")

async def main():
    asyncio.create_task(bug())

asyncio.run(main())
```

출력:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
  exception=Exception('not consumed')>

Traceback (most recent call last):
  File "test.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

태스크가 만들어진 곳의 트레이스백을 얻으려면 디버그 모드를 활성화하세요:

```
asyncio.run(main(), debug=True)
```

디버그 모드에서의 출력:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
    exception=Exception('not consumed') created at asyncio/tasks.py:321>

source_traceback: Object created at (most recent call last):
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

< .. >

Traceback (most recent call last):
  File "../t.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

참고

asyncio의 소스 코드는 Lib/asyncio/에서 찾을 수 있습니다.

19.2 socket — Low-level networking interface

소스 코드: Lib/socket.py

이 모듈은 BSD *socket* 인터페이스에 대한 액세스를 제공합니다. 모든 현대 유닉스 시스템, 윈도우, MacOS, 그리고 아마 추가 플랫폼에서 사용할 수 있습니다.

참고

호출이 운영 체제 소켓 API로 이루어지기 때문에, 일부 동작은 플랫폼에 따라 다를 수 있습니다.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python' s object-oriented style: the `socket()` function returns a *socket object* whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with `read()` and `write()` operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

더 보기

모듈 *socketserver*
: 네트워크 서버 작성을 단순화하는 클래스.

모듈 *ssl*
: 소켓 객체용 TLS/SSL 래퍼.

19.2.1 소켓 패밀리

시스템과 빌드 옵션에 따라, 다양한 소켓 패밀리가 이 모듈에서 지원됩니다.

특정 소켓 객체가 요구하는 주소 형식은 소켓 객체를 만들 때 지정된 주소 패밀리에 따라 자동으로 선택됩니다. 소켓 주소는 다음과 같이 표현됩니다:

- 파일 시스템 노드에 바인드 된 *AF_UNIX* 소켓의 주소는 파일 시스템 인코딩과 `'surrogateescape'` 에러 처리기(**PEP 383**을 참조하세요)를 사용하는 문자열로 표현됩니다. 리눅스의 추상 이름 공간(abstract namespace)에 있는 주소는 처음에 널 바이트가 있는 바이트열류 객체로 반환됩니다; 이 이름 공간의 소켓은 일반 파일 시스템 소켓과 통신 할 수 있으므로, 리눅스에서 실행하려는 프로그램은 두 가지 유형의 주소를 모두 다뤄야 할 수도 있습니다. 문자열이나 바이트열류 객체는 인자로 전달할 때 두 가지 유형의 주소에 모두 사용할 수 있습니다.

  버전 3.3에서 변경: 이전에는, *AF_UNIX* 소켓 경로가 UTF-8 인코딩을 사용한다고 가정했습니다.

  버전 3.5에서 변경: 이제 쓰기 가능한 바이트열류 객체를 받아들입니다.

- A pair `(host, port)` is used for the *AF_INET* address family, where *host* is a string representing either a hostname in internet domain notation like `'daring.cwi.nl'` or an IPv4 address like `'100.50.200.5'`, and *port* is an integer.

  - IPv4 주소의 경우, 호스트 주소 대신 두 개의 특수 형식이 허용됩니다: `''`는 모든 인터페이스에 바인딩하는 데 사용되는 `INADDR_ANY`를 나타내며 `'<broadcast>'` 문자열은 `INADDR_BROADCAST`를 나타냅니다. 이 동작은 IPv6와 호환되지 않으므로, 여러분의 파이썬 프로그램에서 IPv6를 지원하려는 경우에는 이것들을 사용하지 않을 수 있습니다.

- *AF_INET6* 주소 패밀리의 경우, 4-튜플 `(host, port, flowinfo, scope_id)`가 사용됩니다. 여기서 *flowinfo* 와 *scope_id*는 C에서 `struct sockaddr_in6`의 `sin6_flowinfo` 와 `sin6_scope_id` 멤버를 나타냅니다. `socket` 모듈 메서드의 경우, *flowinfo* 와 *scope_id*는 이전 버전과의 호환성을 위해 생략할 수 있습니다. 그러나, *scope_id*를 생략하면 스코프가 지정된(scoped) IPv6 주소를 조작하는 데 문제가 발생할 수 있습니다.

  버전 3.7에서 변경: 멀티캐스트 주소(의미 있는 *scope_id*를 가진)의 경우, *address*에는 `%scope_id` (또는 `zone id`) 부분이 포함될 수 없습니다. 이 정보는 불필요하므로 안전하게 생략할 수 있습니다(권장 사항).

- `AF_NETLINK` 소켓은 `(pid, groups)` 쌍으로 표현됩니다.

- TIPC에 대한 리눅스 전용 지원은 `AF_TIPC` 주소 패밀리를 사용하여 사용할 수 있습니다. TIPC는 클러스터 된 컴퓨터 환경에서 사용하도록 설계된 개방형 비 IP 기반 네트워크 프로토콜입니다. 주소는 튜플로 표현되며 필드는 주소 유형에 따라 다릅니다. 일반적인 튜플 형식은 `(addr_type, v1, v2, v3 [, scope])` 입니다. 이때:

  - *addr_type*은 `TIPC_ADDR_NAMESEQ`, `TIPC_ADDR_NAME` 또는 `TIPC_ADDR_ID` 중 하나입니다.

  - *scope*는 `TIPC_ZONE_SCOPE`, `TIPC_CLUSTER_SCOPE` 또는 `TIPC_NODE_SCOPE` 중 하나입니다.

  - *addr_type*이 `TIPC_ADDR_NAME`이면, *v1*은 서버 유형이고, *v2*는 포트 식별자이며, *v3*은 0이어야 합니다.

    *addr_type*이 `TIPC_ADDR_NAMESEQ`면, *v1*은 서버 유형이고, *v2*는 하위 포트 번호이며, *v3*는 상위 포트 번호입니다.

    *addr_type*이 `TIPC_ADDR_ID`면, *v1*은 노드이고, *v2*는 참조이며, *v3*는 0으로 설정되어야 합니다.

- 튜플 `(interface, )`가 *AF_CAN* 주소 패밀리에 사용됩니다. 여기서 *interface*는 `'can0'`과 같은 네트워크 인터페이스 이름을 나타내는 문자열입니다. 네트워크 인터페이스 이름 `''`는 이 패밀리의 모든 네트워크 인터페이스에서 패킷을 수신하는 데 사용할 수 있습니다.

  - *CAN_ISOTP* 프로토콜은 튜플 `(interface, rx_addr, tx_addr)`를 요구하는데, 두 개의 추가 매개 변수는 모두 CAN 식별자(표준 또는 확장)를 나타내는 부호 없는 long 정수입니다.

  - *CAN_J1939* 프로토콜에는 `(interface, name, pgn, addr)`가 필요한데, 여기서 추가 파라미터는 ECU 이름을 나타내는 64 비트 부호 없는 정수, PGN(Parameter Group Number)을 나타내는 32비트 부호 없는 정수 및 주소를 나타내는 8비트 정수입니다.

- A string or a tuple `(id, unit)` is used for the `SYSPROTO_CONTROL` protocol of the `PF_SYSTEM` family. The string is the name of a kernel control using a dynamically assigned ID. The tuple can be used if ID and unit number of the kernel control are known or if a registered ID is used.

  Added in version 3.3.

- *AF_BLUETOOTH*는 다음 프로토콜 및 주소 형식을 지원합니다:
  - *BTPROTO_L2CAP* accepts a tuple `(bdaddr, psm[, cid[, bdaddr_type]])` where:
    - `bdaddr` is a string specifying the Bluetooth address.
    - `psm` is an integer specifying the Protocol/Service Multiplexer.
    - `cid` is an optional integer specifying the Channel Identifier. If not given, defaults to zero.
    - `bdaddr_type` is an optional integer specifying the address type; one of *BDADDR_BREDR* (default), *BDADDR_LE_PUBLIC*, *BDADDR_LE_RANDOM*.

    버전 3.14에서 변경: Added `cid` and `bdaddr_type` fields.
  - *BTPROTO_RFCOMM*은 `(bdaddr, channel)`를 받아들입니다. 여기서 `bdaddr`은 문자열 블루투스 주소이고 `channel`은 정수입니다.
  - *BTPROTO_HCI* accepts a format that depends on your OS.
    - On Linux it accepts an integer `device_id` or a tuple `(device_id, [channel])` where `device_id` specifies the number of the Bluetooth device, and `channel` is an optional integer specifying the HCI channel (*HCI_CHANNEL_RAW* by default).
    - On FreeBSD, NetBSD and DragonFly BSD it accepts `bdaddr` where `bdaddr` is the Bluetooth address as a string.

    버전 3.2에서 변경: NetBSD 및 DragonFlyBSD 지원이 추가되었습니다.

    버전 3.13.3에서 변경: FreeBSD support added.

    버전 3.14에서 변경: Added `channel` field. `device_id` not packed in a tuple is now accepted.
  - *BTPROTO_SCO* accepts `bdaddr` where `bdaddr` is the Bluetooth address as a string or a *bytes* object. (ex. `'12:23:34:45:56:67'` or `b'12:23:34:45:56:67'`)

    버전 3.14에서 변경: FreeBSD support added.

- *AF_ALG*는 커널 암호 인터페이스에 기반한 리눅스 전용 소켓입니다. 알고리즘 소켓은 2~4개의 요소를 갖는 `(type, name [, feat [, mask]])` 튜플로 구성됩니다. 여기서:
  - *type*은 문자열의 알고리즘 유형입니다, 예를 들어, `aead`, `hash`, `skcipher` 또는 `rng`.
  - *name*은 알고리즘 이름과 연산 모드 문자열입니다, 예를 들어, `sha256`, `hmac(sha256)`, `cbc(aes)` 또는 `drbg_nopr_ctr_aes256`.
  - *feat* 과 *mask*는 부호 없는 32비트 정수입니다.

  가용성: Linux >= 2.6.38.

  Some algorithm types require more recent Kernels.

  Added in version 3.6.

- *AF_VSOCK*은 가상 기계와 호스트가 통신할 수 있게 합니다. 소켓은 `(CID, port)` 튜플로 표현되는데, 컨텍스트 ID 또는 CID와 port는 정수입니다.

  가용성: Linux >= 3.9

  See *vsock(7)*

  Added in version 3.7.

- *AF_PACKET* is a low-level interface directly to network devices. The addresses are represented by the tuple `(ifname, proto[, pkttype[, hatype[, addr]]])` where:
  - *ifname* - 장치 이름을 지정하는 문자열

- *proto* - The Ethernet protocol number. May be `ETH_P_ALL` to capture all protocols, one of the *ETHERTYPE_* constants* or any other Ethernet protocol number.
- *pkttype* - 패킷 유형을 지정하는 선택적 정수.:
  * `PACKET_HOST` (기본값) - 로컬 호스트로 향하는 패킷.
  * `PACKET_BROADCAST` - 물리 계층 브로드캐스트 패킷.
  * `PACKET_MULTICAST` - Packet sent to a physical-layer multicast address.
  * `PACKET_OTHERHOST` - 무차별 모드의 장치 관리자에 의해 포착된 다른 호스트로 향하는 패킷.
  * `PACKET_OUTGOING` - 패킷 소켓으로 루프 백 된 로컬 호스트에서 시작된 패킷.
- *hatype* - ARP 하드웨어 주소 유형을 지정하는 선택적 정수.
- *addr* - 하드웨어 물리 주소를 지정하는 선택적 바이트열류 객체, 해석은 장치에 따라 다릅니다.

가용성: Linux >= 2.2.

- `AF_QIPCRTR`는 Qualcomm 플랫폼의 코 프로세서에서 실행되는 서비스와 통신하기 위한 리눅스 전용 소켓 기반 인터페이스입니다. 주소 패밀리는 `(node, port)` 튜플로 표현되는데, *node*와 *port*는 음수가 아닌 정수입니다.

  가용성: Linux >= 4.7.

  Added in version 3.8.

- `IPPROTO_UDPLITE`는 UDP의 변형으로, 체크섬으로 커버되는 패킷 부분을 지정할 수 있습니다. 변경할 수 있는 두 개의 소켓 옵션이 추가되었습니다. `self.setsockopt(IPPROTO_UDPLITE, UDPLITE_SEND_CSCOV, length)`는 체크섬으로 커버되는 나가는 패킷 부분을 변경하고 `self.setsockopt(IPPROTO_UDPLITE, UDPLITE_RECV_CSCOV, length)`는 너무 적은 데이터를 커버하는 패킷을 걸러냅니다. 두 경우 모두 `length`는 `range(8, 2**16, 8)`에 있어야합니다.

  이러한 소켓은 IPv4의 경우 `socket(AF_INET, SOCK_DGRAM, IPPROTO_UDPLITE)` 또는 IPv6의 경우 `socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDPLITE)`로 구성되어야 합니다.

  가용성: Linux >= 2.6.20, FreeBSD >= 10.1

  Added in version 3.9.

- `AF_HYPERV` is a Windows-only socket based interface for communicating with Hyper-V hosts and guests. The address family is represented as a `(vm_id, service_id)` tuple where the `vm_id` and `service_id` are UUID strings.

  The `vm_id` is the virtual machine identifier or a set of known VMID values if the target is not a specific virtual machine. Known VMID constants defined on `socket` are:

  - `HV_GUID_ZERO`
  - `HV_GUID_BROADCAST`
  - `HV_GUID_WILDCARD` - Used to bind on itself and accept connections from all partitions.
  - `HV_GUID_CHILDREN` - Used to bind on itself and accept connection from child partitions.
  - `HV_GUID_LOOPBACK` - Used as a target to itself.
  - `HV_GUID_PARENT` - When used as a bind accepts connection from the parent partition. When used as an address target it will connect to the parent partition.

  The `service_id` is the service identifier of the registered service.

  Added in version 3.12.

IPv4/v6 소켓 주소의 *host* 부분에 호스트 명을 사용하면, 파이썬이 DNS 결정에서 반환된 첫 번째 주소를 사용하기 때문에, 프로그램은 비결정적인 동작을 보일 수 있습니다. 소켓 주소는 DNS 결정 결과 및/또는 호스트 구성에 따라 실제 IPv4/v6 주소로 다르게 결정됩니다. 결정론적 동작을 위해서는 *host* 부분에 숫자 주소를 사용하십시오.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised. Errors related to socket or address semantics raise *OSError* or one of its subclasses.

비 블로킹 모드는 *setblocking()*을 통해 지원됩니다. 시간제한을 기반으로 하는 일반화는 *settimeout()*을 통해 지원됩니다.

19.2.2 모듈 내용

모듈 *socket*은 다음 요소를 노출합니다.

예외

exception socket.**error**

*OSError*의 폐지된 별칭.

버전 3.3에서 변경: **PEP 3151**을 따라, 이 클래스는 *OSError*의 별칭이 되었습니다.

exception socket.**herror**

*OSError*의 서브 클래스, 이 예외는 주소 관련 에러에서 발생합니다. 즉 *gethostbyname_ex()*와 *gethostbyaddr()*를 포함하는 POSIX C API의 *h_errno*를 사용하는 함수들. 수반되는 값은 라이브러리 호출이 반환한 에러를 나타내는 `(h_errno, string)` 쌍입니다. *h_errno*는 숫자 값이고, *string*은 `hstrerror()` C 함수에 의해 반환된 *h_errno*의 설명을 나타냅니다.

버전 3.3에서 변경: 이 클래스는 *OSError*의 서브 클래스가 되었습니다.

exception socket.**gaierror**

A subclass of *OSError*, this exception is raised for address-related errors by *getaddrinfo()* and *getnameinfo()*. The accompanying value is a pair `(error, string)` representing an error returned by a library call. *string* represents the description of *error*, as returned by the `gai_strerror()` C function. The numeric *error* value will match one of the `EAI_*` constants defined in this module.

버전 3.3에서 변경: 이 클래스는 *OSError*의 서브 클래스가 되었습니다.

exception socket.**timeout**

A deprecated alias of *TimeoutError*.

*OSError*의 서브 클래스, 이 예외는 앞서 *settimeout()* 호출을 통해 (또는 묵시적으로 *setdefaulttimeout()*를 통해) 시간제한이 활성화된 소켓에서 시간 초과가 일어날 때 발생합니다. 수반되는 값은 현재는 항상 "timed out" 값을 갖는 문자열입니다.

버전 3.3에서 변경: 이 클래스는 *OSError*의 서브 클래스가 되었습니다.

버전 3.10에서 변경: This class was made an alias of *TimeoutError*.

상수

AF_* 와 SOCK_* 상수는 이제 `AddressFamily` 와 `SocketKind` *IntEnum* 컬렉션입니다.

Added in version 3.4.

socket.**AF_UNIX**

socket.**AF_INET**

socket.**AF_INET6**

These constants represent the address (and protocol) families, used for the first argument to *socket()*. If the *AF_UNIX* constant is not defined then this protocol is unsupported. More constants may be available depending on the system.

socket.**AF_UNSPEC**

AF_UNSPEC means that *getaddrinfo()* should return socket addresses for any address family (either IPv4, IPv6, or any other) that can be used.

socket.**SOCK_STREAM**

socket.**SOCK_DGRAM**

socket.**SOCK_RAW**

socket.**SOCK_RDM**

socket.**SOCK_SEQPACKET**

These constants represent the socket types, used for the second argument to *socket()*. More constants may be available depending on the system. (Only *SOCK_STREAM* and *SOCK_DGRAM* appear to be generally useful.)

socket.**SOCK_CLOEXEC**

socket.**SOCK_NONBLOCK**

이 두 상수는, 정의되었다면, 소켓 유형과 결합하여 일부 플래그를 원자 적으로 설정할 수 있도록 합니다 (따라서 경쟁 조건의 가능성과 별도 호출의 필요성을 피할 수 있습니다).

> **더 보기**
>
> Secure File Descriptor Handling for a more thorough explanation.

가용성: Linux >= 2.6.27.

Added in version 3.2.

SO_*

socket.**SOMAXCONN**

MSG_*

SOL_*

SCM_*

IPPROTO_*

IPPORT_*

INADDR_*

IP_*

IPV6_*

EAI_*

AI_*

NI_*

TCP_*

Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the *setsockopt()* and *getsockopt()* methods of socket objects. In most cases, only those symbols that are defined in the Unix header files are defined; for a few symbols, default values are provided.

버전 3.6에서 변경: SO_DOMAIN, SO_PROTOCOL, SO_PEERSEC, SO_PASSSEC, TCP_USER_TIMEOUT, TCP_CONGESTION가 추가되었습니다.

버전 3.6.5에서 변경: Added support for TCP_FASTOPEN, TCP_KEEPCNT on Windows platforms when available.

버전 3.7에서 변경: TCP_NOTSENT_LOWAT가 추가되었습니다.

Added support for TCP_KEEPIDLE, TCP_KEEPINTVL on Windows platforms when available.

버전 3.10에서 변경: IP_RECVTOS was added. Added TCP_KEEPALIVE. On MacOS this constant can be used in the same way that TCP_KEEPIDLE is used on Linux.

버전 3.11에서 변경: Added TCP_CONNECTION_INFO. On MacOS this constant can be used in the same way that TCP_INFO is used on Linux and BSD.

버전 3.12에서 변경: Added SO_RTABLE and SO_USER_COOKIE. On OpenBSD and FreeBSD respectively those constants can be used in the same way that SO_MARK is used on Linux. Also added missing TCP socket options from Linux: TCP_MD5SIG, TCP_THIN_LINEAR_TIMEOUTS, TCP_THIN_DUPACK, TCP_REPAIR, TCP_REPAIR_QUEUE, TCP_QUEUE_SEQ, TCP_REPAIR_OPTIONS, TCP_TIMESTAMP, TCP_CC_INFO,

TCP_SAVE_SYN, TCP_SAVED_SYN, TCP_REPAIR_WINDOW, TCP_FASTOPEN_CONNECT, TCP_ULP, TCP_MD5SIG_EXT, TCP_FASTOPEN_KEY, TCP_FASTOPEN_NO_COOKIE, TCP_ZEROCOPY_RECEIVE, TCP_INQ, TCP_TX_DELAY. Added IP_PKTINFO, IP_UNBLOCK_SOURCE, IP_BLOCK_SOURCE, IP_ADD_SOURCE_MEMBERSHIP, IP_DROP_SOURCE_MEMBERSHIP.

버전 3.13에서 변경: Added SO_BINDTOIFINDEX. On Linux this constant can be used in the same way that SO_BINDTODEVICE is used, but with the index of a network interface instead of its name.

버전 3.14에서 변경: Added missing IP_FREEBIND, IP_RECVERR, IPV6_RECVERR, IP_RECVTTL, and IP_RECVORIGDSTADDR on Linux.

버전 3.14에서 변경: Added support for TCP_QUICKACK on Windows platforms when available.

socket.**AF_CAN**

socket.**PF_CAN**

SOL_CAN_*

CAN_*

리눅스 설명서에 설명되어있는 이 형식의 많은 상수는 소켓 모듈에도 정의되어 있습니다.

가용성: Linux >= 2.6.25, NetBSD >= 8.

Added in version 3.3.

버전 3.11에서 변경: NetBSD support was added.

버전 3.14에서 변경: Restored missing CAN_RAW_ERR_FILTER on Linux.

socket.**CAN_BCM**

CAN_BCM_*

CAN 프로토콜 패밀리에서 CAN_BCM은 브로드캐스트 관리자 (Broadcast Manager, BCM) 프로토콜입니다. 리눅스 설명서에서 설명된 브로드캐스트 관리자 상수도 소켓 모듈에 정의되어 있습니다.

가용성: Linux >= 2.6.25.

> **참고**
>
> CAN_BCM_CAN_FD_FRAME 플래그는 리눅스 >= 4.8 에서만 사용 가능합니다.

Added in version 3.4.

socket.**CAN_RAW_FD_FRAMES**

CAN_RAW 소켓에서 CAN FD 지원을 활성화합니다. 기본적으로 비활성화되어 있습니다. 여러분의 응용 프로그램이 CAN과 CAN FD 프레임을 모두 보낼 수 있도록 합니다; 그러나 소켓에서 읽을 때 CAN과 CAN FD 프레임을 모두 받아들여야 합니다.

이 상수는 리눅스 설명서에 설명되어 있습니다.

가용성: Linux >= 3.6.

Added in version 3.5.

socket.**CAN_RAW_JOIN_FILTERS**

주어진 모든 CAN 필터와 일치하는 CAN 프레임 만 사용자 공간으로 전달되도록 적용된 CAN 필터를 결합합니다.

이 상수는 리눅스 설명서에 설명되어 있습니다.

가용성: Linux >= 4.1.

Added in version 3.9.

socket.**CAN_ISOTP**

CAN 프로토콜 패밀리의 CAN_ISOTP는 ISO-TP (ISO 15765-2) 프로토콜입니다. ISO-TP 상수는 리눅스 설명서에 설명되어 있습니다.

가용성: Linux >= 2.6.25.

Added in version 3.7.

socket.**CAN_J1939**

CAN 프로토콜 패밀리의 CAN_J1939는 SAE J1939 프로토콜입니다. J1939 상수는 리눅스 설명서에 설명되어 있습니다.

가용성: Linux >= 5.4.

Added in version 3.9.

socket.**AF_DIVERT**

socket.**PF_DIVERT**

These two constants, documented in the FreeBSD divert(4) manual page, are also defined in the socket module.

가용성: FreeBSD >= 14.0.

Added in version 3.12.

socket.**AF_PACKET**

socket.**PF_PACKET**

PACKET_*

리눅스 설명서에 설명되어있는 이 형식의 많은 상수는 소켓 모듈에도 정의되어 있습니다.

가용성: Linux >= 2.2.

socket.**ETH_P_ALL**

`ETH_P_ALL` can be used in the `socket` constructor as *proto* for the `AF_PACKET` family in order to capture every packet, regardless of protocol.

For more information, see the `packet(7)` manpage.

가용성: Linux.

Added in version 3.12.

socket.**AF_RDS**

socket.**PF_RDS**

socket.**SOL_RDS**

RDS_*

리눅스 설명서에 설명되어있는 이 형식의 많은 상수는 소켓 모듈에도 정의되어 있습니다.

가용성: Linux >= 2.6.30.

Added in version 3.3.

socket.**SIO_RCVALL**

socket.**SIO_KEEPALIVE_VALS**

socket.**SIO_LOOPBACK_FAST_PATH**

RCVALL_*

윈도우 WSAIoctl()용 상수. 이 상수는 소켓 객체의 `ioctl()` 메서드에 대한 인자로 사용됩니다.

버전 3.6에서 변경: `SIO_LOOPBACK_FAST_PATH`가 추가되었습니다.

TIPC_*

TIPC 관련 상수. C 소켓 API에서 내보낸 것과 일치합니다. 자세한 정보는 TIPC 설명서를 참조하십시오.

socket.**AF_ALG**

socket.**SOL_ALG**

ALG_*

리눅스 커널 암호화용 상수.

가용성: Linux >= 2.6.38.

Added in version 3.6.

socket.**AF_VSOCK**

socket.**IOCTL_VM_SOCKETS_GET_LOCAL_CID**

VMADDR*

SO_VM*

리눅스 호스트/게스트 통신용 상수.

가용성: Linux >= 4.8.

Added in version 3.7.

socket.**AF_LINK**

가용성: BSD, macOS.

Added in version 3.4.

socket.**has_ipv6**

이 상수는 이 플랫폼에서 IPv6가 지원되는지를 나타내는 논릿값을 포함합니다.

socket.**AF_BLUETOOTH**

socket.**BTPROTO_L2CAP**

socket.**BTPROTO_RFCOMM**

socket.**BTPROTO_HCI**

socket.**BTPROTO_SCO**

Integer constants for use with Bluetooth addresses.

socket.**BDADDR_ANY**

socket.**BDADDR_LOCAL**

이들은 특수한 의미를 지닌 블루투스 주소를 포함하는 문자열 상수입니다. 예를 들어, *BDADDR_ANY*는 바인딩 소켓을 *BTPROTO_RFCOMM*로 지정할 때 임의의(any) 주소를 나타내는 데 사용할 수 있습니다.

socket.**BDADDR_BREDR**

socket.**BDADDR_LE_PUBLIC**

socket.**BDADDR_LE_RANDOM**

These constants describe the Bluetooth address type when binding or connecting a *BTPROTO_L2CAP* socket.

가용성: Linux, FreeBSD

Added in version 3.14.

socket.**SOL_RFCOMM**

socket.**SOL_L2CAP**

socket.**SOL_HCI**

socket.**SOL_SCO**

socket.**SOL_BLUETOOTH**

Used in the level argument to the *setsockopt()* and *getsockopt()* methods of Bluetooth socket objects.

SOL_BLUETOOTH is only available on Linux. Other constants are available if the corresponding protocol is supported.

SO_L2CAP_*

socket.**L2CAP_LM**

L2CAP_LM_*

SO_RFCOMM_*

RFCOMM_LM_*

SO_SCO_*

SO_BTH_*

BT_*

Used in the option name and value argument to the *setsockopt()* and *getsockopt()* methods of Bluetooth socket objects.

`BT_*` and *L2CAP_LM* are only available on Linux. `SO_BTH_*` are only available on Windows. Other constants may be available on Linux and various BSD platforms.

Added in version 3.14.

socket.**HCI_FILTER**

socket.**HCI_TIME_STAMP**

socket.**HCI_DATA_DIR**

socket.**SO_HCI_EVT_FILTER**

socket.**SO_HCI_PKT_FILTER**

Option names for use with *BTPROTO_HCI*. Availability and format of the option values depend on platform.

버전 3.14에서 변경: Added `SO_HCI_EVT_FILTER` and `SO_HCI_PKT_FILTER` on NetBSD and DragonFly BSD. Added `HCI_DATA_DIR` on FreeBSD, NetBSD and DragonFly BSD.

socket.**HCI_DEV_NONE**

The `device_id` value used to create an HCI socket that isn' t specific to a single Bluetooth adapter.

가용성: Linux

Added in version 3.14.

socket.**HCI_CHANNEL_RAW**

socket.**HCI_CHANNEL_USER**

socket.**HCI_CHANNEL_MONITOR**

socket.**HCI_CHANNEL_CONTROL**

socket.**HCI_CHANNEL_LOGGING**

Possible values for `channel` field in the *BTPROTO_HCI* address.

가용성: Linux

Added in version 3.14.

socket.**AF_QIPCRTR**

원격 프로세서를 제공하는 서비스와 통신하는 데 사용되는 Qualcomm의 IPC 라우터 프로토콜용 상수.

가용성: Linux >= 4.7.

socket.**SCM_CREDS2**

socket.**LOCAL_CREDS**

socket.**LOCAL_CREDS_PERSISTENT**

LOCAL_CREDS and LOCAL_CREDS_PERSISTENT can be used with SOCK_DGRAM, SOCK_STREAM sockets, equivalent to Linux/DragonFlyBSD SO_PASSCRED, while LOCAL_CREDS sends the credentials at first read, LOCAL_CREDS_PERSISTENT sends for each read, SCM_CREDS2 must be then used for the latter for the message type.

Added in version 3.11.

가용성: FreeBSD.

socket.**SO_INCOMING_CPU**

Constant to optimize CPU locality, to be used in conjunction with `SO_REUSEPORT`.

Added in version 3.11.

가용성: Linux >= 3.9

socket.**SO_REUSEPORT_LB**

Constant to enable duplicate address and port bindings with load balancing.

Added in version 3.14.

가용성: FreeBSD >= 12.0

socket.**AF_HYPERV**

socket.**HV_PROTOCOL_RAW**

socket.**HVSOCKET_CONNECT_TIMEOUT**

socket.**HVSOCKET_CONNECT_TIMEOUT_MAX**

socket.**HVSOCKET_CONNECTED_SUSPEND**

socket.**HVSOCKET_ADDRESS_FLAG_PASSTHRU**

socket.**HV_GUID_ZERO**

socket.**HV_GUID_WILDCARD**

socket.**HV_GUID_BROADCAST**

socket.**HV_GUID_CHILDREN**

socket.**HV_GUID_LOOPBACK**

socket.**HV_GUID_PARENT**

Constants for Windows Hyper-V sockets for host/guest communications.

가용성: Windows.

Added in version 3.12.

socket.**ETHERTYPE_ARP**

socket.**ETHERTYPE_IP**

socket.**ETHERTYPE_IPV6**

socket.**ETHERTYPE_VLAN**

IEEE 802.3 protocol number. constants.

가용성: Linux, FreeBSD, macOS.

Added in version 3.12.

socket.**SHUT_RD**

socket.**SHUT_WR**

socket.**SHUT_RDWR**

These constants are used by the `shutdown()` method of socket objects.

가용성: not WASI.

함수

소켓 만들기

다음 함수는 모두 소켓 객체를 만듭니다.

class socket.**socket**(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)

지정된 주소 패밀리, 소켓 유형, 및 프로토콜 번호를 사용하여 새로운 소켓을 만듭니다. 주소 패밀리는 `AF_INET` (기본값), `AF_INET6`, `AF_UNIX`, `AF_CAN`, `AF_PACKET` 또는 `AF_RDS` 여야 합니다. 소켓 유형은 `SOCK_STREAM` (기본값), `SOCK_DGRAM`, `SOCK_RAW` 또는 기타 `SOCK_` 상수 중 하나여야 합니다. 프로토콜 번호는 일반적으로 0이며 생략될 수도 있고, 주소 패밀리가 `AF_CAN` 일 때 프로토콜은 `CAN_RAW`, `CAN_BCM`, `CAN_ISOTP` 또는 `CAN_J1939` 중 하나여야 합니다.

If *fileno* is specified, the values for *family*, *type*, and *proto* are auto-detected from the specified file descriptor. Auto-detection can be overruled by calling the function with explicit *family*, *type*, or *proto* arguments. This only affects how Python represents e.g. the return value of `socket.getpeername()` but not the actual OS

resource. Unlike `socket.fromfd()`, *fileno* will return the same socket and not a duplicate. This may help close a detached socket using `socket.close()`.

새로 만들어진 소켓은 상속 불가능합니다.

`self`, `family`, `type`, `protocol`를 인자로 감사 이벤트(*auditing event*) `socket.__new__`를 발생시킵니다.

버전 3.3에서 변경: AF_CAN 패밀리가 추가되었습니다. AF_RDS 패밀리가 추가되었습니다.

버전 3.4에서 변경: CAN_BCM 프로토콜이 추가되었습니다.

버전 3.4에서 변경: 반환된 소켓은 이제 상속 불가능합니다.

버전 3.7에서 변경: CAN_ISOTP 프로토콜이 추가되었습니다.

버전 3.7에서 변경: When `SOCK_NONBLOCK` or `SOCK_CLOEXEC` bit flags are applied to *type* they are cleared, and `socket.type` will not reflect them. They are still passed to the underlying system `socket()` call. Therefore,

```
sock = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM | socket.SOCK_NONBLOCK)
```

는 여전히 `SOCK_NONBLOCK`를 지원하는 OS에서 비 블로킹 소켓을 만들지만, `sock.type`은 `socket.SOCK_STREAM`로 설정됩니다.

버전 3.9에서 변경: CAN_J1939 프로토콜이 추가되었습니다.

버전 3.10에서 변경: The IPPROTO_MPTCP protocol was added.

`socket.`**`socketpair`**(*[family[, type[, proto]]]*)

Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the `socket()` function above. The default family is `AF_UNIX` if defined on the platform; otherwise, the default is `AF_INET`.

새로 만들어진 소켓은 상속 불가능합니다.

버전 3.2에서 변경: 반환된 소켓 객체는 이제 부분 집합이 아닌 전체 소켓 API를 지원합니다.

버전 3.4에서 변경: 반환된 소켓은 이제 상속 불가능합니다.

버전 3.5에서 변경: 윈도우 지원이 추가되었습니다.

`socket.`**`create_connection`**(*address, timeout=GLOBAL_DEFAULT, source_address=None, *, all_errors=False*)

Connect to a TCP service listening on the internet *address* (a 2-tuple `(host, port)`), and return the socket object. This is a higher-level function than `socket.connect()`: if *host* is a non-numeric hostname, it will try to resolve it for both `AF_INET` and `AF_INET6`, and then try to connect to all possible addresses in turn until a connection succeeds. This makes it easy to write clients that are compatible to both IPv4 and IPv6.

선택적 *timeout* 매개 변수를 전달하면 연결을 시도하기 전에 소켓 인스턴스의 시간제한을 설정합니다. *timeout*이 제공되지 않으면, `getdefaulttimeout()`에 의해 반환된 전역 기본 시간제한 설정이 사용됩니다.

제공되면, *source_address*는 연결하기 전에 소켓이 소스 주소로 바인드 할 2-튜플 `(host, port)` 여야 합니다. 호스트나 포트가 각각 '' 나 0이면 OS 기본 동작이 사용됩니다.

When a connection cannot be created, an exception is raised. By default, it is the exception from the last address in the list. If *all_errors* is `True`, it is an `ExceptionGroup` containing the errors of all attempts.

버전 3.2에서 변경: *source_address*가 추가되었습니다.

버전 3.11에서 변경: *all_errors* was added.

socket.**create_server**(*address, *, family=AF_INET, backlog=None, reuse_port=False, dualstack_ipv6=False*)

Convenience function which creates a TCP socket bound to *address* (a 2-tuple `(host, port)`) and returns the socket object.

family should be either `AF_INET` or `AF_INET6`. *backlog* is the queue size passed to `socket.listen()`; if not specified , a default reasonable value is chosen. *reuse_port* dictates whether to set the `SO_REUSEPORT` socket option.

If *dualstack_ipv6* is true, *family* is `AF_INET6` and the platform supports it the socket will be able to accept both IPv4 and IPv6 connections, else it will raise `ValueError`. Most POSIX platforms and Windows are supposed to support this functionality. When this functionality is enabled the address returned by `socket.getpeername()` when an IPv4 connection occurs will be an IPv6 address represented as an IPv4-mapped IPv6 address. If *dualstack_ipv6* is false it will explicitly disable this functionality on platforms that enable it by default (e.g. Linux). This parameter can be used in conjunction with `has_dualstack_ipv6()`:

```
import socket

addr = ("", 8080)  # all interfaces, port 8080
if socket.has_dualstack_ipv6():
    s = socket.create_server(addr, family=socket.AF_INET6, dualstack_ipv6=True)
else:
    s = socket.create_server(addr)
```

> **참고**
>
> POSIX 플랫폼에서 `SO_REUSEADDR` 소켓 옵션은 같은 *address*에 바인드 되었고 TIME_WAIT 상태로 남아 있던 이전 소켓을 즉시 재사용하기 위해 설정됩니다.

Added in version 3.8.

socket.**has_dualstack_ipv6**()

플랫폼이 IPv4와 IPv6 연결을 모두 처리할 수 있는 TCP 소켓을 만드는 것을 지원하면 `True`를 반환합니다.

Added in version 3.8.

socket.**fromfd**(*fd, family, type, proto=0*)

Duplicate the file descriptor *fd* (an integer as returned by a file object' s `fileno()` method) and build a socket object from the result. Address family, socket type and protocol number are as for the `socket()` function above. The file descriptor should refer to a socket, but this is not checked — subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode.

새로 만들어진 소켓은 상속 불가능합니다.

버전 3.4에서 변경: 반환된 소켓은 이제 상속 불가능합니다.

socket.**fromshare**(*data*)

`socket.share()` 메서드에서 얻은 데이터로 소켓의 인스턴스를 만듭니다. 소켓은 블로킹 모드로 간주합니다.

가용성: Windows.

Added in version 3.3.

socket.**SocketType**

이것은 소켓 객체 형을 나타내는 파이썬 형 객체입니다. `type(socket(...))`과 같습니다.

기타 함수

socket 모듈은 또한 다양한 네트워크 관련 서비스를 제공합니다:

socket.**close**(*fd*)

소켓 파일 기술자를 닫습니다. 이것은 *os.close()*와 비슷하지만, 소켓 용입니다. 일부 플랫폼(가장 눈에 띄는 것은 윈도우)에서는 *os.close()*가 소켓 파일 기술자에 대해 작동하지 않습니다.

Added in version 3.7.

socket.**getaddrinfo**(*host, port, family=AF_UNSPEC, type=0, proto=0, flags=0*)

This function wraps the C function `getaddrinfo` of the underlying system.

host/*port* 인자를 해당 서비스에 연결된 소켓을 만드는 데 필요한 모든 인자가 들어있는 5-튜플의 시퀀스로 변환합니다. *host*는 도메인 이름, IPv4/v6 주소의 문자열 표현 또는 `None` 입니다. *port*는 `'http'`와 같은 문자열 서비스 이름, 숫자 포트 번호 또는 `None` 입니다. `None`을 *host* 와 *port*의 값으로 전달해서, `NULL`을 하부 C API에 전달할 수 있습니다.

The *family*, *type* and *proto* arguments can be optionally specified in order to provide options and limit the list of addresses returned. Pass their default values (*AF_UNSPEC*, 0, and 0, respectively) to not limit the results. See the note below for details.

The *flags* argument can be one or several of the `AI_*` constants, and will influence how results are computed and returned. For example, `AI_NUMERICHOST` will disable domain name resolution and will raise an error if *host* is a domain name.

이 함수는 다음과 같은 구조의 5-튜플의 리스트를 반환합니다:

```
(family, type, proto, canonname, sockaddr)
```

In these tuples, *family*, *type*, *proto* are all integers and are meant to be passed to the *socket()* function. *canonname* will be a string representing the canonical name of the *host* if `AI_CANONNAME` is part of the *flags* argument; else *canonname* will be empty. *sockaddr* is a tuple describing a socket address, whose format depends on the returned *family* (a `(address, port)` 2-tuple for *AF_INET*, a `(address, port, flowinfo, scope_id)` 4-tuple for *AF_INET6*), and is meant to be passed to the *socket.connect()* method.

> **참고**
>
> If you intend to use results from `getaddrinfo()` to create a socket (rather than, for example, retrieve *canonname*), consider limiting the results by *type* (e.g. *SOCK_STREAM* or *SOCK_DGRAM*) and/or *proto* (e.g. `IPPROTO_TCP` or `IPPROTO_UDP`) that your application can handle.
>
> The behavior with default values of *family*, *type*, *proto* and *flags* is system-specific.
>
> Many systems (for example, most Linux configurations) will return a sorted list of all matching addresses. These addresses should generally be tried in order until a connection succeeds (possibly tried in parallel, for example, using a Happy Eyeballs algorithm). In these cases, limiting the *type* and/or *proto* can help eliminate unsuccessful or unusable connection attempts.
>
> Some systems will, however, only return a single address. (For example, this was reported on Solaris and AIX configurations.) On these systems, limiting the *type* and/or *proto* helps ensure that this address is usable.

`host`, `port`, `family`, `type`, `protocol`을 인자로 감사 이벤트(*auditing event*) `socket.getaddrinfo`를 발생시킵니다.

다음 예제는 `example.org`의 포트 80으로 가는 가상의 TCP 연결에 대한 주소 정보를 가져옵니다 (IPv6가 활성화되지 않았으면 여러분의 시스템에서는 결과가 다를 수 있습니다):

```
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(socket.AF_INET6, socket.SOCK_STREAM,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
(socket.AF_INET, socket.SOCK_STREAM,
6, '', ('93.184.216.34', 80))]
```

버전 3.2에서 변경: 매개 변수는 이제 키워드 인자를 사용하여 전달할 수 있습니다.

버전 3.7에서 변경: IPv6 멀티캐스트 주소의 경우, 주소를 나타내는 문자열에는 `%scope_id` 부분이 포함되지 않습니다.

`socket.`**`getfqdn`**([*name*])

Return a fully qualified domain name for *name*. If *name* is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by `gethostbyaddr()` is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available and *name* was provided, it is returned unchanged. If *name* was empty or equal to `'0.0.0.0'`, the hostname from `gethostname()` is returned.

`socket.`**`gethostbyname`**(*hostname*)

호스트 이름을 IPv4 주소 형식으로 변환합니다. IPv4 주소는 `'100.50.200.5'`와 같은 문자열로 반환됩니다. 호스트 이름이 IPv4 주소면 변경되지 않고 반환됩니다. 더욱 완전한 인터페이스는 `gethostbyname_ex()`를 참조하십시오. `gethostbyname()`는 IPv6 이름 결정을 지원하지 않으며, IPv4/v6 이중 스택 지원을 위해서는 대신 `getaddrinfo()`를 사용해야 합니다.

`hostname`을 인자로 감사 이벤트(*auditing event*) `socket.gethostbyname`을 발생시킵니다.

가용성: not WASI.

`socket.`**`gethostbyname_ex`**(*hostname*)

Translate a host name to IPv4 address format, extended interface. Return a 3-tuple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the host' s primary host name, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). `gethostbyname_ex()` does not support IPv6 name resolution, and `getaddrinfo()` should be used instead for IPv4/v6 dual stack support.

`hostname`을 인자로 감사 이벤트(*auditing event*) `socket.gethostbyname`을 발생시킵니다.

가용성: not WASI.

`socket.`**`gethostname`**()

파이썬 인터프리터가 현재 실행 중인 기계의 호스트 명을 포함한 문자열을 반환합니다.

인자 없이 감사 이벤트(*auditing event*) `socket.gethostname`을 발생시킵니다.

참고: `gethostname()`은 항상 완전히 정규화된 도메인 이름을 반환하지는 않습니다; 원한다면 `getfqdn()`을 사용하십시오.

가용성: not WASI.

`socket.`**`gethostbyaddr`**(*ip_address*)

Return a 3-tuple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the primary host name responding to the given *ip_address*, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4/v6 addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function `getfqdn()`. `gethostbyaddr()` supports both IPv4 and IPv6.

`ip_address`를 인자로 감사 이벤트(*auditing event*) `socket.gethostbyaddr`을 발생시킵니다.

가용성: not WASI.

`socket.`**`getnameinfo`**(*sockaddr*, *flags*)

Translate a socket address *sockaddr* into a 2-tuple `(host, port)`. Depending on the settings of *flags*, the result can contain a fully qualified domain name or numeric address representation in *host*. Similarly, *port* can contain a string port name or a numeric port number.

IPv6 주소의 경우, *sockaddr*에 의미 있는 *scope_id*가 있으면 `%scope_id`를 host 부분에 덧붙입니다. 보통 이것은 멀티캐스트 주소에서 일어납니다.

*flags*에 대한 자세한 내용은 `getnameinfo(3)`을 참조하십시오.

`sockaddr`을 인자로 감사 이벤트(*auditing event*) `socket.getnameinfo`를 발생시킵니다.

가용성: not WASI.

`socket.`**`getprotobyname`**(*protocolname*)

Translate an internet protocol name (for example, `'icmp'`) to a constant suitable for passing as the (optional) third argument to the `socket()` function. This is usually only needed for sockets opened in "raw" mode (`SOCK_RAW`); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

가용성: not WASI.

`socket.`**`getservbyname`**(*servicename*[, *protocolname*])

Translate an internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

`servicename`, `protocolname`을 인자로 감사 이벤트(*auditing event*) `socket.getservbyname`을 발생시킵니다.

가용성: not WASI.

`socket.`**`getservbyport`**(*port*[, *protocolname*])

Translate an internet port number and protocol name to a service name for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

`port`, `protocolname`을 인자로 감사 이벤트(*auditing event*) `socket.getservbyport`를 발생시킵니다.

가용성: not WASI.

`socket.`**`ntohl`**(*x*)

32비트 양의 정수를 네트워크 바이트 순서에서 호스트 바이트 순서로 변환합니다. 호스트 바이트 순서가 네트워크 바이트 순서와 같은 시스템에서, 이것은 아무 일도 하지 않습니다; 그렇지 않으면, 4바이트 스와프 연산을 수행합니다.

`socket.`**`ntohs`**(*x*)

16비트 양의 정수를 네트워크 바이트 순서에서 호스트 바이트 순서로 변환합니다. 호스트 바이트 순서가 네트워크 바이트 순서와 같은 시스템에서, 이것은 아무 일도 하지 않습니다; 그렇지 않으면, 2바이트 스와프 연산을 수행합니다.

버전 3.10에서 변경: Raises `OverflowError` if *x* does not fit in a 16-bit unsigned integer.

`socket.`**`htonl`**(*x*)

32비트 양의 정수를 호스트 바이트 순서에서 네트워크 바이트 순서로 변환합니다. 호스트 바이트 순서가 네트워크 바이트 순서와 같은 시스템에서, 이것은 아무 일도 하지 않습니다; 그렇지 않으면, 4바이트 스와프 연산을 수행합니다.

`socket.`**`htons`**(*x*)

16비트 양의 정수를 호스트 바이트 순서에서 네트워크 바이트 순서로 변환합니다. 호스트 바이트 순서가 네트워크 바이트 순서와 같은 시스템에서, 이것은 아무 일도 하지 않습니다; 그렇지 않으면, 2바이트 스와프 연산을 수행합니다.

버전 3.10에서 변경: Raises `OverflowError` if *x* does not fit in a 16-bit unsigned integer.

`socket.`**`inet_aton`**(*ip_string*)

Convert an IPv4 address from dotted-quad string format (for example, '123.45.67.89') to 32-bit packed binary format, as a bytes object four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type `in_addr`, which is the C type for the 32-bit packed binary this function returns.

`inet_aton()`는 3점 미만의 문자열도 허용합니다; 자세한 내용은 유닉스 매뉴얼 페이지 `inet(3)`을 참조하십시오.

이 함수에 전달된 IPv4 주소 문자열이 유효하지 않으면, `OSError`가 발생합니다. 정확히 무엇이 유효한지는 `inet_aton()`의 하부 C 구현에 따라 달라짐에 유의하십시오.

*inet_aton()*은 IPv6를 지원하지 않으며, IPv4/v6 이중 스택 지원을 위해서는 대신 *inet_pton()*를 사용해야 합니다.

socket.**inet_ntoa**(*packed_ip*)

Convert a 32-bit packed IPv4 address (a *bytes-like object* four bytes in length) to its standard dotted-quad string representation (for example, '123.45.67.89'). This is useful when conversing with a program that uses the standard C library and needs objects of type `in_addr`, which is the C type for the 32-bit packed binary data this function takes as an argument.

이 함수에 전달된 바이트 시퀀스가 정확히 4바이트 길이가 아니면, *OSError*가 발생합니다. *inet_ntoa()*는 IPv6를 지원하지 않으며, IPv4/v6 이중 스택 지원을 위해서는 대신 *inet_ntop()*를 사용해야 합니다.

버전 3.5에서 변경: 이제 쓰기 가능한 바이트열류 객체를 받아들입니다.

socket.**inet_pton**(*address_family*, *ip_string*)

Convert an IP address from its family-specific string format to a packed, binary format. *inet_pton()* is useful when a library or network protocol calls for an object of type `in_addr` (similar to *inet_aton()*) or `in6_addr`.

*address_family*에 대해 지원되는 값은 현재 *AF_INET*과 *AF_INET6*입니다. IP 주소 문자열 *ip_string*가 유효하지 않으면, *OSError*가 발생합니다. 정확히 무엇이 유효한지는 *address_family*의 값과 `inet_pton()`의 하부 구현에 따라 달라집니다.

가용성: Unix, Windows.

버전 3.4에서 변경: 윈도우 지원이 추가되었습니다

socket.**inet_ntop**(*address_family*, *packed_ip*)

Convert a packed IP address (a *bytes-like object* of some number of bytes) to its standard, family-specific string representation (for example, `'7.10.0.5'` or `'5aef:2b::8'`). *inet_ntop()* is useful when a library or network protocol returns an object of type `in_addr` (similar to *inet_ntoa()*) or `in6_addr`.

*address_family*에 대해 지원되는 값은 현재 *AF_INET*과 *AF_INET6*입니다. 바이트열 객체 *packed_ip*가 지정된 주소 패밀리의 올바른 길이가 아니면, *ValueError*가 발생합니다. *inet_ntop()* 호출로 인한 에러에는 *OSError*가 발생합니다.

가용성: Unix, Windows.

버전 3.4에서 변경: 윈도우 지원이 추가되었습니다

버전 3.5에서 변경: 이제 쓰기 가능한 바이트열류 객체를 받아들입니다.

socket.**CMSG_LEN**(*length*)

주어진 *length*의 연관된 데이터가 있는 보조(ancillary) 데이터 항목의 (후행 패딩을 제외한) 총 길이를 반환합니다. 이 값은 *recvmsg()*가 보조 데이터의 단일 항목을 수신하기 위한 버퍼 크기로 종종 사용될 수 있지만, **RFC 3542**는 이식성 있는 응용 프로그램에서 *CMSG_SPACE()*를 사용하도록 요구하는데, 항목이 버퍼의 마지막 부분일 때도 패딩을 위한 공간을 포함합니다. *length*가 허용되는 값 범위를 벗어나면 *OverflowError*를 발생시킵니다.

가용성: Unix, not WASI.

Most Unix platforms.

Added in version 3.3.

socket.**CMSG_SPACE**(*length*)

주어진 *length*의 연관된 데이터가 있는 보조(ancillary) 데이터 항목을 수신하기 위해 *recvmsg()*에 필요한 버퍼 크기를 반환하는데, 후행 패딩을 포함합니다. 여러 항목을 수신하는 데 필요한 버퍼 공간은 연관된 데이터 길이에 대한 *CMSG_SPACE()* 값의 합입니다. *length*가 허용되는 값 범위를 벗어나면 *OverflowError*를 발생시킵니다.

일부 시스템에서는 이 함수를 제공하지 않으면서 보조(ancillary) 데이터를 지원할 수 있음에 유의하십시오. 또한, 이 함수의 결과를 사용하여 버퍼 크기를 설정하면 수신할 수 있는 보조 데이터의 양이 정확하게 제한되지 않을 수 있음에도 유의하십시오. 추가 데이터가 패딩 영역에 들어갈 수 있기 때문입니다.

가용성: Unix, not WASI.

most Unix platforms.

Added in version 3.3.

socket.**getdefaulttimeout**()

새로운 소켓 객체의 기본 시간제한을 초 단위로 (float) 반환합니다. None 값은 새 소켓 객체가 시간제한이 없음을 나타냅니다. 소켓 모듈을 처음 임포트 할 때 기본값은 None 입니다.

socket.**setdefaulttimeout**(*timeout*)

새 소켓 객체의 기본 시간제한을 초 단위로 (float) 설정합니다. 소켓 모듈을 처음 임포트 할 때 기본값은 None 입니다. 가능한 값과 해당 의미는 *settimeout()*을 참조하십시오.

socket.**sethostname**(*name*)

기계의 호스트 명을 *name*으로 설정합니다. 충분한 권한이 없으면 *OSError*가 발생합니다.

name을 인자로 감사 이벤트(*auditing event*) socket.sethostname을 발생시킵니다.

가용성: Unix, not Android.

Added in version 3.3.

socket.**if_nameindex**()

네트워크 인터페이스 정보 (인덱스 정수, 이름 문자열) 튜플의 리스트를 반환합니다. 시스템 호출이 실패하면 *OSError*.

가용성: Unix, Windows, not WASI.

Added in version 3.3.

버전 3.8에서 변경: 윈도우 지원이 추가되었습니다.

참고

윈도우에서 네트워크 인터페이스는 다른 문맥에서 다른 이름을 갖습니다 (모든 이름은 예입니다):

- UUID: {FB605B73-AAC2-49A6-9A2F-25416AEA0573}
- 이름: ethernet_32770
- 친숙한 이름: vEthernet (nat)
- 설명: Hyper-V Virtual Ethernet Adapter

이 함수는 목록에서 두 번째 형식의 이름을 반환합니다, 이 예의 경우 ethernet_32770.

socket.**if_nametoindex**(*if_name*)

인터페이스 이름에 대응하는 네트워크 인터페이스 인덱스 번호를 반환합니다. 주어진 이름을 가진 인터페이스가 없으면 *OSError*.

가용성: Unix, Windows, not WASI.

Added in version 3.3.

버전 3.8에서 변경: 윈도우 지원이 추가되었습니다.

더 보기

"인터페이스 이름"은 *if_nameindex()*에 설명된 이름입니다.

socket.**if_indextoname**(*if_index*)

인터페이스 인덱스 번호에 해당하는 네트워크 인터페이스 이름을 반환합니다. 지정된 인덱스의 인터페이스가 없으면 *OSError*.

가용성: Unix, Windows, not WASI.

Added in version 3.3.

버전 3.8에서 변경: 윈도우 지원이 추가되었습니다.

> **더 보기**
>
> "인터페이스 이름"은 `if_nameindex()`에 설명된 이름입니다.

`socket.send_fds`(*sock*, *buffers*, *fds*[, *flags*[, *address*]])

Send the list of file descriptors *fds* over an `AF_UNIX` socket *sock*. The *fds* parameter is a sequence of file descriptors. Consult `sendmsg()` for the documentation of these parameters.

가용성: Unix, not WASI.

Unix platforms supporting `sendmsg()` and `SCM_RIGHTS` mechanism.

Added in version 3.9.

`socket.recv_fds`(*sock*, *bufsize*, *maxfds*[, *flags*])

Receive up to *maxfds* file descriptors from an `AF_UNIX` socket *sock*. Return `(msg, list(fds), flags, addr)`. Consult `recvmsg()` for the documentation of these parameters.

가용성: Unix, not WASI.

Unix platforms supporting `recvmsg()` and `SCM_RIGHTS` mechanism.

Added in version 3.9.

> **참고**
>
> 파일 기술자 리스트 끝에 있는 모든 잘린 정수.

19.2.3 소켓 객체

소켓 객체에는 다음과 같은 메서드가 있습니다. `makefile()`를 제외하고, 이것들은 소켓에 적용할 수 있는 유닉스 시스템 호출에 해당합니다.

버전 3.2에서 변경: 컨텍스트 관리자 프로토콜 지원이 추가되었습니다. 컨텍스트 관리자를 빠져나가는 것은 `close()`를 호출하는 것과 동등합니다.

`socket.accept`()

연결을 받아들입니다. 소켓은 주소에 바인드되어 연결을 리스닝하고 있어야 합니다. 반환 값은 `(conn, address)` 쌍입니다. 여기서 *conn*는 연결에서 데이터를 보내고 받을 수 있는 새로운 소켓 객체이고, *address*는 연결의 다른 끝에 있는 소켓에 바인드 된 주소입니다.

새로 만들어진 소켓은 상속 불가능합니다.

버전 3.4에서 변경: 소켓은 이제 상속 불가능합니다.

버전 3.5에서 변경: 시스템 호출이 인터럽트 되고 시그널 처리기가 예외를 발생시키지 않으면, 메서드는 이제 `InterruptedError` 예외를 발생시키는 대신 시스템 호출을 재시도합니다 (이유는 **PEP 475**를 참조하십시오).

`socket.bind`(*address*)

소켓을 *address*에 바인드 합니다. 소켓은 이미 바인드 되어 있으면 안 됩니다. (*address*의 형식은 주소 패밀리에 따라 다릅니다 — 위를 보십시오.)

`self, address`을 인자로 감사 이벤트(*auditing event*) `socket.bind`를 발생시킵니다.

가용성: not WASI.

socket.**close**()

Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from *makefile()* are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

소켓은 가비지 수집될 때 자동으로 닫히지만, 명시적으로 *close()* 하거나 `with` 문을 사용하는 것이 좋습니다.

버전 3.6에서 변경: 하부 `close()` 호출이 수행될 때 에러가 발생하면 이제 *OSError*가 발생합니다.

> **참고**
>
> *close()* releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call *shutdown()* before *close()*.

socket.**connect**(*address*)

*address*에 있는 원격 소켓에 연결합니다. (*address*의 형식은 주소 패밀리에 따라 다릅니다 — 위를 보십시오.)

If the connection is interrupted by a signal, the method waits until the connection completes, or raise a *TimeoutError* on timeout, if the signal handler doesn' t raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises an *InterruptedError* exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

`self`, `address`를 인자로 감사 이벤트 (*auditing event*) `socket.connect`를 발생시킵니다.

버전 3.5에서 변경: 연결이 시그널에 의해 인터럽트 되고, 시그널 처리기가 예외를 발생시키지 않고, 소켓이 블로킹하거나 시간제한을 가지면, 이 메서드는 이제 *InterruptedError* 예외를 발생시키는 대신 연결이 완료될 때까지 대기합니다 (이유는 **PEP 475**를 참조하십시오).

가용성: not WASI.

socket.**connect_ex**(*address*)

`connect(address)`와 비슷하지만, C 수준의 `connect()` 호출로 반환된 에러에 대한 예외를 발생시키는 대신 에러 표시기를 반환합니다 ("호스트를 찾을 수 없음"과 같은 다른 문제는 여전히 예외를 발생시킬 수 있습니다). 연산이 성공하면 에러 표시기는 `0`이고, 그렇지 않으면 `errno` 변수의 값입니다. 예를 들어 비동기 연결을 지원하는 데 유용합니다.

`self`, `address`를 인자로 감사 이벤트 (*auditing event*) `socket.connect`를 발생시킵니다.

가용성: not WASI.

socket.**detach**()

하부 파일 기술자를 실제로 닫지 않으면서 소켓 객체를 닫힌 상태로 만듭니다. 파일 기술자가 반환되고, 다른 용도로 재사용 될 수 있습니다.

Added in version 3.2.

socket.**dup**()

소켓을 복제합니다.

새로 만들어진 소켓은 상속 불가능합니다.

버전 3.4에서 변경: 소켓은 이제 상속 불가능합니다.

가용성: not WASI.

socket.**fileno**()

소켓의 파일 기술자(작은 정수)를 반환하거나, 실패하면 -1을 반환합니다. 이것은 *select.select()*에서 유용합니다.

윈도우에서, 이 메서드가 돌려주는 작은 정수는 파일 기술자를 사용할 수 있는 곳(가령 *os.fdopen()*)에 사용할 수 없습니다. 유닉스에는 이러한 제한이 없습니다.

socket.**get_inheritable**()

소켓의 파일 기술자나 소켓 핸들의 상속 가능 플래그를 가져옵니다: 소켓이 자식 프로세스에서 상속될 수 있으면 True, 그렇지 않으면 False.

Added in version 3.4.

socket.**getpeername**()

소켓이 연결된 원격 주소를 반환합니다. 이것은 예를 들어, 원격 IPv4/v6 소켓의 포트 번호를 찾는 데 유용합니다. (반환되는 주소의 형식은 주소 패밀리에 따라 다릅니다 — 위를 보십시오.) 일부 시스템에서는 이 함수가 지원되지 않습니다.

socket.**getsockname**()

소켓 자신의 주소를 반환합니다. 이것은 예를 들어 IPv4/v6 소켓의 포트 번호를 찾는 데 유용합니다. (반환되는 주소의 형식은 주소 패밀리에 따라 다릅니다 — 위를 보십시오.)

socket.**getsockopt**(*level*, *optname*[, *buflen*])

Return the value of the given socket option (see the Unix man page *getsockopt(2)*). The needed symbolic constants (*SO_* etc.*) are defined in this module. If *buflen* is absent, an integer option is assumed and its integer value is returned by the function. If *buflen* is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a bytes object. It is up to the caller to decode the contents of the buffer (see the optional built-in module *struct* for a way to decode C structures encoded as byte strings).

가용성: not WASI.

socket.**getblocking**()

소켓이 블로킹 모드면 True를 반환하고, 비 블로킹이면 False를 반환합니다.

This is equivalent to checking socket.gettimeout() != 0.

Added in version 3.7.

socket.**gettimeout**()

소켓 연산에 관련한 시간제한을 초(float)로 돌려줍니다. 시간제한이 설정되어 있지 않으면 None를 돌려줍니다. 이것은 *setblocking()* 이나 *settimeout()*에 대한 마지막 호출을 반영합니다.

socket.**ioctl**(*control*, *option*)

ioctl() 메서드는 WSAIoctl 시스템 인터페이스에 대한 제한된 인터페이스입니다. 자세한 내용은 Win32 설명서를 참조하십시오.

다른 플랫폼에서는, 범용 *fcntl.fcntl()* 과 *fcntl.ioctl()* 함수를 사용할 수 있습니다; 첫 번째 인자로 소켓 객체를 받아들입니다.

현재 다음 제어 코드만 지원됩니다: SIO_RCVALL, SIO_KEEPALIVE_VALS 및 SIO_LOOPBACK_FAST_PATH.

가용성: Windows

버전 3.6에서 변경: SIO_LOOPBACK_FAST_PATH가 추가되었습니다.

socket.**listen**([*backlog*])

서버가 연결을 수락하도록 합니다. *backlog*가 지정되면, 0 이상이어야 합니다 (더 낮으면 0으로 설정됩니다); 새로운 연결을 거부하기 전에 시스템이 허락할 수락되지 않은 연결 수를 지정합니다. 지정하지 않으면, 기본값으로 적당한 값이 선택됩니다.

가용성: not WASI.

버전 3.5에서 변경: 이제 *backlog* 매개 변수가 선택적입니다.

socket.**makefile**(*mode='r'*, *buffering=None*, **, encoding=None*, *errors=None*, *newline=None*)

Return a *file object* associated with the socket. The exact returned type depends on the arguments given to *makefile()*. These arguments are interpreted the same way as by the built-in *open()* function, except the only supported *mode* values are 'r' (default), 'w', 'b', or a combination of those.

소켓은 블로킹 모드 여야 합니다; 시간제한을 가질 수 있지만, 시간 초과가 발생하면 파일 객체의 내부 버퍼가 일관성없는 상태로 끝날 수 있습니다.

makefile() 에 의해 반환된 파일 객체를 닫는 것은, 다른 모든 파일 객체가 닫혔고 소켓 객체에서 socket.close() 가 호출되었지 않은 한 원래 소켓을 닫지는 않습니다.

> **참고**
>
> 윈도우에서, makefile() 로 만든 파일류 객체는 파일 기술자가 있는 파일 객체가 필요한 곳에서는 사용할 수 없습니다, 가령 subprocess.Popen() 의 stream 인자.

socket.**recv**(*bufsize*[, *flags*])

Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by *bufsize*. A returned empty bytes object indicates that the client has disconnected. See the Unix manual page recv(2) for the meaning of the optional argument *flags*; it defaults to zero.

버전 3.5에서 변경: 시스템 호출이 인터럽트 되고 시그널 처리기가 예외를 발생시키지 않으면, 메서드는 이제 InterruptedError 예외를 발생시키는 대신 시스템 호출을 재시도합니다 (이유는 **PEP 475**를 참조하십시오).

socket.**recvfrom**(*bufsize*[, *flags*])

소켓에서 데이터를 수신합니다. 반환 값은 (bytes, address) 쌍입니다. 여기서 *bytes*는 수신한 데이터를 나타내는 바이트열 객체이고, *address*는 데이터를 보내는 소켓의 주소입니다. 선택적 인자 *flags*의 의미는 유닉스 매뉴얼 페이지 recv(2)를 보십시오; 기본값은 0입니다. (*address*의 형식은 주소 패밀리에 따라 다릅니다 — 위를 보십시오.)

버전 3.5에서 변경: 시스템 호출이 인터럽트 되고 시그널 처리기가 예외를 발생시키지 않으면, 메서드는 이제 InterruptedError 예외를 발생시키는 대신 시스템 호출을 재시도합니다 (이유는 **PEP 475**를 참조하십시오).

버전 3.7에서 변경: 멀티캐스트 IPv6 주소의 경우, *address*의 첫 번째 항목에는 %scope_id 부분이 더는 포함되지 않습니다. 전체 IPv6 주소를 얻으려면 getnameinfo() 를 사용하십시오.

socket.**recvmsg**(*bufsize*[, *ancbufsize*[, *flags*]])

일반 데이터(최대 *bufsize* 바이트)와 보조(ancillary) 데이터를 소켓에서 수신합니다. *ancbufsize* 인자는 보조 데이터 수신에 사용되는 내부 버퍼의 크기를 바이트 단위로 설정합니다; 기본값은 0이며 보조 데이터가 수신되지 않는다는 뜻입니다. 보조 데이터를 위한 적절한 버퍼 크기는 CMSG_SPACE() 나 CMSG_LEN() 를 사용하여 계산할 수 있으며, 버퍼에 들어가지 않는 항목은 잘리거나 삭제될 수 있습니다. *flags* 인자의 기본값은 0이고 recv() 와 같은 의미입니다.

반환 값은 4-튜플입니다: (data, ancdata, msg_flags, address). *data* 항목은 일반 데이터를 담은 bytes 객체입니다. *ancdata* 항목은 수신된 보조 데이터(제어 메시지)를 나타내는 0개 이상의 튜플 (cmsg_level, cmsg_type, cmsg_data) 의 리스트입니다: *cmsg_level* 와 *cmsg_type*는 각각 프로토콜 수준과 프로토콜 특정 형을 지정하는 정수이고, *cmsg_data*는 연결된 데이터를 담은 bytes 객체입니다. *msg_flags* 항목은 수신된 메시지의 조건을 나타내는 다양한 플래그의 비트별 OR 입니다; 자세한 내용은 시스템 설명서를 참조하십시오. 수신 소켓이 연결되어있지 않으면, *address*는 송신 소켓의 주소입니다, (사용 가능하다면); 그렇지 않으면 값은 지정되지 않습니다.

On some systems, sendmsg() and recvmsg() can be used to pass file descriptors between processes over an AF_UNIX socket. When this facility is used (it is often restricted to SOCK_STREAM sockets), recvmsg() will return, in its ancillary data, items of the form (socket.SOL_SOCKET, socket.SCM_RIGHTS, fds), where *fds* is a bytes object representing the new file descriptors as a binary array of the native C int type. If recvmsg() raises an exception after the system call returns, it will first attempt to close any file descriptors received via this mechanism.

일부 시스템은 부분적으로만 수신된 보조 데이터 항목의 절단 길이를 나타내지 않습니다. 항목이 버퍼의 끝을 넘어 확장된 것처럼 보이면, recvmsg() 는 RuntimeWarning 를 발생시키고, 관련 데이터의 시작 전에 절단되지 않은 버퍼 내에 있는 부분을 반환합니다.

SCM_RIGHTS 메커니즘을 지원하는 시스템에서, 다음 함수는 최대 *maxfds* 파일 기술자를 수신하여, 메시지 데이터와 기술자를 담은 리스트를 반환합니다 (관련 없는 수신되는 제어 메시지와 같은 예기치 않은 조건은 무시하면서). sendmsg() 를 참조하십시오.

```
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")   # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG_LEN(maxfds *␣
↪fds.itemsize))
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS:
            # Append data, ignoring any truncated integers at the end.
            fds.frombytes(cmsg_data[:len(cmsg_data) - (len(cmsg_data) % fds.
↪itemsize)])
    return msg, list(fds)
```

가용성: Unix.

Most Unix platforms.

Added in version 3.3.

버전 3.5에서 변경: 시스템 호출이 인터럽트 되고 시그널 처리기가 예외를 발생시키지 않으면, 메서드는 이제 `InterruptedError` 예외를 발생시키는 대신 시스템 호출을 재시도합니다 (이유는 **PEP 475**를 참조하십시오).

socket.**recvmsg_into**(*buffers*[, *ancbufsize*[, *flags*]])

`recvmsg()` 처럼 동작해서, 일반 데이터와 보조 데이터를 소켓에서 수신하지만, 새로운 바이트열 객체를 반환하는 대신 일반 데이터를 일련의 버퍼로 분산시킵니다. *buffers* 인자는 쓰기 가능한 버퍼(예를 들어, `bytearray` 객체)를 내보내는 객체의 이터러블이어야 합니다; 이것들은 모두 기록되었거나 버퍼가 더는 없을 때까지 일반 데이터의 연속적인 덩어리로 채워질 것입니다. 운영 체제는 사용할 수 있는 버퍼 수에 제한(`sysconf()` 값 `SC_IOV_MAX`)을 설정할 수 있습니다. *ancbufsize* 와 *flags* 인자는 `recvmsg()` 와 같은 의미가 있습니다.

반환 값은 4-튜플입니다: `(nbytes, ancdata, msg_flags, address)`. 여기서 *nbytes*는 버퍼에 기록된 일반 데이터의 총 바이트 수이며, *ancdata*, *msg_flags* 및 *address*는 `recvmsg()` 와 같습니다.

예제:

```
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]
```

가용성: Unix.

Most Unix platforms.

Added in version 3.3.

socket.**recvfrom_into**(*buffer*[, *nbytes*[, *flags*]])

소켓에서 데이터를 수신하는데, 새로운 바이트열을 만드는 대신 *buffer*에 씁니다. 반환 값은 쌍 `(nbytes, address)` 입니다. 여기서 *nbytes*는 수신 된 바이트 수이고, *address*는 데이터를 보내는 소켓의 주소입니다. 선택적 인자 *flags*의 의미에 대해서는 유닉스 매뉴얼 페이지 `recv(2)` 를 보십시오; 기본값은 0입니다. (*address*의 형식은 주소 패밀리에 따라 다릅니다 — 위를 보십시오.)

socket.**recv_into**(*buffer*[, *nbytes*[, *flags*]])

소켓에서 최대 *nbytes* 바이트까지 수신하는데, 새 바이트열을 만드는 대신 데이터를 버퍼에 저장합

니다. *nbytes*가 지정되지 않으면 (또는 0), 지정된 버퍼에서 사용 가능한 크기까지 수신합니다. 수신한 바이트 수를 반환합니다. 선택적 인자 *flags*의 의미에 대해서는 유닉스 매뉴얼 페이지 `recv(2)`를 보십시오; 기본값은 0입니다.

socket.**send**(*bytes*[, *flags*])

소켓에 데이터를 보냅니다. 소켓은 원격 소켓에 연결되어야 합니다. 선택적 *flags* 인자는 위의 `recv()`와 같은 의미입니다. 전송된 바이트 수를 반환합니다. 응용 프로그램은 모든 데이터가 전송되었는지 확인해야 합니다; 일부 데이터만 전송되었으면, 응용 프로그램은 나머지 데이터의 전달을 시도해야 합니다. 이 주제에 대한 자세한 정보는, socket-howto를 참조하십시오.

버전 3.5에서 변경: 시스템 호출이 인터럽트 되고 시그널 처리기가 예외를 발생시키지 않으면, 메서드는 이제 `InterruptedError` 예외를 발생시키는 대신 시스템 호출을 재시도합니다 (이유는 **PEP 475**를 참조하십시오).

socket.**sendall**(*bytes*[, *flags*])

소켓에 데이터를 보냅니다. 소켓은 원격 소켓에 연결되어야 합니다. 선택적 *flags* 인자는 위의 `recv()`와 같은 의미입니다. `send()`와 달리, 이 메서드는 모든 데이터가 전송되거나 에러가 발생할 때까지 *bytes*의 데이터를 계속 전송합니다. 성공하면 `None`이 반환됩니다. 에러가 발생하면, 예외가 발생하는데, 성공적으로 전송된 데이터양을 (있기는 하다면) 확인하는 방법은 없습니다.

버전 3.5에서 변경: The socket timeout is no longer reset each time data is sent successfully. The socket timeout is now the maximum total duration to send all data.

버전 3.5에서 변경: 시스템 호출이 인터럽트 되고 시그널 처리기가 예외를 발생시키지 않으면, 메서드는 이제 `InterruptedError` 예외를 발생시키는 대신 시스템 호출을 재시도합니다 (이유는 **PEP 475**를 참조하십시오).

socket.**sendto**(*bytes*, *address*)

socket.**sendto**(*bytes*, *flags*, *address*)

소켓에 데이터를 보냅니다. 대상 소켓이 *address*로 지정되므로, 소켓은 원격 소켓에 연결되지 않아야 합니다. 선택적 *flags* 인자는 위의 `recv()`와 같은 의미가 있습니다. 전송된 바이트 수를 반환합니다. (*address*의 형식은 주소 패밀리에 따라 다릅니다 — 위를 보십시오.)

`self`, `address`를 인자로 감사 이벤트 (*auditing event*) `socket.sendto`를 발생시킵니다.

버전 3.5에서 변경: 시스템 호출이 인터럽트 되고 시그널 처리기가 예외를 발생시키지 않으면, 메서드는 이제 `InterruptedError` 예외를 발생시키는 대신 시스템 호출을 재시도합니다 (이유는 **PEP 475**를 참조하십시오).

socket.**sendmsg**(*buffers*[, *ancdata*[, *flags*[, *address*]]])

소켓에 일반과 보조 데이터를 보는데, 일련의 버퍼에서 일반 데이터를 모아서 단일 메시지로 연결합니다. *buffers* 인자는 일반 데이터를 바이트열류 객체의 이터러블로 지정합니다 (예를 들어, `bytes` 객체); 운영 체제는 사용할 수 있는 버퍼 수에 제한(`sysconf()` 값 `SC_IOV_MAX`)을 설정할 수 있습니다. *ancdata* 인자는 보조 데이터 (제어 메시지)를 0개 이상의 튜플 `(cmsg_level, cmsg_type, cmsg_data)`의 이터러블로 지정합니다. 여기서 *cmsg_level*와 *cmsg_type*는 각각 프로토콜 수준과 프로토콜 특정 형을 지정하는 정수이고, *cmsg_data*는 연결된 데이터를 담은 바이트열류 객체입니다. 일부 시스템(특히, `CMSG_SPACE()`가 없는 시스템)은 호출 당 하나의 제어 메시지를 송신하는 것만 지원할 수 있습니다. *flags* 인자의 기본값은 0이고 `send()`와 같은 의미입니다. *address*가 제공되고 `None`이 아니면, 메시지의 대상 주소를 설정합니다. 반환 값은 전송된 일반 데이터의 바이트 수입니다.

다음 함수는 `SCM_RIGHTS` 메커니즘을 지원하는 시스템에서, `AF_UNIX` 소켓을 통해 파일 기술자 리스트 *fds*를 보냅니다. `recvmsg()`도 참조하세요.

```
import socket, array

def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, socket.SCM_RIGHTS, array.
↪array("i", fds))])
```

가용성: Unix, not WASI.

Most Unix platforms.

self, address를 인자로 감사 이벤트(*auditing event*) socket.sendmsg를 발생시킵니다.

Added in version 3.3.

버전 3.5에서 변경: 시스템 호출이 인터럽트 되고 시그널 처리기가 예외를 발생시키지 않으면, 메서드는 이제 *InterruptedError* 예외를 발생시키는 대신 시스템 호출을 재시도합니다 (이유는 **PEP 475**를 참조하십시오).

socket.**sendmsg_afalg**([*msg*,]*, *op*[, *iv*[, *assoclen*[, *flags*]]])

AF_ALG 소켓용, *sendmsg()*의 특수한 버전. *AF_ALG* 소켓에 대한 모드, IV, AEAD 관련 데이터 길이 및 플래그를 설정합니다.

가용성: Linux >= 2.6.38.

Added in version 3.6.

socket.**sendfile**(*file*, *offset=0*, *count=None*)

고성능 *os.sendfile*을 사용하여 EOF에 도달할 때까지 파일을 보내고, 보낸 총 바이트 수를 반환합니다. *file*은 바이너리 모드로 열린 일반 파일 객체여야 합니다. *os.sendfile*을 사용할 수 없거나 (예를 들어, 윈도우) *file*가 일반 파일이 아니면, *send()*가 대신 사용됩니다. *offset*은 파일 읽기 시작할 위치를 알려줍니다. 지정되면, *count*는 EOF에 도달할 때까지 파일을 전송하는 대신 전송할 총 바이트 수입니다. 파일 위치는 반환하거나 에러가 발생했을 때 갱신됩니다. 이때 *file.tell()*을 사용하여 전송된 바이트 수를 계산할 수 있습니다. 소켓은 *SOCK_STREAM* 유형이어야 합니다. 비 블로킹 소켓은 지원되지 않습니다.

Added in version 3.5.

socket.**set_inheritable**(*inheritable*)

소켓의 파일 기술자나 소켓 핸들의 상속 가능 플래그를 설정합니다.

Added in version 3.4.

socket.**setblocking**(*flag*)

소켓의 블로킹이나 비 블로킹 모드를 설정합니다. *flag*가 거짓이면, 소켓은 비 블로킹으로 설정되고, 그렇지 않으면 블로킹 모드로 설정됩니다.

이 메서드는 특정 *settimeout()* 호출의 줄인 표현입니다:

- sock.setblocking(True)는 sock.settimeout(None)와 동등합니다
- sock.setblocking(False)는 sock.settimeout(0.0)와 동등합니다

버전 3.7에서 변경: 이 메서드는 더는 *socket.type*에 *SOCK_NONBLOCK* 플래그를 적용하지 않습니다.

socket.**settimeout**(*value*)

Set a timeout on blocking socket operations. The *value* argument can be a nonnegative floating-point number expressing seconds, or None. If a non-zero value is given, subsequent socket operations will raise a *timeout* exception if the timeout period *value* has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If None is given, the socket is put in blocking mode.

자세한 내용은, 소켓 시간제한에 대한 참고 사항을 보십시오.

버전 3.7에서 변경: 이 메서드는 더는 *socket.type*의 *SOCK_NONBLOCK* 플래그를 토글하지 않습니다.

socket.**setsockopt**(*level*, *optname*, *value:* int)

socket.**setsockopt**(*level*, *optname*, *value: buffer*)

socket.**setsockopt**(*level*, *optname*, *None*, *optlen: int*)

Set the value of the given socket option (see the Unix manual page *setsockopt(2)*). The needed symbolic constants are defined in this module (SO_* etc. <socket-unix-constants>). The value can be an integer, None or a *bytes-like object* representing a buffer. In the later case it is up to the caller to ensure that the bytestring contains the proper bits (see the optional built-in module *struct* for a way to encode C structures as bytestrings). When *value* is set to None, *optlen* argument is required. It' s equivalent to call setsockopt() C function with optval=NULL and optlen=optlen.

버전 3.5에서 변경: 이제 쓰기 가능한 바이트열류 객체를 받아들입니다.

버전 3.6에서 변경: setsockopt(level, optname, None, optlen: int) 형식이 추가되었습니다.

가용성: not WASI.

socket.**shutdown**(*how*)

연결의 한쪽 또는 양쪽 절반을 닫습니다. *how*가 `SHUT_RD`면, 추가 수신이 허용되지 않습니다. *how*가 `SHUT_WR`이면, 추가 전송이 허용되지 않습니다. *how*가 `SHUT_RDWR`이면, 추가 송수신이 허용되지 않습니다.

가용성: not WASI.

socket.**share**(*process_id*)

소켓을 복제하고 대상 프로세스와 공유할 수 있도록 준비합니다. 대상 프로세스는 *process_id*로 제공되어야 합니다. 결과 바이트열 객체는 어떤 프로세스 간 통신의 형태를 사용하여 대상 프로세스로 전달될 수 있으며 그곳에서 `fromshare()`를 사용하여 소켓을 다시 만들 수 있습니다. 일단, 이 메서드가 호출되면, 운영 체제가 이미 대상 프로세스를 위해 이를 복제 했으므로 소켓을 닫아도 안전합니다.

가용성: Windows.

Added in version 3.3.

메서드 `read()` 나 `write()` 가 없다는 점에 유의하십시오; 대신 `recv()` 와 `send()`를 *flags* 인자 없이 사용하십시오.

소켓 객체는 또한 `socket` 생성자에 지정된 값에 대응하는 다음과 같은 (읽기 전용) 어트리뷰트를 가집니다.

socket.**family**

소켓 패밀리.

socket.**type**

소켓 유형.

socket.**proto**

소켓 프로토콜.

19.2.4 소켓 시간제한에 대한 참고 사항

소켓 객체는 세 가지 모드 중 하나일 수 있습니다: 블로킹, 비 블로킹, 또는 시간제한. 소켓은 기본적으로 항상 블로킹 모드로 생성되지만, 이는 `setdefaulttimeout()`를 호출하여 변경할 수 있습니다.

- 블로킹 모드에서, 연산은 완료되거나 시스템에서 에러(가령 연결 시간 초과)를 반환할 때까지 블록합니다.
- In *non-blocking mode*, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately: functions from the `select` module can be used to know when and whether a socket is available for reading or writing.
- 시간제한 모드에서, 연산은 소켓에 대해 지정된 제한 시간 내에 완료할 수 없거나 (`timeout` 예외 발생), 시스템이 에러를 반환하면 실패합니다.

> **참고**
>
> At the operating system level, sockets in *timeout mode* are internally set in non-blocking mode. Also, the blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. This implementation detail can have visible consequences if e.g. you decide to use the `fileno()` of a socket.

시간제한과 connect 메서드

`connect()` 연산도 시간제한 설정의 영향을 받으며, 일반적으로 `connect()`를 호출하기 전에 `settimeout()`를 호출하거나 `create_connection()`에 timeout 매개 변수를 전달하는 것이 좋습니다. 그러나, 시스템 네트워크 스택은 파이썬 소켓 시간제한 설정과 관계없이 자체의 연결 시간제한 에러를 반환할 수 있습니다.

시간제한과 accept 메서드

*getdefaulttimeout()*가 *None*이 아니면, *accept()* 메서드에서 반환된 소켓은 그 시간제한을 상속합니다. 그렇지 않으면, 동작은 리스닝 소켓의 설정에 따라 다릅니다:

- 리스닝 소켓이 블로킹 모드 나 시간제한 모드에 있으면, *accept()*에 의해 반환된 소켓은 블로킹 모드에 있습니다.
- 리스닝 소켓이 비 블로킹 모드에 있으면, *accept()*에 의해 반환된 소켓이 블로킹 모드인지 비 블로킹 모드인지는 운영 체제에 따라 다릅니다. 플랫폼 간 동작을 보장하려면, 이 설정을 직접 재정의하는 것이 좋습니다.

19.2.5 예제

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence *socket()*, *bind()*, *listen()*, *accept()* (possibly repeating the *accept()* to service more than one client), while a client only needs the sequence *socket()*, *connect()*. Also note that the server does not *sendall()*/*recv()* on the socket it is listening on but on the new socket returned by *accept()*.

처음 두 예제는 IPv4만 지원합니다.

```
# Echo server program
import socket

HOST = ''                   # Symbolic name meaning all available interfaces
PORT = 50007                # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'      # The remote host
PORT = 50007                # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to all the addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

```
# Echo server program
import socket
import sys

HOST = None                 # Symbolic name meaning all available interfaces
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
PORT = 50007              # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.send(data)
```

```
# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
with s:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
print('Received', repr(data))
```

다음 예제는 윈도우에서 원시(raw) 소켓으로 매우 간단한 네트워크 스니퍼를 작성하는 방법을 보여줍니다. 이 예제는 인터페이스를 수정하기 위해 관리자 권한이 필요합니다:

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packets
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a packet
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

다음 예제는 원시(raw) 소켓 프로토콜을 사용하여, 소켓 인터페이스를 사용하여 CAN 네트워크와 통신하는 방법을 보여줍니다. 대신 브로드캐스트 관리자 프로토콜로 CAN을 사용하려면, 소켓을 이렇게 여십시오:

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

After binding (`CAN_RAW`) or connecting (`CAN_BCM`) the socket, you can use the `socket.send()` and `socket.recv()` operations (and their counterparts) on the socket object as usual.

이 마지막 예제는 특별한 권한이 필요할 수 있습니다:

```
import socket
import struct


# CAN frame packing/unpacking (see 'struct can_frame' in <linux/can.h>)

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])


# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_can_frame(cf))

    try:
        s.send(cf)
    except OSError:
        print('Error sending CAN frame')

    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')
```

실행 간격이 너무 짧게 여러 번 예제를 실행하면 이 에러가 발생할 수 있습니다:

```
OSError: [Errno 98] Address already in use
```

이것은 이전 실행이 소켓을 `TIME_WAIT` 상태로 남겨 두었고, 즉시 재사용할 수 없기 때문입니다.

There is a *socket* flag to set, in order to prevent this, `socket.SO_REUSEADDR`:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

`SO_REUSEADDR` 플래그는 자연스러운 시간제한이 만료되기를 기다리지 않고 `TIME_WAIT` 상태의 지역 소켓을 재사용하도록 커널에 알립니다.

더 보기

(C로 하는) 소켓 프로그래밍에 대한 소개는 다음 논문을 참조하십시오:

- *An Introductory 4.3BSD Interprocess Communication Tutorial*, Stuart Sechrest 저
- *An Advanced 4.3BSD Interprocess Communication Tutorial*, Samuel J. Leffler 외 저,

둘 다 유닉스 프로그래머 매뉴얼, 보충 문서 1 (섹션 PS1:7과 PS1:8)에 있습니다. 다양한 소켓 관련 시스템 호출에 대한 플랫폼별 레퍼런스 자료는 소켓 의미의 세부 정보에 대한 중요한 소스입니다. 유닉스에서는 매뉴얼 페이지를 참조하십시오; 윈도우에서는, WinSock (또는 Winsock 2) 명세를 참조하십시오. IPv6 지원 API의 경우, 독자는 Basic Socket Interface Extensions for IPv6라는 제목의 **RFC 3493**를 참조하고 싶을 겁니다.

19.3 `ssl` — TLS/SSL wrapper for socket objects

소스 코드: Lib/ssl.py

This module provides access to Transport Layer Security (often known as "Secure Sockets Layer") encryption and peer authentication facilities for network sockets, both client-side and server-side. This module uses the OpenSSL library. It is available on all modern Unix systems, Windows, macOS, and probably additional platforms, as long as OpenSSL is installed on that platform.

> **참고**
>
> Some behavior may be platform dependent, since calls are made to the operating system socket APIs. The installed version of OpenSSL may also cause variations in behavior. For example, TLSv1.3 comes with OpenSSL version 1.1.1.

> **경고**
>
> 보안 고려 사항을 읽지 않고 이 모듈을 사용하지 마십시오. 그렇게 하면 ssl 모듈의 기본 설정이 반드시 여러분의 응용 프로그램에 적합하지는 않으므로 잘못된 보안 인식으로 이어질 수 있습니다.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

이 절에서는 `ssl` 모듈의 객체와 함수를 설명합니다; TLS, SSL 및 인증서에 대한보다 일반적인 정보는, 하단의 "더 보기" 절에 있는 문서를 참조하십시오.

This module provides a class, *`ssl.SSLSocket`*, which is derived from the *`socket.socket`* type, and provides a socket-like wrapper that also encrypts and decrypts the data going over the socket with SSL. It supports additional methods such as `getpeercert()`, which retrieves the certificate of the other side of the connection, `cipher()`, which retrieves the cipher being used for the secure connection or `get_verified_chain()`, `get_unverified_chain()` which retrieves certificate chain.

더욱 정교한 응용 프로그램의 경우, *`ssl.SSLContext`* 클래스는 설정과 인증서를 관리하는 데 도움이 되며, *`SSLContext.wrap_socket()`* 메서드를 통해 만들어진 SSL 소켓이 상속할 수 있습니다.

버전 3.5.3에서 변경: OpenSSL 1.1.0과의 링크를 지원하도록 갱신되었습니다

버전 3.6에서 변경: OpenSSL 0.9.8, 1.0.0 및 1.0.1은 폐지되었으며 더는 지원되지 않습니다. 미래에는 ssl 모듈이 최소한 OpenSSL 1.0.2 나 1.1.0을 요구할 것입니다.

버전 3.10에서 변경: **PEP 644** has been implemented. The ssl module requires OpenSSL 1.1.1 or newer.

Use of deprecated constants and functions result in deprecation warnings.

19.3.1 함수, 상수 및 예외

소켓 생성

Instances of *`SSLSocket`* must be created using the *`SSLContext.wrap_socket()`* method. The helper function *`create_default_context()`* returns a new context with secure default settings.

기본 컨텍스트와 IPv4/IPv6 이중 스택을 사용하는 클라이언트 소켓 예제:

```
import socket
import ssl

hostname = 'www.python.org'
context = ssl.create_default_context()

with socket.create_connection((hostname, 443)) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

사용자 정의 컨텍스트와 IPv4를 사용하는 클라이언트 소켓 예제:

```
hostname = 'www.python.org'
# PROTOCOL_TLS_CLIENT requires valid cert chain and hostname
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations('path/to/cabundle.pem')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

localhost IPv4에서 리스닝하는 서버 소켓 예제:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain('/path/to/certchain.pem', '/path/to/private.key')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    sock.bind(('127.0.0.1', 8443))
    sock.listen(5)
    with context.wrap_socket(sock, server_side=True) as ssock:
        conn, addr = ssock.accept()
        ...
```

컨텍스트 생성

편리 함수는 공통 목적을 위한 `SSLContext` 객체를 만드는 데 도움이 됩니다.

`ssl.`**`create_default_context`**(*purpose=Purpose.SERVER_AUTH, *, cafile=None, capath=None, cadata=None*)

지정된 *purpose*를 위한 기본 설정으로 새 `SSLContext` 객체를 반환합니다. 설정은 `ssl` 모듈에 의해 선택되며, 일반적으로 `SSLContext` 생성자를 직접 호출할 때 보다 높은 보안 수준을 나타냅니다.

cafile, *capath*, *cadata*는, `SSLContext.load_verify_locations()`에서와 같이, 인증서 확인을 위해 신뢰할 수 있는 선택적 CA 인증서를 나타냅니다. 세 개 모두가 `None`이면, 이 함수는 대신 시스템의 기본 CA 인증서를 신뢰하도록 선택할 수 있습니다.

The settings are: `PROTOCOL_TLS_CLIENT` or `PROTOCOL_TLS_SERVER`, `OP_NO_SSLv2`, and `OP_NO_SSLv3` with high encryption cipher suites without RC4 and without unauthenticated cipher suites. Passing `SERVER_AUTH` as *purpose* sets `verify_mode` to `CERT_REQUIRED` and either loads CA certificates (when at least one of *cafile*, *capath* or *cadata* is given) or uses `SSLContext.load_default_certs()` to load default CA certificates.

`keylog_filename`이 지원되고 환경 변수 `SSLKEYLOGFILE`이 설정될 때, `create_default_context()`는 키 로깅을 활성화합니다.

The default settings for this context include `VERIFY_X509_PARTIAL_CHAIN` and `VERIFY_X509_STRICT`. These make the underlying OpenSSL implementation behave more like a conforming implementation of **RFC 5280**, in exchange for a small amount of incompatibility with older X.509 certificates.

> **참고**
>
> 프로토콜, 옵션, 사이퍼 및 기타 설정은 사전 폐지 없이 언제든지 더욱 제한적인 값으로 변경될 수 있습니다. 이 값은 호환성과 보안 간의 적절한 균형을 나타냅니다.
>
> 응용 프로그램에 특정 설정이 필요하면, `SSLContext`를 만들어 설정을 직접 적용해야 합니다.

> **참고**
>
> 특정 이전 클라이언트나 서버가 이 함수로 만든 `SSLContext`로 연결을 시도할 때 "Protocol or cipher suite mismatch" 라는 에러가 발생하면, 이 함수가 `OP_NO_SSLv3`를 사용해서 제외하는 SSL3.0만 지

원하는 것일 수 있습니다. SSL3.0은 완전히 망가진것으로 널리 인식되고 있습니다. 이 함수를 계속 사용하면서 SSL 3.0 연결을 계속 허용하려면 다음과 같이 다시 활성화할 수 있습니다:

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options &= ~ssl.OP_NO_SSLv3
```

참고

This context enables *VERIFY_X509_STRICT* by default, which may reject pre-**RFC 5280** or malformed certificates that the underlying OpenSSL implementation otherwise would accept. While disabling this is not recommended, you can do so using:

```
ctx = ssl.create_default_context()
ctx.verify_flags &= ~ssl.VERIFY_X509_STRICT
```

Added in version 3.4.

버전 3.4.4에서 변경: RC4는 기본 사이퍼 문자열에서 삭제되었습니다.

버전 3.6에서 변경: ChaCha20/Poly1305가 기본 사이퍼 문자열에 추가되었습니다.

3DES가 기본 사이퍼 문자열에서 삭제되었습니다.

버전 3.8에서 변경: `SSLKEYLOGFILE`에 대한 키 로깅 지원이 추가되었습니다.

버전 3.10에서 변경: The context now uses *PROTOCOL_TLS_CLIENT* or *PROTOCOL_TLS_SERVER* protocol instead of generic *PROTOCOL_TLS*.

버전 3.13에서 변경: The context now uses *VERIFY_X509_PARTIAL_CHAIN* and *VERIFY_X509_STRICT* in its default verify flags.

예외

exception ssl.**SSLError**

하부 SSL 구현(현재 OpenSSL 라이브러리에서 제공)으로부터의 에러를 알리기 위해 발생합니다. 이는 하부 네트워크 연결에 겹쳐진 상위 수준의 암호화와 인증 계층에서 문제가 있음을 나타냅니다. 이 에러는 *OSError*의 서브 형입니다. *SSLError* 인스턴스의 에러 코드와 메시지는 OpenSSL 라이브러리에 의해 제공됩니다.

버전 3.3에서 변경: *SSLError*는 *socket.error*의 서브 형이었습니다.

library

`SSL`, `PEM` 또는 `X509`와 같이, 에러가 발생한 OpenSSL 하위 모듈을 지정하는 문자열 기호입니다. 가능한 값의 범위는 OpenSSL 버전에 따라 다릅니다.

Added in version 3.3.

reason

이 에러가 발생한 이유를 나타내는 문자열 기호, 예를 들어, `CERTIFICATE_VERIFY_FAILED`. 가능한 값의 범위는 OpenSSL 버전에 따라 다릅니다.

Added in version 3.3.

exception ssl.**SSLZeroReturnError**

읽기나 쓰기를 시도하고 SSL 연결이 정상적으로 닫혔을 때 발생하는 *SSLError*의 서브 클래스. 이것이 하부 트랜스포트(TCP 읽기)가 닫혔음을 뜻하지는 않습니다.

Added in version 3.3.

exception ssl.**SSLWantReadError**

데이터를 읽거나 쓰려고 하지만, 요청을 만족하려면 하부 TCP 트랜스포트에서 데이터를 더 수신해야 할 때, 비 블로킹 *SSL* 소켓에 의해 발생하는 *SSLError*의 서브 클래스.

Added in version 3.3.

exception ssl.**SSLWantWriteError**

데이터를 읽거나 쓰려고 하지만, 요청을 만족하려면 하부 TCP 트랜스포트로 데이터를 더 보내야 할 때, 비 블로킹 *SSL* 소켓에 의해 발생하는 *SSLError*의 서브 클래스.

Added in version 3.3.

exception ssl.**SSLSyscallError**

SSL 소켓에서 작업을 수행하는 동안 시스템 에러를 만났을 때 발생하는 *SSLError*의 서브 클래스. 불행히도 원래의 errno 번호를 검사하는 쉬운 방법은 없습니다.

Added in version 3.3.

exception ssl.**SSLEOFError**

SSL 연결이 갑자기 종료되었을 때 발생하는 *SSLError*의 서브 클래스. 일반적으로, 이 에러가 발생하면 하부 트랜스포트를 다시 사용하지 않아야 합니다.

Added in version 3.3.

exception ssl.**SSLCertVerificationError**

인증서 유효성 검사가 실패했을 때 발생하는 *SSLError*의 서브 클래스.

Added in version 3.7.

verify_code

유효성 검사 에러를 나타내는 숫자 에러 번호.

verify_message

사람이 읽을 수 있는 유효성 검사 에러 문자열.

exception ssl.**CertificateError**

*SSLCertVerificationError*의 별칭.

버전 3.7에서 변경: 예외는 이제 *SSLCertVerificationError*의 별칭입니다.

난수 생성

ssl.**RAND_bytes**(*num*, /)

길이 *num*의 암호학적으로 강한 의사 난수 바이트열을 반환합니다. PRNG에 충분한 데이터가 시드(seed) 되지 않았거나 현재 RAND 메서드에서 지원되지 않는 연산이면 *SSLError*를 발생시킵니다. *RAND_status()*를 PRNG의 상태를 확인하는 데 사용할 수 있으며 *RAND_add()*는 PRNG를 시드 하는 데 사용할 수 있습니다.

거의 모든 응용 프로그램에서 *os.urandom()*을 선호합니다.

암호학적으로 강한 생성기의 요구 사항을 얻으려면 위키피디아 기사 Cryptographically secure pseudo-random number generator (CSPRNG)를 읽으십시오.

Added in version 3.3.

ssl.**RAND_status**()

SSL 의사 난수 생성기에 '충분한' 임의성이 시드 되었으면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다. `ssl.RAND_egd()`와 *ssl.RAND_add()*를 사용하여 의사 난수 생성기의 임의성을 높일 수 있습니다.

ssl.**RAND_add**(*bytes*, *entropy*, /)

Mix the given *bytes* into the SSL pseudo-random number generator. The parameter *entropy* (a float) is a lower bound on the entropy contained in string (so you can always use `0.0`). See **RFC 1750** for more information on sources of entropy.

버전 3.5에서 변경: 이제 쓰기 가능한 바이트열류 객체를 받아들입니다.

인증서 처리

ssl.**cert_time_to_seconds**(*cert_time*)

인증서의 "notBefore" 나 "notAfter" 날짜를 나타내는 `"%b %d %H:%M:%S %Y %Z"` strptime 형식(C 로케일)의 `cert_time` 문자열이 지정하는 시간을 Epoch 이후 초 단위로 반환합니다.

여기 예제가 있습니다:

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

"notBefore" 나 "notAfter" 날짜는 GMT(**RFC 5280**)를 사용해야 합니다.

버전 3.5에서 변경: 입력된 시간을 입력 문자열의 'GMT' 시간대로 지정된 UTC 시간으로 해석합니다. 이전에는 지역 시간대가 사용되었습니다. 정수를 반환합니다(입력 형식에는 부분 초가 없습니다).

ssl.**get_server_certificate**(*addr, ssl_version=PROTOCOL_TLS_CLIENT, ca_certs=None*[, *timeout*])

Given the address `addr` of an SSL-protected server, as a (*hostname, port-number*) pair, fetches the server' s certificate, and returns it as a PEM-encoded string. If `ssl_version` is specified, uses that version of the SSL protocol to attempt to connect to the server. If *ca_certs* is specified, it should be a file containing a list of root certificates, the same format as used for the *cafile* parameter in `SSLContext.load_verify_locations()`. The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails. A timeout can be specified with the `timeout` parameter.

버전 3.3에서 변경: 이 함수는 이제 IPv6와 호환됩니다.

버전 3.5에서 변경: 최신 서버와의 호환성을 최대화하기 위해 기본 *ssl_version*이 `PROTOCOL_SSLv3`에서 `PROTOCOL_TLS`로 변경되었습니다.

버전 3.10에서 변경: The *timeout* parameter was added.

ssl.**DER_cert_to_PEM_cert**(*der_cert_bytes*)

인증서가 DER-인코딩된 바이트열로 주어지면, 같은 인증서의 PEM-인코딩된 문자열 버전을 반환합니다.

ssl.**PEM_cert_to_DER_cert**(*pem_cert_string*)

인증서가 ASCII PEM 문자열로 주어지면, 같은 인증서의 DER-인코딩된 바이트열 시퀀스를 반환합니다.

ssl.**get_default_verify_paths**()

OpenSSL의 기본 cafile 및 capath에 대한 경로가 있는 네임드 튜플을 반환합니다. 경로는 `SSLContext.set_default_verify_paths()`에서 사용하는 경로와 같습니다. 반환 값은 네임드 튜플 `DefaultVerifyPaths`입니다.:

- `cafile` - cafile에 대한 확인된 경로나 파일이 존재하지 않으면 `None`,
- `capath` - capath에 대한 확인된 경로나 디렉터리가 존재하지 않으면 `None`,
- `openssl_cafile_env` - cafile을 가리키는 OpenSSL의 환경 키,
- `openssl_cafile` - cafile에 대한 하드 코딩된 경로,
- `openssl_capath_env` - capath를 가리키는 OpenSSL의 환경 키,
- `openssl_capath` - capath 디렉터리에 대한 하드 코딩된 경로

Added in version 3.4.

ssl.**enum_certificates**(*store_name*)

윈도우의 시스템 인증서 저장소에서 인증서를 꺼냅니다. *store_name*은 CA, ROOT 또는 MY 중 하나일 수 있습니다. 윈도우가 추가 인증서 저장소를 제공 할 수도 있습니다.

이 함수는 (cert_bytes, encoding_type, trust) 튜플의 리스트를 반환합니다. encoding_type은 cert_bytes의 인코딩을 지정합니다. X.509 ASN.1 데이터를 위한 x509_asn이거나 PKCS#7 ASN.1 데이터를 위한 pkcs_7_asn입니다. Trust는 인증서의 목적을 OIDS 집합으로 지정하거나, 인증서가 모든 목적에 대해 신뢰할 수 있으면 정확히 True입니다.

예제:

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

가용성: Windows.

Added in version 3.4.

ssl.**enum_crls**(*store_name*)

윈도우의 시스템 인증서 저장소에서 CRL을 꺼냅니다. *store_name*는 CA, ROOT 또는 MY 중 하나일 수 있습니다. 윈도우가 추가 인증서 저장소를 제공 할 수도 있습니다.

이 함수는 (cert_bytes, encoding_type, trust) 튜플의 리스트를 반환합니다. encoding_type은 cert_bytes의 인코딩을 지정합니다. X.509 ASN.1 데이터를 위한 x509_asn이거나 PKCS#7 ASN.1 데이터를 위한 pkcs_7_asn입니다.

가용성: Windows.

Added in version 3.4.

상수

모든 상수는 이제 *enum.IntEnum* 이나 *enum.IntFlag* 컬렉션입니다.

Added in version 3.6.

ssl.**CERT_NONE**

Possible value for *SSLContext.verify_mode*. Except for *PROTOCOL_TLS_CLIENT*, it is the default mode. With client-side sockets, just about any cert is accepted. Validation errors, such as untrusted or expired cert, are ignored and do not abort the TLS/SSL handshake.

서버 모드에서는, 클라이언트에서 인증서를 요청하지 않으므로 클라이언트는 클라이언트 인증서 인증을 위해 인증서를 보내지 않습니다.

아래의 보안 고려 사항의 논의를 참조하십시오.

ssl.**CERT_OPTIONAL**

Possible value for *SSLContext.verify_mode*. In client mode, *CERT_OPTIONAL* has the same meaning as *CERT_REQUIRED*. It is recommended to use *CERT_REQUIRED* for client-side sockets instead.

서버 모드에서는, 클라이언트 인증서 요청이 클라이언트로 전송됩니다. 클라이언트는 요청을 무시하거나 TLS 클라이언트 인증서 인증을 수행하기 위해 인증서를 보낼 수 있습니다. 클라이언트가 인증서를 보내기로 선택하면, 인증서가 유효성 검사됩니다. 모든 유효성 검사 에러는, TLS 핸드셰이크를 즉시 중단합니다.

Use of this setting requires a valid set of CA certificates to be passed to *SSLContext.load_verify_locations()*.

ssl.**CERT_REQUIRED**

Possible value for *SSLContext.verify_mode*. In this mode, certificates are required from the other side of the socket connection; an *SSLError* will be raised if no certificate is provided, or if its validation fails. This mode is **not** sufficient to verify a certificate in client mode as it does not match hostnames. *check_hostname* must be enabled as well to verify the authenticity of a cert. *PROTOCOL_TLS_CLIENT* uses *CERT_REQUIRED* and enables *check_hostname* by default.

서버 소켓에서, 이 모드는 필수 TLS 클라이언트 인증서 인증을 제공합니다. 클라이언트 인증서 요청이 클라이언트에 보내지고 클라이언트는 유효하고 신뢰할 수 있는 인증서를 제공해야 합니다.

Use of this setting requires a valid set of CA certificates to be passed to *SSLContext.load_verify_locations()*.

class ssl.**VerifyMode**

CERT_* 상수의 *enum.IntEnum* 컬렉션.

Added in version 3.6.

ssl.**VERIFY_DEFAULT**

*SSLContext.verify_flags*의 가능한 값. 이 모드에서는 인증서 해지 목록(CRL)을 검사하지 않습니다. 기본적으로 OpenSSL은 CRL을 요구하지도 검사하지도 않습니다.

Added in version 3.4.

ssl.**VERIFY_CRL_CHECK_LEAF**

*SSLContext.verify_flags*의 가능한 값. 이 모드에서는, 피어 인증서만 확인할 뿐 중간 CA 인증서는 확인하지 않습니다. 이 모드는 피어 인증서의 발급자(그것의 직계 조상 CA)가 서명한 유효한 CRL을 요구합니다. 적절한 CRL이 *SSLContext.load_verify_locations*로 로드되지 않았으면 유효성 검사가 실패합니다.

Added in version 3.4.

ssl.**VERIFY_CRL_CHECK_CHAIN**

*SSLContext.verify_flags*의 가능한 값. 이 모드에서는, 피어 인증서 체인의 모든 인증서에 대한 CRL이 확인됩니다.

Added in version 3.4.

ssl.**VERIFY_X509_STRICT**

망가진 X.509 인증서에 대한 우회를 사용하지 못하도록 하는 *SSLContext.verify_flags*의 가능한 값.

Added in version 3.4.

ssl.**VERIFY_ALLOW_PROXY_CERTS**

Possible value for *SSLContext.verify_flags* to enables proxy certificate verification.

Added in version 3.10.

ssl.**VERIFY_X509_TRUSTED_FIRST**

*SSLContext.verify_flags*의 가능한 값. OpenSSL이 인증서의 유효성을 검사하기 위해 트러스트 체인을 구축할 때 신뢰할 수 있는 인증서를 선호하도록 지시합니다. 이 플래그는 기본적으로 활성화됩니다.

Added in version 3.4.4.

ssl.**VERIFY_X509_PARTIAL_CHAIN**

Possible value for *SSLContext.verify_flags*. It instructs OpenSSL to accept intermediate CAs in the trust store to be treated as trust-anchors, in the same way as the self-signed root CA certificates. This makes it possible to trust certificates issued by an intermediate CA without having to trust its ancestor root CA.

Added in version 3.10.

class ssl.**VerifyFlags**

VERIFY_* 상수의 *enum.IntFlag* 컬렉션.

Added in version 3.6.

ssl.**PROTOCOL_TLS**

클라이언트와 서버가 모두 지원하는 가장 높은 프로토콜 버전을 선택합니다. 이름에도 불구하고, 이 옵션은 "SSL" 과 "TLS" 프로토콜을 모두 선택할 수 있습니다.

Added in version 3.6.

버전 3.10부터 폐지됨: TLS clients and servers require different default settings for secure communication. The generic TLS protocol constant is deprecated in favor of *PROTOCOL_TLS_CLIENT* and *PROTOCOL_TLS_SERVER*.

ssl.**PROTOCOL_TLS_CLIENT**

Auto-negotiate the highest protocol version that both the client and server support, and configure the context client-side connections. The protocol enables *CERT_REQUIRED* and *check_hostname* by default.

Added in version 3.6.

ssl.**PROTOCOL_TLS_SERVER**

Auto-negotiate the highest protocol version that both the client and server support, and configure the context server-side connections.

Added in version 3.6.

ssl.**PROTOCOL_SSLv23**

*PROTOCOL_TLS*의 별칭.

버전 3.6부터 폐지됨: 대신 *PROTOCOL_TLS*를 사용하십시오.

ssl.**PROTOCOL_SSLv3**

채널 암호화 프로토콜로 SSL 버전 3을 선택합니다.

This protocol is not available if OpenSSL is compiled with the `no-ssl3` option.

> **경고**
>
> SSL 버전 3은 안전하지 않습니다. 사용하지 말도록 강력히 권고합니다.

버전 3.6부터 폐지됨: OpenSSL has deprecated all version specific protocols. Use the default protocol *PROTOCOL_TLS_SERVER* or *PROTOCOL_TLS_CLIENT* with *SSLContext.minimum_version* and *SSLContext.maximum_version* instead.

ssl.**PROTOCOL_TLSv1**

채널 암호화 프로토콜로 TLS 버전 1.0을 선택합니다.

버전 3.6부터 폐지됨: OpenSSL has deprecated all version specific protocols.

ssl.**PROTOCOL_TLSv1_1**

채널 암호화 프로토콜로 TLS 버전 1.1을 선택합니다. openssl 버전 1.0.1+ 에서만 사용할 수 있습니다.

Added in version 3.4.

버전 3.6부터 폐지됨: OpenSSL has deprecated all version specific protocols.

ssl.**PROTOCOL_TLSv1_2**

Selects TLS version 1.2 as the channel encryption protocol. Available only with openssl version 1.0.1+.

Added in version 3.4.

버전 3.6부터 폐지됨: OpenSSL has deprecated all version specific protocols.

ssl.**OP_ALL**

다른 SSL 구현에 있는 다양한 버그에 대한 해결 방법을 활성화합니다. 이 옵션은 기본적으로 설정됩니다. 반드시 OpenSSL의 `SSL_OP_ALL` 상수와 같은 플래그를 설정할 필요는 없습니다.

Added in version 3.2.

ssl.**OP_NO_SSLv2**

SSLv2 연결을 방지합니다. 이 옵션은 *PROTOCOL_TLS*와 결합해서만 적용할 수 있습니다. 피어가 SSLv2를 프로토콜 버전으로 선택하지 못하도록 합니다.

Added in version 3.2.

버전 3.6부터 폐지됨: SSLv2는 폐지되었습니다.

ssl.**OP_NO_SSLv3**

SSLv3 연결을 방지합니다. 이 옵션은 *PROTOCOL_TLS*와 결합해서만 적용할 수 있습니다. 피어가 프로토콜 버전으로 SSLv3을 선택하지 못하게 합니다.

Added in version 3.2.

버전 3.6부터 폐지됨: SSLv3은 폐지되었습니다.

ssl.**OP_NO_TLSv1**

TLSv1 연결을 금지합니다. 이 옵션은 *PROTOCOL_TLS*와 결합해서만 적용할 수 있습니다. 피어가 TLSv1을 프로토콜 버전으로 선택하지 못하게 합니다.

Added in version 3.2.

버전 3.7부터 폐지됨: 이 옵션은 OpenSSL 1.1.0부터 폐지되었습니다, 새로운 *SSLContext.minimum_version*과 *SSLContext.maximum_version*을 대신 사용하십시오.

ssl.**OP_NO_TLSv1_1**

TLSv1.1 연결을 금지합니다. 이 옵션은 *PROTOCOL_TLS*와 결합해서만 적용할 수 있습니다. 피어가 TLSv1.1을 프로토콜 버전으로 선택하지 못하게 합니다. openssl 버전 1.0.1+ 에서만 사용할 수 있습니다.

Added in version 3.4.

버전 3.7부터 폐지됨: 이 옵션은 OpenSSL 1.1.0부터 폐지되었습니다.

ssl.**OP_NO_TLSv1_2**

TLSv1.2 연결을 방지합니다. 이 옵션은 *PROTOCOL_TLS*와 결합해서만 적용할 수 있습니다. 피어가 프로토콜 버전으로 TLSv1.2를 선택하지 못하게 합니다. openssl 버전 1.0.1+ 에서만 사용할 수 있습니다.

Added in version 3.4.

버전 3.7부터 폐지됨: 이 옵션은 OpenSSL 1.1.0부터 폐지되었습니다.

ssl.**OP_NO_TLSv1_3**

TLSv1.3 연결을 방지합니다. 이 옵션은 *PROTOCOL_TLS*와 결합해서만 적용할 수 있습니다. 피어가 프로토콜 버전으로 TLSv1.3을 선택하지 못하게 합니다. TLS 1.3은 OpenSSL 1.1.1 이상에서 사용할 수 있습니다. 파이썬이 OpenSSL의 이전 버전에 대해 컴파일되면, 플래그의 기본값은 *0*입니다.

Added in version 3.6.3.

버전 3.7부터 폐지됨: The option is deprecated since OpenSSL 1.1.0. It was added to 2.7.15 and 3.6.3 for backwards compatibility with OpenSSL 1.0.2.

ssl.**OP_NO_RENEGOTIATION**

TLSv1.2와 그 이전 버전에서 모든 재협상을 비활성화합니다. HelloRequest 메시지를 보내지 않고, ClientHello를 통한 재협상 요청을 무시합니다.

이 옵션은 OpenSSL 1.1.0h 이상에서만 사용할 수 있습니다.

Added in version 3.7.

ssl.**OP_CIPHER_SERVER_PREFERENCE**

클라이언트보다는 서버의 사이퍼 순서 선호를 사용합니다. 이 옵션은 클라이언트 소켓과 SSLv2 서버 소켓에는 영향을 미치지 않습니다.

Added in version 3.3.

ssl.**OP_SINGLE_DH_USE**

Prevents reuse of the same DH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

Added in version 3.3.

ssl.**OP_SINGLE_ECDH_USE**

Prevents reuse of the same ECDH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

Added in version 3.3.

ssl.**OP_ENABLE_MIDDLEBOX_COMPAT**

TLS 1.3 연결을 더 TLS 1.2 연결처럼 보이게 하려고 TLS 1.3 핸드 셰이크에서 더미 암호 변경 사양 (CCS - Change Cipher Spec) 메시지를 보냅니다.

이 옵션은 OpenSSL 1.1.1 이상에서만 사용할 수 있습니다.

Added in version 3.8.

ssl.**OP_NO_COMPRESSION**

SSL 채널에서 압축을 사용하지 않습니다. 응용 프로그램 프로토콜이 자체 압축 방법을 지원할 때 유용합니다.

Added in version 3.3.

class ssl.**Options**

OP_* 상수의 *enum.IntFlag* 컬렉션.

ssl.**OP_NO_TICKET**

클라이언트 측에서 세션 티켓을 요청하지 못하게 합니다.

Added in version 3.6.

ssl.**OP_IGNORE_UNEXPECTED_EOF**

Ignore unexpected shutdown of TLS connections.

This option is only available with OpenSSL 3.0.0 and later.

Added in version 3.10.

ssl.**OP_ENABLE_KTLS**

Enable the use of the kernel TLS. To benefit from the feature, OpenSSL must have been compiled with support for it, and the negotiated cipher suites and extensions must be supported by it (a list of supported ones may vary by platform and kernel version).

Note that with enabled kernel TLS some cryptographic operations are performed by the kernel directly and not via any available OpenSSL Providers. This might be undesirable if, for example, the application requires all cryptographic operations to be performed by the FIPS provider.

This option is only available with OpenSSL 3.0.0 and later.

Added in version 3.12.

ssl.**OP_LEGACY_SERVER_CONNECT**

Allow legacy insecure renegotiation between OpenSSL and unpatched servers only.

Added in version 3.12.

ssl.**HAS_ALPN**

OpenSSL 라이브러리가 **RFC 7301**에서 설명한 대로 응용 계층 프로토콜 협상(*Application-Layer Protocol Negotiation*) TLS 확장에 대한 지원을 기본 제공하는지 여부

Added in version 3.5.

ssl.**HAS_NEVER_CHECK_COMMON_NAME**

OpenSSL 라이브러리가 SCN(subject common name)을 검사하지 않는 지원을 기본 제공하고 *SSLContext.hostname_checks_common_name*가 쓰기 가능한지 아닌지.

Added in version 3.7.

ssl.**HAS_ECDH**

OpenSSL 라이브러리가 타원 곡선(Elliptic Curve) 기반 Diffie-Hellman 키 교환 지원을 기본 제공하는지 여부. 기능이 배포자에 의해 명시적으로 비활성화되어 있지 않은 한, 참이어야 합니다.

Added in version 3.3.

ssl.**HAS_SNI**

OpenSSL 라이브러리가 서버 이름 표시(*Server Name Indication*) 확장(**RFC 6066**에 정의된 대로)에 대한 지원을 기본 제공하는지 여부.

Added in version 3.2.

ssl.**HAS_NPN**

OpenSSL 라이브러리가 Application Layer Protocol Negotiation에 설명된 대로 *NPN*(*Next Protocol Negotiation*)에 대한 지원을 기본 제공하는지 여부. 참이면 `SSLContext.set_npn_protocols()` 메서드를 사용하여 지원할 프로토콜을 알릴 수 있습니다.

Added in version 3.3.

ssl.**HAS_SSLv2**

OpenSSL 라이브러리가 SSL 2.0 프로토콜 지원을 기본 제공하는지 여부

Added in version 3.7.

ssl.**HAS_SSLv3**

OpenSSL 라이브러리가 SSL 3.0 프로토콜 지원을 기본 제공하는지 여부

Added in version 3.7.

ssl.**HAS_TLSv1**

OpenSSL 라이브러리가 TLS 1.0 프로토콜 지원을 기본 제공하는지 여부

Added in version 3.7.

ssl.**HAS_TLSv1_1**

OpenSSL 라이브러리가 TLS 1.1 프로토콜 지원을 기본 제공하는지 여부

Added in version 3.7.

ssl.**HAS_TLSv1_2**

OpenSSL 라이브러리가 TLS 1.2 프로토콜 지원을 기본 제공하는지 여부

Added in version 3.7.

ssl.**HAS_TLSv1_3**

OpenSSL 라이브러리가 TLS 1.3 프로토콜 지원을 기본 제공하는지 여부

Added in version 3.7.

ssl.**HAS_PSK**

Whether the OpenSSL library has built-in support for TLS-PSK.

Added in version 3.13.

ssl.**HAS_PHA**

Whether the OpenSSL library has built-in support for TLS-PHA.

Added in version 3.14.

ssl.**CHANNEL_BINDING_TYPES**

지원되는 TLS 채널 바인딩 유형의 리스트. 이 리스트의 문자열은 `SSLSocket.get_channel_binding()`에 대한 인자로 사용될 수 있습니다.

Added in version 3.3.

ssl.**OPENSSL_VERSION**

인터프리터에 의해 로드된 OpenSSL 라이브러리의 버전 문자열:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

Added in version 3.2.

ssl.**OPENSSL_VERSION_INFO**

OpenSSL 라이브러리에 대한 버전 정보를 나타내는 5개의 정수 튜플:

```
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

Added in version 3.2.

ssl.**OPENSSL_VERSION_NUMBER**

단일 정수로 표현되는, OpenSSL 라이브러리의 원시 버전 번호:

```
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```

Added in version 3.2.

ssl.**ALERT_DESCRIPTION_HANDSHAKE_FAILURE**

ssl.**ALERT_DESCRIPTION_INTERNAL_ERROR**

ALERT_DESCRIPTION_*

RFC 5246 및 기타의 경고 설명. IANA TLS Alert Registry에는 이 목록과 그 의미가 정의된 RFC에 대한 참조가 들어 있습니다.

*SSLContext.set_servername_callback()*에서 콜백 함수의 반환 값으로 사용됩니다.

Added in version 3.4.

class ssl.**AlertDescription**

ALERT_DESCRIPTION_* 상수의 *enum.IntEnum* 컬렉션.

Added in version 3.6.

Purpose.**SERVER_AUTH**

Option for *create_default_context()* and *SSLContext.load_default_certs()*. This value indicates that the context may be used to authenticate web servers (therefore, it will be used to create client-side sockets).

Added in version 3.4.

Purpose.**CLIENT_AUTH**

Option for *create_default_context()* and *SSLContext.load_default_certs()*. This value indicates that the context may be used to authenticate web clients (therefore, it will be used to create server-side sockets).

Added in version 3.4.

class ssl.**SSLErrorNumber**

SSL_ERROR_* 상수의 *enum.IntEnum* 컬렉션.

Added in version 3.6.

class ssl.**TLSVersion**

*SSLContext.maximum_version*과 *SSLContext.minimum_version* 용 SSL과 TLS 버전의 *enum.IntEnum* 컬렉션.

Added in version 3.7.

TLSVersion.**MINIMUM_SUPPORTED**

TLSVersion.**MAXIMUM_SUPPORTED**

지원되는 SSL 또는 TLS 버전의 최소 또는 최대. 이것들은 마법 상수(magic constant)입니다. 이들의 값은 사용 가능한 가장 낮거나 높은 TLS/SSL 버전을 반영하지 않습니다.

TLSVersion.**SSLv3**

TLSVersion.**TLSv1**

TLSVersion.**TLSv1_1**

TLSVersion.**TLSv1_2**

TLSVersion.**TLSv1_3**

SSL 3.0 에서 TLS 1.3.

버전 3.10부터 폐지됨: All *TLSVersion* members except *TLSVersion.TLSv1_2* and *TLSVersion.TLSv1_3* are deprecated.

19.3.2 SSL 소켓

class ssl.**SSLSocket**(*socket.socket*)

SSL 소켓은 다음과 같은 소켓 객체 메서드를 제공합니다:

- *accept()*
- *bind()*
- *close()*
- *connect()*
- *detach()*
- *fileno()*
- *getpeername()*, *getsockname()*
- *getsockopt()*, *setsockopt()*
- *gettimeout()*, *settimeout()*, *setblocking()*
- *listen()*
- *makefile()*
- *recv()*, *recv_into()* (but passing a non-zero `flags` argument is not allowed)
- *send()*, *sendall()* (with the same limitation)
- *sendfile()* (but *os.sendfile* will be used for plain-text sockets only, else *send()* will be used)
- *shutdown()*

그러나 SSL (및 TLS) 프로토콜은 TCP 위에 자체 프레임을 가지고 있으므로, SSL 소켓 추상화는 특정 측면에서 정상적인 OS 수준 소켓의 사양에서 벗어날 수 있습니다. 특히 비 블로킹 소켓에 대한 참고 사항을 보십시오.

*SSLSocket*의 인스턴스는 *SSLContext.wrap_socket()* 메서드를 사용하여 만들어야 합니다.

버전 3.5에서 변경: `sendfile()` 메서드가 추가되었습니다.

버전 3.5에서 변경: The `shutdown()` does not reset the socket timeout each time bytes are received or sent. The socket timeout is now the maximum total duration of the shutdown.

버전 3.6부터 폐지됨: *SSLSocket* 인스턴스를 직접 만드는 것은 폐지되었습니다, *SSLContext.wrap_socket()*를 사용하여 소켓을 감싸십시오.

버전 3.7에서 변경: *SSLSocket* 인스턴스는 *wrap_socket()*로 만들어야 합니다. 이전 버전에서는 직접 인스턴스를 만들 수 있었습니다. 이것은 문서로 만들어지거나 공식적으로 지원된 적이 없습니다.

버전 3.10에서 변경: Python now uses `SSL_read_ex` and `SSL_write_ex` internally. The functions support reading and writing of data larger than 2 GB. Writing zero-length data no longer fails with a protocol violation error.

SSL 소켓에는 다음과 같은 추가 메서드와 어트리뷰트도 있습니다:

`SSLSocket.`**`read`**(*len=1024*, *buffer=None*)

SSL 소켓에서 최대 *len* 바이트의 데이터를 읽고 그 결과를 `bytes` 인스턴스로 반환합니다. *buffer*가 지정되면, 대신 버퍼로 읽어 들이고, 읽은 바이트 수를 반환합니다.

소켓이 비 블로킹이고 읽기가 블록 되려고 하면 *SSLWantReadError* 나 *SSLWantWriteError*를 발생시킵니다.

언제나 재협상이 가능하므로, *read()*를 호출해도 쓰기 연산이 발생할 수 있습니다.

버전 3.5에서 변경: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration to read up to *len* bytes.

버전 3.6부터 폐지됨: *read()* 대신 `recv()`를 사용하십시오.

`SSLSocket.`**`write`**(*data*)

Write *data* to the SSL socket and return the number of bytes written. The *data* argument must be an object supporting the buffer interface.

소켓이 비 블로킹이고, 쓰기가 블록 하려고 하면 *SSLWantReadError* 나 *SSLWantWriteError*를 발생시킵니다.

언제나 재협상이 가능하므로, *write()*를 호출해도 읽기 연산이 발생할 수 있습니다.

버전 3.5에서 변경: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration to write *data*.

버전 3.6부터 폐지됨: *write()* 대신 `send()`를 사용하십시오.

> **참고**
>
> *read()* 과 *write()* 메서드는 암호화되지 않은 응용 프로그램 수준 데이터를 읽고 쓰고 그것을 암호화되고 와이어 수준(wire-level) 데이터로 복호화/암호화하는 저수준 메서드입니다. 이 메서드는 활성화된 SSL 연결, 즉, 핸드 셰이크가 완료되고, *SSLSocket.unwrap()*가 호출되지 않은 것이 필요합니다.
>
> 일반적으로 이러한 메서드 대신 *recv()* 와 *send()*와 같은 소켓 API 메서드를 사용해야합니다.

`SSLSocket.`**`do_handshake`**(*block=False*)

SSL 설정 핸드 셰이크를 수행합니다.

If *block* is true and the timeout obtained by *gettimeout()* is zero, the socket is set in blocking mode until the handshake is performed.

버전 3.4에서 변경: The handshake method also performs `match_hostname()` when the *check_hostname* attribute of the socket' s *context* is true.

버전 3.5에서 변경: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration of the handshake.

버전 3.7에서 변경: Hostname or IP address is matched by OpenSSL during handshake. The function `match_hostname()` is no longer used. In case OpenSSL refuses a hostname or IP address, the handshake is aborted early and a TLS alert message is sent to the peer.

SSLSocket.**getpeercert**(*binary_form=False*)

연결의 다른 끝의 피어에 대한 인증서가 없으면 None을 반환합니다. SSL 핸드 셰이크가 아직 수행되지 않았으면, *ValueError*를 발생시킵니다.

binary_form 매개 변수가 *False*이고, 피어에서 인증서를 받았으면, 이 메서드는 *dict* 인스턴스를 반환합니다. 인증서의 유효성을 검사하지 않았으면, 딕셔너리는 비어 있습니다. 인증서의 유효성을 검사했으면 subject(인증서가 발행된 주체)와 issuer(인증서를 발급한 주체)와 같은 몇 가지 키가 있는 dict를 반환합니다. 인증서가 *SAN*(*Subject Alternative Name*) 확장(**RFC 3280** 참조)의 인스턴스를 포함하면 딕셔너리에 subjectAltName 키도 있습니다.

subject 와 issuer 필드는 각 필드에 대한 인증서의 데이터 구조에 제공된 RDN(relative distinguished name)의 시퀀스를 포함하는 튜플이며, 각 RDN은 이름-값 쌍의 시퀀스입니다. 실제 예를 들어 보겠습니다:

```
{'issuer': ((('countryName', 'IL'),),
            (('organizationName', 'StartCom Ltd.'),),
            (('organizationalUnitName',
              'Secure Digital Certificate Signing'),),
            (('commonName',
              'StartCom Class 2 Primary Intermediate Server CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
             (('commonName', '*.eff.org'),),
             (('emailAddress', 'hostmaster@eff.org'),)),
 'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
 'version': 3}
```

binary_form 매개 변수가 *True*이고, 인증서가 제공되었으면, 이 메서드는 전체 인증서의 DER-인코딩 형식을 바이트 시퀀스로 반환하고, 피어가 인증서를 제공하지 않았으면 *None*을 반환합니다. 피어가 인증서를 제공하는지는 SSL 소켓의 역할에 따라 다릅니다:

- 클라이언트 SSL 소켓의 경우, 서버는 유효성 검사가 필요한지에 관계없이 항상 인증서를 제공합니다.
- 서버 SSL 소켓의 경우, 클라이언트는 서버가 요청할 때만 인증서를 제공합니다; 따라서 *CERT_NONE*(*CERT_OPTIONAL* 나 *CERT_REQUIRED* 대신)을 사용하면 *getpeercert()*는 *None*을 반환합니다.

See also *SSLContext.check_hostname*.

버전 3.2에서 변경: 반환된 딕셔너리에는 issuer 와 notBefore와 같은 추가 항목이 포함됩니다.

버전 3.4에서 변경: 핸드 셰이크가 완료되지 않았으면 *ValueError*가 발생합니다. 반환된 딕셔너리에는 crlDistributionPoints, caIssuers 및 OCSP URI와 같은 추가 X509v3 확장 항목이 포함됩니다.

버전 3.9에서 변경: IPv6 주소 문자열에는 더는 후행 줄 바꿈이 없습니다.

SSLSocket.**get_verified_chain**()

Returns verified certificate chain provided by the other end of the SSL channel as a list of DER-encoded bytes. If certificate verification was disabled method acts the same as *get_unverified_chain()*.

Added in version 3.13.

SSLSocket.**get_unverified_chain**()

Returns raw certificate chain provided by the other end of the SSL channel as a list of DER-encoded bytes.

Added in version 3.13.

SSLSocket.**cipher**()

사용되는 사이퍼의 이름, 그것의 사용을 정의하는 SSL 프로토콜의 버전 및 사용되는 비밀 비트의 수를 포함하는 3-튜플을 반환합니다. 연결이 이루어지지 않았으면 None을 반환합니다.

SSLSocket.**shared_ciphers**()

Return the list of ciphers available in both the client and server. Each entry of the returned list is a three-value tuple containing the name of the cipher, the version of the SSL protocol that defines its use, and the number of secret bits the cipher uses. *shared_ciphers()* returns None if no connection has been established or the socket is a client socket.

Added in version 3.5.

SSLSocket.**compression**()

사용되는 압축 알고리즘을 문자열로 반환하거나, 연결이 압축되지 않으면 None을 반환합니다.

상위-수준 프로토콜이 자체 압축 메커니즘을 지원하면, *OP_NO_COMPRESSION*을 사용하여 SSL-수준 압축을 비활성화할 수 있습니다.

Added in version 3.3.

SSLSocket.**get_channel_binding**(*cb_type='tls-unique'*)

현재 연결에 대한 채널 바인딩 데이터를 바이트열 객체로 가져옵니다. 연결되어 있지 않거나 핸드 셰이크가 완료되지 않았으면 None을 반환합니다.

cb_type 매개 변수를 사용하여 원하는 채널 바인딩 유형을 선택할 수 있습니다. 유효한 채널 바인딩 유형은 *CHANNEL_BINDING_TYPES* 리스트에 나열됩니다. 현재는 **RFC 5929**가 정의한 'tls-unique' 채널 바인딩만 지원됩니다. 지원되지 않는 채널 바인딩 유형이 요청되면 *ValueError*가 발생합니다.

Added in version 3.3.

SSLSocket.**selected_alpn_protocol**()

TLS 핸드 셰이크 중에 선택된 프로토콜을 반환합니다. *SSLContext.set_alpn_protocols()*가 호출되지 않았거나, 상대방이 ALPN을 지원하지 않거나, 이 소켓이 클라이언트가 제안한 프로토콜 중 어떤 것도 지원하지 않거나, 핸드 셰이크가 아직 발생하지 않았으면 None이 반환됩니다.

Added in version 3.5.

SSLSocket.**selected_npn_protocol**()

TLS/SSL 핸드 셰이크 중에 선택된 상위-수준의 프로토콜을 반환합니다. *SSLContext.set_npn_protocols()*가 호출되지 않았거나, 상대방이 NPN을 지원하지 않거나, 핸드 셰이크가 아직 발생하지 않았으면 None을 반환합니다.

Added in version 3.3.

버전 3.10부터 폐지됨: NPN has been superseded by ALPN

SSLSocket.**unwrap**()

SSL 종료 핸드 셰이크를 수행해서 하부 소켓에서 TLS 계층을 제거하고, 하부 소켓 객체를 반환합니다. 이것은 연결을 통한 암호화된 연산에서 암호화되지 않은 것으로 이동하는 데 사용할 수 있습니다. 원래 소켓이 아닌 반환된 소켓을 연결의 다른 쪽과 계속 통신하기 위해 항상 사용해야 합니다.

SSLSocket.**verify_client_post_handshake**()

TLS 1.3 클라이언트로부터 PHA(post-handshake authentication)를 요청합니다. PHA는 양쪽에서 PHA가 활성화된 초기 TLS 핸드 셰이크 후에 서버 측 소켓에서 TLS 1.3 연결에 대해서만 시작할 수 있습니다, *SSLContext.post_handshake_auth*를 참조하세요.

이 메서드는 즉시 인증서 교환을 수행하지 않습니다. 서버 측은 다음 쓰기 이벤트 중에 CertificateRequest를 보내고 클라이언트가 다음 읽기 이벤트에서 인증서로 응답할 것으로 기대합니다.

사전 조건이 모두 충족되지 않으면 (예를 들어, TLS 1.3이 아니거나 PHA가 활성화되지 않았을 때), *SSLError*가 발생합니다.

> **참고**
>
> OpenSSL 1.1.1과 TLS 1.3이 활성화된 경우에만 사용할 수 있습니다. TLS 1.3 지원이 없으면, 이 메서드는 *NotImplementedError*를 발생시킵니다.

Added in version 3.8.

`SSLSocket.`**`version`**`()`

Return the actual SSL protocol version negotiated by the connection as a string, or `None` if no secure connection is established. As of this writing, possible return values include `"SSLv2"`, `"SSLv3"`, `"TLSv1"`, `"TLSv1.1"` and `"TLSv1.2"`. Recent OpenSSL versions may define more return values.

Added in version 3.5.

`SSLSocket.`**`pending`**`()`

접속에 계류 중인, 읽기용으로 이미 복호화된 바이트의 수를 돌려줍니다.

`SSLSocket.`**`context`**

The *SSLContext* object this SSL socket is tied to.

Added in version 3.2.

`SSLSocket.`**`server_side`**

서버 측 소켓에서는 `True`이고 클라이언트 측 소켓에서는 `False` 인 논릿값.

Added in version 3.2.

`SSLSocket.`**`server_hostname`**

서버의 호스트 이름: *str* 형, 또는 서버 측 소켓이거나 호스트 이름이 생성자에 지정되지 않았으면 `None`.

Added in version 3.2.

버전 3.7에서 변경: 어트리뷰트는 이제 항상 ASCII 텍스트입니다. `server_hostname`이 국제화 된 도메인 이름(IDN)일 때, 이 어트리뷰트는 이제 U-레이블 형식(`"pythön.org"`) 대신 A-레이블 형식(`"xn--pythn-mua.org"`)을 저장합니다.

`SSLSocket.`**`session`**

이 SSL 연결을 위한 *SSLSession*. 이 세션은 TLS 핸드 셰이크가 수행된 후 클라이언트와 서버 측 소켓에서 사용할 수 있습니다. 클라이언트 소켓의 경우 세션을 다시 사용하기 위해 *do_handshake()*가 호출되기 전에 세션을 설정할 수 있습니다.

Added in version 3.6.

`SSLSocket.`**`session_reused`**

Added in version 3.6.

19.3.3 SSL 컨텍스트

Added in version 3.2.

SSL 컨텍스트는 SSL 구성 옵션, 인증서 및 개인 키와 같이 단일 SSL 연결보다 수명이 긴 다양한 데이터를 보관합니다. 또한, 같은 클라이언트의 반복된 연결 속도를 높이기 위해 서버 측 소켓에 대한 SSL 세션 캐시를 관리합니다.

`class` `ssl.`**`SSLContext`**(*protocol=None*)

새 SSL 컨텍스트를 만듭니다. *protocol*를 전달할 수 있는데, 이 모듈에 정의된 `PROTOCOL_*` 상수 중 하나여야 합니다. 매개 변수는 사용할 SSL 프로토콜의 버전을 지정합니다. 일반적으로 서버는 특정 프로토콜 버전을 선택하고, 클라이언트는 서버의 선택에 적응해야 합니다. 대부분의 버전은 다른 버전과 상호 운용할 수 없습니다. 지정하지 않으면, 기본값은 *PROTOCOL_TLS*입니다; 다른 버전과의 호환성이 가장 뛰어납니다.

다음은 클라이언트의 어느 버전(행)이 서버의 어떤 버전(열)에 연결할 수 있는지 보여주는 표입니다:

클라이언트 / 서버	SSLv2	SSLv3	TLS[3]	TLSv1	TLSv1.1	TLSv1.2
SSLv2	yes	no	no[1]	no	no	no
SSLv3	no	yes	no[2]	no	no	no
TLS (SSLv23)[3]	no[1]	no[2]	yes	yes	yes	yes
TLSv1	no	no	yes	yes	no	no
TLSv1.1	no	no	yes	no	yes	no
TLSv1.2	no	no	yes	no	no	yes

더 보기

`create_default_context()`는 `ssl` 모듈이 주어진 목적을 위한 보안 설정을 선택할 수 있게 합니다.

버전 3.6에서 변경: The context is created with secure default values. The options `OP_NO_COMPRESSION`, `OP_CIPHER_SERVER_PREFERENCE`, `OP_SINGLE_DH_USE`, `OP_SINGLE_ECDH_USE`, `OP_NO_SSLv2`, and `OP_NO_SSLv3` (except for `PROTOCOL_SSLv3`) are set by default. The initial cipher suite list contains only `HIGH` ciphers, no `NULL` ciphers and no `MD5` ciphers.

버전 3.10부터 폐지됨: `SSLContext` without protocol argument is deprecated. The context class will either require `PROTOCOL_TLS_CLIENT` or `PROTOCOL_TLS_SERVER` protocol in the future.

버전 3.10에서 변경: The default cipher suites now include only secure AES and ChaCha20 ciphers with forward secrecy and security level 2. RSA and DH keys with less than 2048 bits and ECC keys with less than 224 bits are prohibited. `PROTOCOL_TLS`, `PROTOCOL_TLS_CLIENT`, and `PROTOCOL_TLS_SERVER` use TLS 1.2 as minimum TLS version.

참고

`SSLContext` only supports limited mutation once it has been used by a connection. Adding new certificates to the internal trust store is allowed, but changing ciphers, verification settings, or mTLS certificates may result in surprising behavior.

참고

`SSLContext` is designed to be shared and used by multiple connections. Thus, it is thread-safe as long as it is not reconfigured after being used by a connection.

`SSLContext` 객체에는 다음과 같은 메서드와 어트리뷰트가 있습니다:

`SSLContext.cert_store_stats()`

로드된 X.509 인증서 수량, CA 인증서로 표시된 X.509 인증서 및 인증서 취소 목록(CRL)의 수에 대한 통계를 딕셔너리로 가져옵니다.

하나의 CA 인증서와 다른 인증서 하나를 가진 컨텍스트 예:

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

Added in version 3.4.

[3] TLS 1.3 프로토콜은 OpenSSL >= 1.1.1에서 `PROTOCOL_TLS`로 사용할 수 있습니다. TLS 1.3만을 위한 전용 PROTOCOL 상수는 없습니다.

[1] `SSLContext`는 기본적으로 `OP_NO_SSLv2`로 SSLv2를 비활성화합니다.

[2] `SSLContext`는 기본적으로 `OP_NO_SSLv3`으로 SSLv3을 비활성화합니다.

SSLContext.**load_cert_chain**(*certfile*, *keyfile=None*, *password=None*)

Load a private key and the corresponding certificate. The *certfile* string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate's authenticity. The *keyfile* string, if present, must point to a file containing the private key. Otherwise the private key will be taken from *certfile* as well. See the discussion of 인증서 for more information on how the certificate is stored in the *certfile*.

password 인자는 개인 키의 복호화를 위한 암호를 얻기 위해 호출하는 함수가 될 수 있습니다. 개인 키가 암호화되어있고 암호가 필요한 경우에만 호출됩니다. 인자 없이 호출되며, 문자열, 바이트열 또는 bytearray를 반환해야 합니다. 반환 값이 문자열이면 키를 해독하기 전에 UTF-8로 인코딩됩니다. 또는 문자열, 바이트열 또는 bytearray 값을 *password* 인자로 직접 제공할 수 있습니다. 개인 키가 암호화되지 않고 암호가 필요 없으면 무시됩니다.

password 인자가 지정되지 않고 암호가 필요하면, OpenSSL의 기본 암호 프롬프트 메커니즘을 사용하여 대화식으로 사용자에게 암호를 묻습니다.

개인 키가 인증서와 일치하지 않으면 `SSLError`가 발생합니다.

버전 3.3에서 변경: 새로운 선택적 인자 *password*.

SSLContext.**load_default_certs**(*purpose=Purpose.SERVER_AUTH*)

Load a set of default "certification authority" (CA) certificates from default locations. On Windows it loads CA certs from the `CA` and `ROOT` system stores. On all systems it calls `SSLContext.set_default_verify_paths()`. In the future the method may load CA certificates from other locations, too.

The *purpose* flag specifies what kind of CA certificates are loaded. The default settings `Purpose.SERVER_AUTH` loads certificates, that are flagged and trusted for TLS web server authentication (client side sockets). `Purpose.CLIENT_AUTH` loads CA certificates for client certificate verification on the server side.

Added in version 3.4.

SSLContext.**load_verify_locations**(*cafile=None*, *capath=None*, *cadata=None*)

`verify_mode`가 `CERT_NONE`가 아닐 때, 다른 피어의 인증서를 확인하는 데 사용되는 "인증 기관" (CA) 인증서 집합을 로드합니다. *cafile* 나 *capath* 중 적어도 하나는 지정해야 합니다.

이 메서드는 PEM 이나 DER 형식으로 인증서 해지 목록(CRL)을 로드할 수도 있습니다. CRL을 사용하려면 `SSLContext.verify_flags`를 올바르게 구성해야 합니다.

cafile 문자열이 있으면 이어붙인 PEM 형식의 CA 인증서 파일 경로입니다. 이 파일에 인증서를 정렬하는 방법에 대한 자세한 내용은 인증서의 논의를 참조하십시오.

The *capath* string, if present, is the path to a directory containing several CA certificates in PEM format, following an OpenSSL specific layout.

cadata 객체가 있으면, 하나 이상의 PEM-인코딩된 인증서의 ASCII 문자열이거나 DER-인코딩된 인증서의 바이트열류 객체입니다. *capath*와 마찬가지로 PEM-인코딩된 인증서 주위에 추가한 줄은 무시되지만 적어도 하나의 인증서가 있어야 합니다.

버전 3.4에서 변경: 새로운 선택적 인자 *cadata*

SSLContext.**get_ca_certs**(*binary_form=False*)

로드된 "인증 기관" (CA) 인증서 목록을 가져옵니다. `binary_form` 매개 변수가 `False`면 각 리스트 항목은 `SSLSocket.getpeercert()`의 출력과 같은 딕셔너리입니다. 그렇지 않으면, 이 메서드는 DER-인코딩된 인증서의 리스트를 반환합니다. 반환된 리스트에는 인증서가 SSL 연결이 요청하고 로드되지 않는 한 *capath*의 인증서가 포함되어 있지 않습니다.

> **참고**
>
> capath 디렉터리의 인증서는 적어도 한 번 이상 사용하지 않으면 로드되지 않습니다.

Added in version 3.4.

SSLContext.**get_ciphers**()

활성화된 사이퍼의 리스트를 가져옵니다. 리스트는 사이퍼 우선순위 순입니다. *SSLContext.set_ciphers()*를 참조하십시오.

예제:

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'digest': None,
  'id': 50380848,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1.2',
  'strength_bits': 256,
  'symmetric': 'aes-256-gcm'},
 {'aead': True,
  'alg_bits': 128,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'digest': None,
  'id': 50380847,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1.2',
  'strength_bits': 128,
  'symmetric': 'aes-128-gcm'}]
```

Added in version 3.6.

SSLContext.**set_default_verify_paths**()

OpenSSL 라이브러리를 빌드할 때 정의된 파일 시스템 경로에서 기본 "인증 기관" (CA) 인증서 집합을 로드합니다. 불행히도, 이 메서드가 성공하는지를 쉽게 알 방법이 없습니다: 인증서를 찾을 수 없어도 에러가 반환되지 않습니다. 하지만 OpenSSL 라이브러리가 운영 체제 일부로 제공되면 올바르게 구성되었을 가능성이 큽니다.

SSLContext.**set_ciphers**(*ciphers*, /)

Set the available ciphers for sockets created with this context. It should be a string in the OpenSSL cipher list format. If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an *SSLError* will be raised.

> **참고**
>
> 연결될 때, SSL 소켓의 *SSLSocket.cipher()* 메서드가 현재 선택된 사이퍼를 제공합니다.
>
> TLS 1.3 cipher suites cannot be disabled with *set_ciphers()*.

SSLContext.**set_alpn_protocols**(*alpn_protocols*)

SSL/TLS 핸드 셰이크 중에 소켓이 알려야 하는 프로토콜을 지정합니다. 우선순위에 따라 정렬된 `['http/1.1', 'spdy/2']`와 같은 ASCII 문자열 리스트여야 합니다. 프로토콜 선택은 핸드 셰이크 중에 발생하며, **RFC 7301**에 따라 처리됩니다. 성공적인 핸드 셰이크가 끝나면, *SSLSocket.selected_alpn_protocol()* 메서드는 합의된 프로토콜을 반환합니다.

*HAS_NPN*이 `False`면, 이 메서드는 *NotImplementedError*를 발생시킵니다.

Added in version 3.5.

`SSLContext.`**`set_npn_protocols`**(*npn_protocols*)

SSL/TLS 핸드 셰이크 중에 소켓이 알려야 하는 프로토콜을 지정합니다. 우선순위에 따라 정렬된 `['http/1.1', 'spdy/2']`와 같은 문자열 리스트여야 합니다. 프로토콜 선택은 핸드 셰이크 중에 발생하며, Application Layer Protocol Negotiation에 따라 처리됩니다. 성공적인 핸드 셰이크가 끝나면, *SSLSocket.selected_npn_protocol()* 메서드는 합의된 프로토콜을 반환합니다.

*HAS_NPN*이 `False`면, 이 메서드는 *NotImplementedError*를 발생시킵니다.

Added in version 3.3.

버전 3.10부터 폐지됨: NPN has been superseded by ALPN

`SSLContext.`**`sni_callback`**

TLS 클라이언트가 서버 이름 표시를 지정할 때 SSL/TLS 서버에서 TLS 클라이언트 Hello 핸드 셰이크 메시지를 받은 후 호출될 콜백 함수를 등록합니다. 서버 이름 표시 메커니즘은 **RFC 6066** section 3 - Server Name Indication에서 지정됩니다.

`SSLContext` 당 하나의 콜백 만 설정할 수 있습니다. *sni_callback*이 `None`으로 설정되면 콜백이 비활성화됩니다. 이 함수를 호출하면 이전에 등록된 콜백이 비활성화됩니다.

콜백 함수는 세 개의 인자로 호출됩니다. 첫 번째는 *ssl.SSLSocket*이고, 두 번째는 클라이언트가 통신하려는 서버 이름을 나타내는 문자열(또는 TLS 클라이언트 Hello에 서버 이름이 없으면 *None*)이며, 세 번째 인자는 원래 *SSLContext*입니다. 서버 이름 인자는 텍스트입니다. 국제화된 도메인 이름의 경우, 서버 이름은 IDN A-레이블(`"xn--pythn-mua.org"`)입니다.

이 콜백은 일반적으로 *ssl.SSLSocket*의 *SSLSocket.context* 어트리뷰트를 서버 이름과 일치하는 인증서 체인을 나타내는 *SSLContext* 형의 새 객체로 변경하는 데 사용됩니다.

Due to the early negotiation phase of the TLS connection, only limited methods and attributes are usable like *SSLSocket.selected_alpn_protocol()* and *SSLSocket.context*. The *SSLSocket.getpeercert()*, *SSLSocket.get_verified_chain()*, *SSLSocket.get_unverified_chain()* *SSLSocket.cipher()* and *SSLSocket.compression()* methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not return meaningful values nor can they be called safely.

TLS 협상을 계속하려면 *sni_callback* 함수가 `None`을 반환해야 합니다. TLS 실패가 필요하면, 상수 *ALERT_DESCRIPTION_**를 반환할 수 있습니다. 다른 반환 값은 *ALERT_DESCRIPTION_INTERNAL_ERROR*로 TLS 치명적인 에러를 발생시킵니다.

sni_callback 함수에서 예외가 발생하면, TLS 연결이 치명적인 TLS 경고 메시지 *ALERT_DESCRIPTION_HANDSHAKE_FAILURE*로 종료됩니다.

이 메서드는 OpenSSL 라이브러리가 빌드될 때 OPENSSL_NO_TLSEXT가 정의되었으면 *NotImplementedError*를 발생시킵니다.

Added in version 3.7.

`SSLContext.`**`set_servername_callback`**(*server_name_callback*)

이전 버전과의 호환성을 위해 유지되는 기존 API입니다. 가능하면, 대신 *sni_callback*을 사용해야 합니다. 주어진 *server_name_callback*은 *sni_callback*과 비슷하지만, 서버 호스트 이름이 IDN-인코딩된 국제화된 도메인 이름일 때 *server_name_callback*은 디코딩된 U-레이블(`"pythön.org"`)을 받습니다.

If there is a decoding error on the server name, the TLS connection will terminate with an *ALERT_DESCRIPTION_INTERNAL_ERROR* fatal TLS alert message to the client.

Added in version 3.4.

`SSLContext.`**`load_dh_params`**(*dhfile*, /)

Diffie-Hellman (DH) 키 교환을 위한 키 생성 매개 변수를 로드합니다. DH 키 교환을 사용하면 계산 자원(서버와 클라이언트 모두)을 희생하여 FS(forward secrecy)를 향상합니다. *dhfile* 매개 변수는 PEM 형식의 DH 매개 변수를 포함하는 파일의 경로여야 합니다.

이 설정은 클라이언트 소켓에는 적용되지 않습니다. *OP_SINGLE_DH_USE* 옵션을 사용하여 보안을 더 향상할 수도 있습니다.

Added in version 3.3.

`SSLContext.`**`set_ecdh_curve`**(*curve_name*, /)

타원 곡선(Elliptic Curve) 기반 Diffie-Hellman (ECDH) 키 교환을 위한 곡선 이름을 설정합니다. 보안성에 대한 논란의 여지는 있지만 ECDH는 일반 DH보다 상당히 빠릅니다. *curve_name* 매개 변수는 잘 알려진 타원 곡선을 설명하는 문자열이어야 합니다, 예를 들어, 널리 지원되는 곡선인 `prime256v1`.

이 설정은 클라이언트 소켓에는 적용되지 않습니다. *OP_SINGLE_ECDH_USE* 옵션을 사용하여 보안을 더 향상할 수도 있습니다.

*HAS_ECDH*가 `False`면 이 메서드를 사용할 수 없습니다.

Added in version 3.3.

더 보기

SSL/TLS & Perfect Forward Secrecy
Vincent Bernat.

`SSLContext.`**`wrap_socket`**(*sock*, *server_side=False*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *server_hostname=None*, *session=None*)

Wrap an existing Python socket *sock* and return an instance of *SSLContext.sslsocket_class* (default *SSLSocket*). The returned SSL socket is tied to the context, its settings and certificates. *sock* must be a *SOCK_STREAM* socket; other socket types are unsupported.

매개 변수 `server_side`는 서버 측과 클라이언트 측 동작 중 어느 것이 소켓에서 필요한지를 식별하는 논릿값입니다.

클라이언트 측 소켓의 경우, 컨텍스트 구성이 지연됩니다; 하부 소켓이 아직 연결되어 있지 않으면, `connect()`가 소켓에서 호출된 후 컨텍스트 생성이 수행됩니다. 서버 측 소켓의 경우, 소켓에 원격 피어가 없으면, 리스닝 소켓이라고 가정하고, 서버 측 SSL 감싸기는 `accept()` 메서드를 통해 받아들인 클라이언트 연결에 대해 자동으로 수행됩니다. 메서드는 *SSLError*를 발생시킬 수 있습니다.

클라이언트 연결에서, 선택적 매개 변수 *server_hostname*는 연결하려는 서비스의 호스트 이름을 지정합니다. 이를 통해 단일 서버는 HTTP 가상 호스트와 매우 흡사하게 서로 다른 인증서로 여러 SSL 기반 서비스를 호스팅 할 수 있습니다. *server_side*가 참일 때 *server_hostname*를 지정하면 *ValueError*가 발생합니다.

매개 변수 `do_handshake_on_connect`는 `socket.connect()`를 수행한 후 SSL 핸드 셰이크를 자동으로 수행할지, 또는 응용 프로그램이 *SSLSocket.do_handshake()* 메서드를 호출하여 명시적으로 호출할지를 지정합니다. *SSLSocket.do_handshake()*를 명시적으로 호출하면, 핸드 셰이크에 수반되는 소켓 I/O의 블로킹 동작을 프로그램에서 제어할 수 있습니다.

매개 변수 `suppress_ragged_eofs`는 `SSLSocket.recv()` 메서드가 연결의 다른 끝으로부터의 예기치 않은 EOF를 알리는 방법을 지정합니다. *True*(기본값)로 지정되면, 하부 소켓에서 발생한 예기치 않은 EOF 에러에 대한 응답으로 정상 EOF(빈 바이트열 객체)를 반환합니다. *False*면 예외를 호출자에게 다시 발생시킵니다.

session, *session*을 참조하십시오.

To wrap an *SSLSocket* in another *SSLSocket*, use *SSLContext.wrap_bio()*.

버전 3.5에서 변경: OpenSSL에 SNI가 없더라도 항상 server_hostname을 전달할 수 있습니다.

버전 3.6에서 변경: *session* 인자가 추가되었습니다.

버전 3.7에서 변경: The method returns an instance of *SSLContext.sslsocket_class* instead of hard-coded *SSLSocket*.

SSLContext.**sslsocket_class**

The return type of *SSLContext.wrap_socket()*, defaults to *SSLSocket*. The attribute can be assigned to on instances of *SSLContext* in order to return a custom subclass of *SSLSocket*.

Added in version 3.7.

SSLContext.**wrap_bio**(*incoming*, *outgoing*, *server_side=False*, *server_hostname=None*, *session=None*)

BIO 객체 *incoming* 과 *outgoing*을 감싸고 *SSLContext.sslobject_class*(기본값 *SSLObject*)의 인스턴스를 반환합니다. SSL 루틴은 incoming BIO에서 입력 데이터를 읽고 outgoing BIO에 데이터를 씁니다.

server_side, *server_hostname* 및 *session* 매개 변수는 *SSLContext.wrap_socket()*에서와 같은 의미입니다.

버전 3.6에서 변경: *session* 인자가 추가되었습니다.

버전 3.7에서 변경: The method returns an instance of *SSLContext.sslobject_class* instead of hard-coded *SSLObject*.

SSLContext.**sslobject_class**

*SSLContext.wrap_bio()*의 반환형, 기본값은 *SSLObject*입니다. 이 어트리뷰트는 *SSLObject*의 사용자 정의 서브 클래스를 반환하기 위해 클래스의 인스턴스에서 재정의될 수 있습니다.

Added in version 3.7.

SSLContext.**session_stats**()

Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each piece of information to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

SSLContext.**check_hostname**

*SSLSocket.do_handshake()*에서 피어 인증서의 호스트 이름을 일치시킬지 여부. 컨텍스트의 *verify_mode*는 *CERT_OPTIONAL*나 *CERT_REQUIRED*로 설정되어야 하며, 호스트 이름을 일치시키려면 *server_hostname*을 *wrap_socket()*로 전달해야 합니다. 호스트 이름 확인을 활성화하면 *verify_mode*가 *CERT_NONE*에서 *CERT_REQUIRED*로 자동 설정됩니다. 호스트 이름 검사가 활성화되어 있으면 *CERT_NONE*로 다시 설정할 수 없습니다. *PROTOCOL_TLS_CLIENT* 프로토콜은 기본적으로 호스트 이름 확인을 활성화합니다. 다른 프로토콜의 경우, 호스트 이름 확인을 명시적으로 활성화해야 합니다.

예제:

```
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

Added in version 3.4.

버전 3.7에서 변경: 호스트 이름 검사가 활성화되고 *verify_mode*가 *CERT_NONE*이면 이제 *verify_mode*가 *CERT_REQUIRED*로 자동 변경됩니다. 이전에는 같은 작업이 *ValueError*로 실패했을 것입니다.

SSLContext.**keylog_filename**

키 자료가 생성되거나 수신될 때마다, TLS 키를 키로그(keylog) 파일에 기록합니다. 키로그 파일은 디버깅 목적으로만 설계되었습니다. 파일 형식은 NSS에 의해 지정되었고 Wireshark과 같은 많은 트래픽 분석기에서 사용됩니다. 로그 파일은 덧붙이기 전용 모드로 열립니다. 쓰기는 스레드 간에 동기화되지만, 프로세스 간에는 동기화되지 않습니다.

Added in version 3.8.

SSLContext.**maximum_version**

A *TLSVersion* enum member representing the highest supported TLS version. The value defaults to *TLSVersion.MAXIMUM_SUPPORTED*. The attribute is read-only for protocols other than *PROTOCOL_TLS*, *PROTOCOL_TLS_CLIENT*, and *PROTOCOL_TLS_SERVER*.

어트리뷰트 *maximum_version*, *minimum_version* 및 *SSLContext.options*는 모두 컨텍스트의 지원되는 SSL과 TLS 버전에 영향을 줍니다. 구현은 부적합한 조합을 방지하지 못합니다. 예를 들어, *options*에 *OP_NO_TLSv1_2*가 있고 *maximum_version*이 *TLSVersion.TLSv1_2*로 설정된 컨텍스트는 TLS 1.2 연결을 이룰 수 없습니다.

Added in version 3.7.

SSLContext.**minimum_version**

가장 낮은 지원 버전 또는 *TLSVersion.MINIMUM_SUPPORTED*이라는 것만 제외하면 *SSLContext.maximum_version*과 같습니다.

Added in version 3.7.

SSLContext.**num_tickets**

Control the number of TLS 1.3 session tickets of a *PROTOCOL_TLS_SERVER* context. The setting has no impact on TLS 1.0 to 1.2 connections.

Added in version 3.8.

SSLContext.**options**

이 컨텍스트에서 활성화된 SSL 옵션 집합을 나타내는 정수. 기본값은 *OP_ALL*이지만, *OP_NO_SSLv2*와 같은 다른 옵션을 함께 OR로 연결하여 지정할 수 있습니다.

버전 3.6에서 변경: *SSLContext.options*는 *Options* 플래그를 반환합니다.:

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

버전 3.7부터 폐지됨: All `OP_NO_SSL*` and `OP_NO_TLS*` options have been deprecated since Python 3.7. Use *SSLContext.minimum_version* and *SSLContext.maximum_version* instead.

SSLContext.**post_handshake_auth**

TLS 1.3 포스트 핸드 셰이크 클라이언트 인증을 사용합니다. 포스트 핸드 셰이크 인증은 기본적으로 사용되지 않으며 서버는 초기 핸드 셰이크 중에 TLS 클라이언트 인증서만 요청할 수 있습니다. 활성화되면, 서버는 핸드 셰이크 후에 언제든지 TLS 클라이언트 인증서를 요청할 수 있습니다.

클라이언트 측 소켓에서 활성화될 때, 클라이언트는 포스트 핸드 셰이크 인증을 지원하는 서버에 신호를 보냅니다.

서버 측 소켓에서 활성화될 때, *SSLContext.verify_mode*도 *CERT_OPTIONAL*이나 *CERT_REQUIRED*로 설정해야 합니다. 실제 클라이언트 인증서 교환은 *SSLSocket.verify_client_post_handshake()*가 호출되고 일부 I/O가 수행될 때까지 지연됩니다.

Added in version 3.8.

SSLContext.**protocol**

컨텍스트를 구성할 때 선택한 프로토콜 버전. 이 어트리뷰트는 읽기 전용입니다.

SSLContext.**hostname_checks_common_name**

*check_hostname*가 SAN(subject alternative name) 확장이 없을 때 인증서의 SCN(subject common name)을 유효성 검사하는 것으로 폴백 할지 아닐지 (기본값: 참)

Added in version 3.7.

버전 3.10에서 변경: The flag had no effect with OpenSSL before version 1.1.1l. Python 3.8.9, 3.9.3, and 3.10 include workarounds for previous versions.

`SSLContext.`**`security_level`**

An integer representing the security level for the context. This attribute is read-only.

Added in version 3.10.

`SSLContext.`**`verify_flags`**

The flags for certificate verification operations. You can set flags like *VERIFY_CRL_CHECK_LEAF* by ORing them together. By default OpenSSL does neither require nor verify certificate revocation lists (CRLs).

Added in version 3.4.

버전 3.6에서 변경: *SSLContext.verify_flags*는 *VerifyFlags* 플래그를 반환합니다.:

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

`SSLContext.`**`verify_mode`**

다른 피어의 인증서를 확인할지와 확인이 실패할 때 어떻게 해야 하는지를 나타냅니다. 이 어트리뷰트는 *CERT_NONE*, *CERT_OPTIONAL* 또는 *CERT_REQUIRED* 중 하나여야 합니다.

버전 3.6에서 변경: *SSLContext.verify_mode*는 *VerifyMode* 열거형을 반환합니다.:

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

`SSLContext.`**`set_psk_client_callback`**(*callback*)

Enables TLS-PSK (pre-shared key) authentication on a client-side connection.

In general, certificate based authentication should be preferred over this method.

The parameter `callback` is a callable object with the signature: `def callback(hint: str | None) -> tuple[str | None, bytes]`. The `hint` parameter is an optional identity hint sent by the server. The return value is a tuple in the form (client-identity, psk). Client-identity is an optional string which may be used by the server to select a corresponding PSK for the client. The string must be less than or equal to `256` octets when UTF-8 encoded. PSK is a *bytes-like object* representing the pre-shared key. Return a zero length PSK to reject the connection.

Setting `callback` to *None* removes any existing callback.

> **참고**
>
> When using TLS 1.3:
>
> - the `hint` parameter is always *None*.
> - client-identity must be a non-empty string.

Example usage:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.check_hostname = False
context.verify_mode = ssl.CERT_NONE
context.maximum_version = ssl.TLSVersion.TLSv1_2
context.set_ciphers('PSK')

# A simple lambda:
psk = bytes.fromhex('c0ffee')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
context.set_psk_client_callback(lambda hint: (None, psk))

# A table using the hint from the server:
psk_table = { 'ServerId_1': bytes.fromhex('c0ffee'),
              'ServerId_2': bytes.fromhex('facade')
}
def callback(hint):
    return 'ClientId_1', psk_table.get(hint, b'')
context.set_psk_client_callback(callback)
```

This method will raise *NotImplementedError* if *HAS_PSK* is `False`.

Added in version 3.13.

`SSLContext.`**`set_psk_server_callback`**(*callback*, *identity_hint=None*)

Enables TLS-PSK (pre-shared key) authentication on a server-side connection.

In general, certificate based authentication should be preferred over this method.

The parameter `callback` is a callable object with the signature: `def callback(identity: str | None) -> bytes`. The `identity` parameter is an optional identity sent by the client which can be used to select a corresponding PSK. The return value is a *bytes-like object* representing the pre-shared key. Return a zero length PSK to reject the connection.

Setting `callback` to *None* removes any existing callback.

The parameter `identity_hint` is an optional identity hint string sent to the client. The string must be less than or equal to `256` octets when UTF-8 encoded.

> **참고**
>
> When using TLS 1.3 the `identity_hint` parameter is not sent to the client.

Example usage:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.maximum_version = ssl.TLSVersion.TLSv1_2
context.set_ciphers('PSK')

# A simple lambda:
psk = bytes.fromhex('c0ffee')
context.set_psk_server_callback(lambda identity: psk)

# A table using the identity of the client:
psk_table = { 'ClientId_1': bytes.fromhex('c0ffee'),
              'ClientId_2': bytes.fromhex('facade')
}
def callback(identity):
    return psk_table.get(identity, b'')
context.set_psk_server_callback(callback, 'ServerId_1')
```

This method will raise *NotImplementedError* if *HAS_PSK* is `False`.

Added in version 3.13.

19.3.4 인증서

인증서는 일반적으로 공개키/개인키 시스템 일부입니다. 이 시스템에서, 각 주체(*principal*)(시스템, 사람 또는 조직일 수 있습니다)에게는 고유한 두 부분으로 된 암호화 키가 지정됩니다. 열쇠의 한 부분은 공개(public)며, 공개키(*public key*)라고 불립니다; 다른 부분은 비밀로 유지되며, 개인키(*private key*)라고 합니다. 두 부분은 관련이 있습니다. 두 부분 중 하나를 사용하여 메시지를 암호화하면, 다른 부분으로 해독할 수 있고, **오직** 다른 부분으로만 해독할 수 있습니다.

인증서에는 두 주체에 대한 정보가 들어 있습니다. 주체(*subject*)의 이름과 주체의 공개키가 들어 있습니다. 또한 발행자(*issuer*)라는 두 번째 주체의 진술을 포함하는데, 해당 주체(subject)는 자신이 주장하는 존재며, 실제로 공개키 또한 주체(subject)의 것이 맞는다는 내용입니다. 발행자의 진술은 발행자만이 알고 있는 발행자의 개인키로 서명됩니다. 그러나 누구든지 발행자의 공개키를 찾고 이를 사용하여 진술을 해독하고 인증서의 다른 정보와 비교함으로써 발행자의 진술을 확인할 수 있습니다. 또한, 인증서에는 유효 기간에 대한 정보가 들어 있습니다. 이것은 "notBefore" 와 "notAfter" 라고 하는 두 개의 필드로 표현됩니다.

파이썬에서 인증서를 사용할 때, 클라이언트나 서버는 인증서를 사용하여 자신이 누구인지 증명할 수 있습니다. 네트워크 연결의 다른 쪽은 인증서 생성을 요구받을 수도 있으며, 해당 인증서는 이러한 유효성 검사가 필요한 클라이언트나 서버를 만족하도록 유효성을 검사할 수 있습니다. 유효성 검증이 실패하면 연결 시도가 예외를 발생시키도록 설정할 수 있습니다. 유효성 검사는 하부 OpenSSL 프레임워크에 의해 자동으로 수행됩니다; 응용 프로그램은 그 메커니즘에 관심을 두지 않아도 됩니다. 그러나 응용 프로그램은 일반적으로 이 절차를 수행할 수 있도록 인증서 집합을 제공해야 합니다.

파이썬은 인증서를 포함하기 위해 파일을 사용합니다. 그들은 "PEM" (**RFC 1422**를 참조하세요)으로 포맷해야 합니다. 머릿줄과 꼬리 줄로 감싼 base-64 로 인코딩된 형식입니다:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

인증서 체인

인증서를 포함하는 파이썬 파일에는 인증서 시퀀스가 포함될 수 있는데, 때로 인증서 체인(**certificate chain*)이라고 부릅니다. 이 체인은 클라이언트 또는 서버 "당사자" 주체에 대한 특정 인증서로 시작해야 하며, 그다음에 그 인증서의 발행자에 대한 인증서가 오고, 그다음에 직전 인증서 발행자에 대한 인증서가 오는 식으로 이어지다가, 결국에는 자체 서명(*self-signed*) 인증서를 얻게 되는데, 주체와 발행자가 같은 인증서로 때로 루트 인증서라고도 부릅니다. 인증서는 인증서 파일에 함께 이어붙여야 합니다. 예를 들어, 서버 인증서에서 서버 인증서에 서명한 인증 기관의 인증서를 거쳐 인증 기관의 인증서를 발행한 기관의 루트 인증서에 이르는 세 개의 인증서 체인이 있다고 가정합시다:

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA 인증서

연결의 반대편의 인증서의 유효성 검사가 필요하면, 신뢰할 수 있는 각 발행자의 인증서 체인으로 채워진 "CA 인증서" 파일을 제공해야 합니다. 다시 말하지만, 이 파일은 단지 이러한 체인들을 함께 이어붙인 것입니다. 유효성 검사를 위해, 파이썬은 일치하는 파일에서 찾은 첫 번째 체인을 사용합니다. 플랫폼의 인증서 파일은 `SSLContext.load_default_certs()`를 호출하여 사용할 수 있습니다, 이는 `create_default_context()`로 자동으로 수행됩니다.

결합한 키와 인증서

Often the private key is stored in the same file as the certificate; in this case, only the `certfile` parameter to `SSLContext.load_cert_chain()` needs to be passed. If the private key is stored with the certificate, it should come before the first certificate in the certificate chain:

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

자체 서명 인증서

SSL-암호화된 연결 서비스를 제공하는 서버를 만들려면, 해당 서비스에 대한 인증서를 얻어야 합니다. 인증 기관에서 사는 등 다양한 방법으로 적절한 인증서를 얻을 수 있습니다. 또 다른 일반적인 관행은 자체 서명 인증서를 생성하는 것입니다. 이렇게 하는 가장 간단한 방법은 OpenSSL 패키지에서 다음과 같은 방법을 사용하는 것입니다:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.......++++++
.............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

자체 서명 인증서의 단점은 그 자신이 루트 인증서이고, 아무도 그들의 알려진(그리고 신뢰할 수 있는) 루트 인증서의 캐시에 이 인증서를 갖고 있지 않다는 것입니다.

19.3.5 예제

SSL 지원 검사하기

파이썬 설치에 SSL 지원이 있는지를 검사하려면, 사용자 코드는 다음과 같은 관용구를 사용해야 합니다:

```
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

클라이언트 측 연산

이 예제는 자동 인증서 확인을 포함하여 클라이언트 소켓에 대해 권장되는 보안 설정을 사용하여 SSL 컨텍스트를 만듭니다:

```
>>> context = ssl.create_default_context()
```

보안 설정을 직접 조정하려면, 처음부터 컨텍스트를 만들 수 있습니다 (그러나 올바른 설정을 얻지 못할 수도 있습니다):

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(이 코드 조각은 운영 체제가 모든 CA 인증서 번들을 `/etc/ssl/certs/ca-bundle.crt` 에 배치한다고 가정합니다; 그렇지 않으면, 에러가 발생하고 위치를 조정해야 합니다)

PROTOCOL_TLS_CLIENT 프로토콜은 인증서 유효성 검사와 호스트 이름 확인을 위한 컨텍스트를 구성합니다. *verify_mode*는 *CERT_REQUIRED*로 설정되고 *check_hostname*는 `True`로 설정됩니다. 다른 모든 프로토콜은 안전하지 않은 기본값으로 SSL 컨텍스트를 만듭니다.

컨텍스트를 사용하여 서버에 연결할 때, *CERT_REQUIRED*와 *check_hostname*은 서버 인증서의 유효성을 검사합니다: 서버 인증서가 CA 인증서 중 하나를 사용하여 서명되었는지 확인하고, 서명의 정확성을 검사하고, 호스트 이름의 유효성과 아이덴티티와 같은 다른 속성을 확인합니다:

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

그런 다음 인증서를 가져올 수 있습니다:

```
>>> cert = conn.getpeercert()
```

시각적인 검사는 인증서가 원하는 서비스(즉, HTTPS 호스트 `www.python.org`)를 식별함을 보여줍니다:

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.
↪crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('organizationalUnitName', 'www.digicert.com'),),
            (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': ((('businessCategory', 'Private Organization'),),
             (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
             (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
             (('serialNumber', '3359300'),),
             (('streetAddress', '16 Allen Rd'),),
             (('postalCode', '03894-4801'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'NH'),),
             (('localityName', 'Wolfeboro'),),
             (('organizationName', 'Python Software Foundation'),),
             (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
                    ('DNS', 'id.python.org')),
 'version': 3}
```

이제 SSL 채널이 설정되고 인증서가 확인되었습니다, 서버와 대화할 수 있습니다:

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'',
 b'']
```

아래의 보안 고려 사항의 논의를 참조하십시오.

서버 측 연산

서버 연산의 경우, 일반적으로 서버 인증서와 개인 키가 각각 파일에 있어야 합니다. 먼저 클라이언트가 여러분의 신원을 확인할 수 있도록 키와 인증서가 있는 컨텍스트를 만듭니다. 그런 다음 소켓을 열고, 포트에 바인드 하고, `listen()` 을 호출한 다음 클라이언트가 연결하기를 기다립니다:

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.example.com', 10023))
bindsocket.listen(5)
```

클라이언트가 연결하면, 소켓에서 `accept()` 를 호출하여 다른 쪽 끝과 연결된 새 소켓을 얻고, 컨텍스트의 *SSLContext.wrap_socket()* 메서드를 사용하여 연결을 위한 서버 측 SSL 소켓을 만듭니다.:

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

그런 다음 connstream에서 데이터를 읽고 클라이언트와 작업을 마칠 때까지 (또는 클라이언트가 마칠 때까지) 뭔가 합니다:

```
def deal_with_client(connstream):
    data = connstream.recv(1024)
    # empty data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.recv(1024)
    # finished with client
```

그리고는 새로운 클라이언트 연결을 리스닝하는 것으로 돌아갑니다 (물론, 실제 서버는 별도의 스레드에서 각 클라이언트 연결을 처리하거나 소켓을 비 블로킹 모드로 만들고 이벤트 루프를 사용합니다).

19.3.6 비 블로킹 소켓에 대한 참고 사항

SSL 소켓은 비 블로킹 모드에서 일반 소켓과 약간 다르게 작동합니다. 비 블로킹 소켓으로 작업할 때 주의해야 할 몇 가지 사항이 있습니다:

- 대부분 *SSLSocket* 메서드는 I/O 연산이 블록하려고 할 때 *BlockingIOError* 대신 *SSLWantWriteError* 나 *SSLWantReadError*를 발생시킵니다. 하부 소켓에서의 읽기 연산이 필요하면 *SSLWantReadError*가 발생하고, 하부 소켓에서의 쓰기 연산이 필요하면 *SSLWantWriteError*가 발생합니다. SSL 소켓에 대한 쓰기 시도는 우선 하부 소켓에서 읽기가 필요할 수 있으며, SSL 소켓에서 읽기를 시도하면 하부 소켓에서 선행 쓰기가 필요할 수 있습니다.

  버전 3.5에서 변경: 이전 파이썬 버전에서는, SSLSocket.send() 메서드가 *SSLWantWriteError* 나 *SSLWantReadError*를 발생시키는 대신 0을 반환했습니다.

- *select()*를 호출하면 OS-수준 소켓을 읽을 수 있음을 (또는 쓸 수 있음을) 알려줄 수 있습니다만, 이것이 상위 SSL 계층에 충분한 데이터가 있음을 의미하지는 않습니다. 예를 들어, SSL 프레임의 일부만 도착했을 수 있습니다. 따라서, SSLSocket.recv() 와 SSLSocket.send() 실패를 처리할 준비가 되어 있어야 하며, *select()*를 다시 호출한 후 재시도해야 합니다.

- 반대로, SSL 계층에는 자체 프레임이 있으므로, SSL 소켓에는 *select()*가 인식하지 못하더라도 읽을 수 있는 데이터가 있을 수 있습니다. 따라서, 먼저 SSLSocket.recv()를 호출하여 잠재적으로 사용 가능한 모든 데이터를 꺼낸 다음, 여전히 필요할 때만 *select()* 호출에 블록해야 합니다.

  (물론, *poll()* 이나 *selectors* 모듈에 있는 것과 같은 다른 프리미티브를 사용할 때도 비슷한 조항이 적용됩니다)

- SSL 핸드 셰이크 자체는 비 블로킹입니다: *SSLSocket.do_handshake()* 메서드는 성공적으로 반환될 때까지 재시도해야 합니다. 다음은 *select()*를 사용하여 소켓의 준비 상태를 기다리는 개요입니다:

  ```
  while True:
      try:
          sock.do_handshake()
          break
  ```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

더 보기

The `asyncio` module supports *non-blocking SSL sockets* and provides a higher level *Streams API*. It polls for events using the `selectors` module and handles `SSLWantWriteError`, `SSLWantReadError` and `BlockingIOError` exceptions. It runs the SSL handshake asynchronously as well.

19.3.7 메모리 BIO 지원

Added in version 3.5.

SSL 모듈이 파이썬 2.6에서 소개된 이래로, `SSLSocket` 클래스는 관련성이 있지만, 별개의 두 기능 영역을 제공했습니다:

- SSL 프로토콜 처리
- 네트워크 IO

네트워크 IO API는 `socket.socket`가 제공하는 것과 같으며, `SSLSocket`는 그 것을 상속합니다. 이렇게 해서 SSL 소켓을 일반 소켓의 드롭 인 대체품으로 사용할 수 있으므로, 기존 응용 프로그램에 SSL 지원을 쉽게 추가 할 수 있습니다.

SSL 프로토콜 처리와 네트워크 IO의 결합은 일반적으로 잘 작동하지만, 그렇지 않은 경우도 있습니다. `socket.socket` 과 내부 OpenSSL 소켓 IO 루틴이 가정하는 "파일 기술자에 대한 select/poll" (준비 상태 기반) 모델과 다른 IO 멀티플렉싱 모델을 사용하려는 비동기 IO 프레임워크가 그 예입니다. 이것은 주로 이 모델이 효율적이지 않은 윈도우와 같은 플랫폼과 관련이 있습니다. 이를 위해, `SSLObject`라는 `SSLSocket`의 축소된 범위 변형이 제공됩니다.

class ssl.**SSLObject**

네트워크 IO 메서드를 포함하지 않는 SSL 프로토콜 인스턴스를 나타내는 `SSLSocket`의 축소 범위 변형입니다. 이 클래스는 일반적으로 메모리 버퍼를 통해 SSL 용 비동기 IO를 구현하려는 프레임워크 작성자가 사용합니다.

이 클래스는 OpenSSL에 의해 구현된 저수준 SSL 객체 위에 인터페이스를 구현합니다. 이 객체는 SSL 연결의 상태를 캡처하지만, 네트워크 IO 자체를 제공하지는 않습니다. IO는 OpenSSL의 IO 추상화 계층인 별도의 "BIO" 객체를 통해 수행되어야 합니다.

이 클래스에는 공개된 생성자가 없습니다. `SSLObject` 인스턴스는 `wrap_bio()` 메서드를 사용해서 만들어야 합니다. 이 메서드는 `SSLObject` 인스턴스를 생성하고 BIO 쌍에 연결합니다. *incoming* BIO는 파이썬에서 SSL 프로토콜 인스턴스로 데이터를 전달하는 데 사용되는 반면, *outgoing* BIO는 반대 방향으로 데이터를 전달하는 데 사용됩니다.

다음 메서드를 사용할 수 있습니다:

- `context`
- `server_side`
- `server_hostname`
- `session`
- `session_reused`
- `read()`
- `write()`
- `getpeercert()`

- *get_verified_chain()*
- *get_unverified_chain()*
- *selected_alpn_protocol()*
- *selected_npn_protocol()*
- *cipher()*
- *shared_ciphers()*
- *compression()*
- *pending()*
- *do_handshake()*
- *verify_client_post_handshake()*
- *unwrap()*
- *get_channel_binding()*
- *version()*

*SSLSocket*와 비교할 때, 이 객체에는 다음과 같은 기능이 없습니다:

- 모든 형태의 네트워크 IO; `recv()` 와 `send()` 는 하부 *MemoryBIO* 버퍼만 읽고 씁니다.
- *do_handshake_on_connect* 기작이 없습니다. 핸드 셰이크를 시작하려면 항상 *do_handshake()* 를 수동으로 호출해야 합니다.
- *suppress_ragged_eofs*는 처리되지 않습니다. 프로토콜을 위반하는 모든 파일 끝(end-of-file) 조건은 *SSLEOFError* 예외를 통해 보고됩니다.
- 메서드 *unwrap()* 호출은 하부 소켓을 반환하는 SSL 소켓과 달리 아무것도 반환하지 않습니다.
- *SSLContext.set_servername_callback()* 에 전달된 *server_name_callback* 콜백은 첫 번째 매개 변수로 *SSLSocket* 인스턴스 대신 *SSLObject* 인스턴스를 받습니다.

SSLObject 사용과 관련된 몇 가지 참고 사항:

- *SSLObject*의 모든 IO는 비 블로킹입니다. 이것은 예를 들어 *read()*는 incoming BIO에 있는 것보다 많은 데이터가 필요하면 *SSLWantReadError*를 발생시킨다는 것을 의미합니다.

버전 3.7에서 변경: *SSLObject* instances must be created with *wrap_bio()*. In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

SSLObject는 메모리 버퍼를 사용하여 바깥세상과 통신합니다. *MemoryBIO* 클래스는 이 목적으로 사용할 수 있는 메모리 버퍼를 제공합니다. OpenSSL 메모리 BIO (Basic IO) 객체를 감쌉니다:

class ssl.**MemoryBIO**

파이썬과 SSL 프로토콜 인스턴스 간에 데이터를 전달하는 데 사용할 수 있는 메모리 버퍼.

pending

현재 메모리 버퍼에 있는 바이트의 수를 반환합니다.

eof

메모리 BIO가 현재 EOF(end-of-file) 위치에 있는지를 나타내는 논릿값입니다.

read(*n=-1*, /)

메모리 버퍼에서 최대 *n* 바이트를 읽습니다. *n*이 지정되지 않았거나 음수면, 모든 바이트가 반환됩니다.

write(*buf*, /)

*buf*의 바이트를 메모리 BIO에 씁니다. *buf* 인자는 버퍼 프로토콜을 지원하는 객체여야 합니다.

반환 값은 기록된 바이트 수인데, 항상 *buf*의 길이와 같습니다.

write_eof()

EOF 마커를 메모리 BIO에 씁니다. 이 메서드가 호출된 후, *write()*를 호출하는 것은 불법입니다. *eof* 어트리뷰트는 현재 버퍼에 있는 모든 데이터를 읽은 후에 참이 됩니다.

19.3.8 SSL 세션

Added in version 3.6.

class ssl.**SSLSession**

*session*에서 사용되는 세션 객체.

id

time

timeout

ticket_lifetime_hint

has_ticket

19.3.9 보안 고려 사항

가장 좋은 기본값

클라이언트의 경우, 보안 정책에 대한 특별한 요구 사항이 없으면, *create_default_context()* 함수를 사용하여 SSL 컨텍스트를 만드는 것이 좋습니다. 시스템의 신뢰할 수 있는 CA 인증서를 로드하고, 인증서 유효성 검사와 호스트 이름 검사를 활성화하고, 합리적으로 안전한 프로토콜과 사이퍼 설정을 선택합니다.

예를 들어, 다음은 *smtplib.SMTP* 클래스를 사용하여 SMTP 서버에 대한 신뢰할 수 있고 안전한 연결을 만드는 방법입니다:

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

연결에 클라이언트 인증서가 필요하면, *SSLContext.load_cert_chain()*으로 추가할 수 있습니다.

대조적으로, *SSLContext* 생성자를 직접 호출하여 SSL 컨텍스트를 만들면, 기본적으로 인증서 유효성 검사나 호스트 이름 확인이 활성화되지 않습니다. 그렇게 하면, 아래 단락을 읽고 적절한 보안 수준을 달성하십시오.

수동 설정

인증서 확인

When calling the *SSLContext* constructor directly, *CERT_NONE* is the default. Since it does not authenticate the other peer, it can be insecure, especially in client mode where most of the time you would like to ensure the authenticity of the server you' re talking to. Therefore, when in client mode, it is highly recommended to use *CERT_REQUIRED*. However, it is in itself not sufficient; you also have to check that the server certificate, which can be obtained by calling *SSLSocket.getpeercert()*, matches the desired service. For many protocols and applications, the service can be identified by the hostname. This common check is automatically performed when *SSLContext.check_hostname* is enabled.

버전 3.7에서 변경: Hostname matchings is now performed by OpenSSL. Python no longer uses `match_hostname()`.

서버 모드에서, (고수준 인증 메커니즘을 사용하는 대신) SSL 계층을 사용하여 클라이언트를 인증하려면, *CERT_REQUIRED*를 지정하고 클라이언트 인증서도 비슷하게 확인해야 합니다.

프로토콜 버전

SSL 버전 2와 3은 안전하지 않은 것으로 간주하므로 사용하기에 위험합니다. 클라이언트와 서버 간에 최대한의 호환성을 원하면 프로토콜 버전으로 *PROTOCOL_TLS_CLIENT* 나 *PROTOCOL_TLS_SERVER*를 사용하는 것이 좋습니다. SSLv2 및 SSLv3은 기본적으로 비활성화되어 있습니다.

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.minimum_version = ssl.TLSVersion.TLSv1_3
>>> client_context.maximum_version = ssl.TLSVersion.TLSv1_3
```

The SSL context created above will only allow TLSv1.3 and later (if supported by your system) connections to a server. *PROTOCOL_TLS_CLIENT* implies certificate validation and hostname checks by default. You have to load certificates into the context.

사이퍼 선택

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the *SSLContext.set_ciphers()* method. Starting from Python 3.2.3, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. Be sure to read OpenSSL' s documentation about the cipher list format. If you want to check which ciphers are enabled by a given cipher list, use *SSLContext.get_ciphers()* or the `openssl ciphers` command on your system.

다중 프로세싱

If using this module as part of a multi-processed application (using, for example the *multiprocessing* or *concurrent.futures* modules), be aware that OpenSSL' s internal random number generator does not properly handle forked processes. Applications must change the PRNG state of the parent process if they use any SSL feature with *os.fork()*. Any successful call of *RAND_add()* or *RAND_bytes()* is sufficient.

19.3.10 TLS 1.3

Added in version 3.7.

The TLS 1.3 protocol behaves slightly differently than previous version of TLS/SSL. Some new TLS 1.3 features are not yet available.

- TLS 1.3은 분리된 사이퍼 스위트 집합을 사용합니다. 모든 AES-GCM과 ChaCha20 사이퍼 스위트는 기본적으로 활성화되어 있습니다. *SSLContext.set_ciphers()* 메서드는 아직 TLS 1.3 사이퍼를 활성화하거나 비활성화할 수 없지만, *SSLContext.get_ciphers()*는 이들을 반환합니다.
- 세션 티켓은 더는 초기 핸드 셰이크의 일부로 전송되지 않고 다르게 처리됩니다. *SSLSocket.session* 과 *SSLSession*은 TLS 1.3과 호환되지 않습니다.
- 클라이언트 측 인증서도 더는 초기 핸드 셰이크 중에 검증되지 않습니다. 서버는 언제든지 인증서를 요청할 수 있습니다. 클라이언트는 서버와 응용 프로그램 데이터를 주고받는 동안 인증서 요청을 처리합니다.
- 초기 데이터(early data), 지연된 TLS 클라이언트 인증서 요청, 서명 알고리즘 구성 및 rekeying과 같은 TLS 1.3 기능은 아직 지원되지 않습니다.

더 보기

***socket.socket* 클래스**
하부 *socket* 클래스의 설명서

SSL/TLS Strong Encryption: An Introduction
Apache HTTP 서버 설명서의 개요

RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management
Steve Kent

RFC 4086: Randomness Requirements for Security
Donald E., Jeffrey I. Schiller

RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile
D. Cooper

RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2
T. Dierks et. al.

RFC 6066: Transport Layer Security (TLS) Extensions
D. Eastlake

IANA TLS: Transport Layer Security (TLS) Parameters
IANA

RFC 7525: Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)
IETF

모질라의 서버 측 TLS 추천
Mozilla

19.4 select — I/O 완료 대기

이 모듈은 대부분의 운영 체제에서 사용 가능한 `select()` 와 `poll()` 함수, Solaris와 파생 제품에서 사용 가능한 `devpoll()`, 리눅스 2.5+ 에서 사용 가능한 `epoll()` 및 대부분의 BSD에서 사용 가능한 `kqueue()` 에 대한 액세스를 제공합니다. 윈도우에서는 소켓에서만 작동함에 유의하십시오; 다른 운영 체제에서는, 다른 파일 유형에서도 작동합니다 (특히, 유닉스에서는 파이프에서도 작동합니다). 일반 파일에서 파일을 마지막으로 읽은 이후에 파일이 커졌는지를 판별하기 위해 사용할 수는 없습니다.

> **참고**
>
> *selectors* 모듈은 *select* 모듈 프리미티브에 기반한 고수준의 효율적인 I/O 멀티플렉싱을 제공합니다. 사용되는 OS 수준 프리미티브에 대한 정확한 제어를 원하지 않는 한, 사용자는 *selectors* 모듈을 대신 사용하는 것이 좋습니다.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

모듈은 다음을 정의합니다:

exception `select.`**error**

*OSError*의 폐지된 별칭.

버전 3.3에서 변경: **PEP 3151**에 따라, 이 클래스는 *OSError*의 별칭이 되었습니다.

`select.`**devpoll**`()`

(Solaris와 파생 제품에서만 지원됩니다.) `/dev/poll` 폴링(polling) 객체를 반환합니다; devpoll 객체가 지원하는 메서드에 대해서는 아래의 */dev/poll 폴링 객체* 섹션을 참조하십시오.

`devpoll()` 객체는 인스턴스 화 시점에 허용되는 파일 기술자 수에 연결됩니다. 프로그램이 이 값을 줄이면, `devpoll()` 이 실패합니다. 프로그램이 이 값을 늘리면, `devpoll()` 은 불완전한 활성 파일 기술자 리스트를 반환할 수 있습니다.

새 파일 기술자는 상속 불가능합니다.

Added in version 3.3.

버전 3.4에서 변경: 새 파일 기술자는 이제 상속 불가능합니다.

select.**epoll**(*sizehint=-1*, *flags=0*)

(리눅스 2.5.44 이상에서만 지원됩니다.) I/O 이벤트를 위한 에지(Edge)나 레벨(Level) 트리거 되는 인터페이스로 사용할 수 있는 에지 폴링 객체를 반환합니다.

*sizehint*는 epoll에 등록될 예상 이벤트 수를 알려줍니다. 양수이거나, 기본값을 사용하려면 `-1`이어야 합니다. `epoll_create1()`을 사용할 수 없는 구형 시스템에서만 사용됩니다; 그렇지 않으면 효과가 없습니다 (값을 여전히 확인하기는 합니다).

*flags*는 폐지되었고 완전히 무시됩니다. 그러나, 제공되면, 값은 `0`이나 `select.EPOLL_CLOEXEC`여야 합니다. 그렇지 않으면 `OSError`가 발생합니다.

epolling 객체가 지원하는 메서드는 아래의 에지와 레벨 트리거 폴링 (*epoll*) 객체 섹션을 참조하십시오.

`epoll` 객체는 컨텍스트 관리자 프로토콜을 지원합니다: `with` 문에서 사용될 때, 새 파일 기술자는 블록 끝에서 자동으로 닫힙니다.

새 파일 기술자는 상속 불가능합니다.

버전 3.3에서 변경: *flags* 매개 변수를 추가했습니다.

버전 3.4에서 변경: `with` 문에 대한 지원이 추가되었습니다. 새로운 파일 기술자는 이제 상속 불가능합니다.

버전 3.4부터 폐지됨: *flags* 매개 변수. 이제 기본적으로 `select.EPOLL_CLOEXEC`가 사용됩니다. 파일 기술자를 상속 가능하게 하려면 `os.set_inheritable()`을 사용하십시오.

select.**poll**()

(모든 운영 체제에서 지원되는 것은 아닙니다.) 파일 기술자 등록과 등록 해지를 지원하고 그런 다음 I/O 이벤트에 대해 폴링하는 폴링 객체를 반환합니다; 폴링 객체가 지원하는 메서드에 대해서는 아래의 폴링 객체 섹션을 참조하십시오.

select.**kqueue**()

(BSD에서만 지원됩니다.) 커널 큐 객체를 반환합니다; kqueue 객체가 지원하는 메서드에 대해서는 아래의 *Kqueue* 객체 섹션을 참조하십시오.

새 파일 기술자는 상속 불가능합니다.

버전 3.4에서 변경: 새 파일 기술자는 이제 상속 불가능합니다.

select.**kevent**(*ident*, *filter=KQ_FILTER_READ*, *flags=KQ_EV_ADD*, *fflags=0*, *data=0*, *udata=0*)

(BSD에서만 지원됩니다.) 커널 이벤트 객체를 반환합니다. kevent 객체가 지원하는 메서드에 대해서는 아래의 *Kevent* 객체 섹션을 참조하십시오.

select.**select**(*rlist*, *wlist*, *xlist*[, *timeout*])

이것은 유닉스 `select()` 시스템 호출에 대한 간단한 인터페이스입니다. 처음 세 개의 인자는 '대기 가능한 객체(waitable objects)' 의 이터러블입니다: 대기 가능한 객체는 파일 기술자를 나타내는 정수나 그런 정수를 반환하는 `fileno()`라는 매개 변수 없는 메서드를 가진 객체입니다:

- *rlist*: 읽을 준비가 될 때까지 기다립니다
- *wlist*: 쓰기 준비가 될 때까지 기다립니다
- *xlist*: "예외 조건"을 기다립니다 (시스템에서 어떤 것들을 이러한 조건으로 간주하는지는 매뉴얼 페이지를 참조하십시오)

빈 이터러블이 허용되지만, 세 개의 빈 이터러블을 받아들일지는 플랫폼에 따라 다릅니다. (유닉스에서는 작동하지만 윈도우에서는 작동하지 않는 것으로 알려져 있습니다.) 선택적 *timeout* 인자는 시간제한을 초 단위의 부동 소수점 숫자로 지정합니다. *timeout* 인자가 생략되면 최소한 하나의 파일 기술자가 준비될 때까지 함수가 블록 합니다. 시간제한 값이 0이면 폴링을 지정하고 절대 블록 하지 않습니다.

반환 값은 준비된 객체의 리스트 3개입니다: 처음 세 인자의 부분 집합입니다. 파일 기술자가 준비되지 않고 시간제한에 도달하면, 세 개의 빈 리스트가 반환됩니다.

이터러블에서 허용되는 객체 형에는 파이썬 파일 객체 (예를 들어 `sys.stdin`, 또는 `open()`이나 `os.popen()`에서 반환된 객체), `socket.socket()`에서 반환된 소켓 객체가 있습니다. 적절한 (단지

임의의 정수가 아니라 실제 파일 기술자를 반환하는) *fileno()* 메서드가 있는 한, 래퍼 (*wrapper*) 클래스를 직접 정의할 수도 있습니다.

> **참고**
>
> 윈도우에서 파일 객체는 허용되지 않지만, 소켓은 허용됩니다. 윈도우에서, 하부 `select()` 함수는 WinSock 라이브러리에서 제공되며, WinSock에서 생성되지 않은 파일 기술자를 처리하지 않습니다.

버전 3.5에서 변경: 시그널 처리기가 *InterruptedError*를 발생시키는 대신 예외를 발생시키는 경우 (이유는 **PEP 475**를 참조하십시오)를 제외하고, 시그널에 의해 인터럽트 될 때 다시 계산된 시간제한으로 함수가 다시 시도됩니다.

`select.`**`PIPE_BUF`**

파이프가 *select()*, `poll()` 또는 이 모듈의 다른 인터페이스에서 쓰기 준비가 된 것으로 보고될 때, 파이프를 블록 하지 않고 쓸 수 있는 최소 바이트 수. 소켓과 같은 다른 파일류 객체에는 적용되지 않습니다.

이 값은 POSIX에서 512 이상이 되도록 보장합니다.

가용성: Unix

Added in version 3.2.

19.4.1 `/dev/poll` 폴링 객체

Solaris와 파생 제품에는 `/dev/poll`이 있습니다. `select()`는 *O*(가장 높은 파일 기술자)이고 `poll()`은 *O*(파일 기술자 수)이지만 `/dev/poll`은 *O*(활성 파일 기술자)입니다.

`/dev/poll` 동작은 표준 `poll()` 객체에 매우 가깝습니다.

`devpoll.`**`close`**`()`

폴링 객체의 파일 기술자를 닫습니다.

Added in version 3.4.

`devpoll.`**`closed`**

폴링 객체가 닫혔으면 `True`.

Added in version 3.4.

`devpoll.`**`fileno`**`()`

폴링 객체의 파일 기술자 번호를 반환합니다.

Added in version 3.4.

`devpoll.`**`register`**`(`*fd*[, *eventmask*]`)`

폴링 객체에 파일 기술자를 등록합니다. 이후 *poll()* 메서드에 대한 호출은 파일 기술자에 계류 중인 I/O 이벤트가 있는지 확인합니다. *fd*는 정수이거나, 정수를 반환하는 *fileno()* 메서드가 있는 객체일 수 있습니다. 파일 객체는 `fileno()`를 구현하므로, 인자로도 사용할 수 있습니다.

*eventmask*는 확인할 이벤트 유형을 설명하는 선택적 비트 마스크입니다. 상수는 `poll()` 객체와 같습니다. 기본값은 상수 `POLLIN`, `POLLPRI` 및 `POLLOUT`의 조합입니다.

> **경고**
>
> 이미 등록된 파일 기술자를 등록하는 것은 에러가 아니지만, 결과는 정의되지 않습니다. 적절한 액션은 먼저 unregister 하거나 modify 하는 것입니다. 이것은 `poll()`과 비교하여 중요한 차이점입니다.

`devpoll.modify(fd[, eventmask])`

이 메서드는 `unregister()` 다음에 `register()`를 수행합니다. 명시적으로 같은 작업을 수행하는 것보다 조금 더 효율적입니다.

`devpoll.unregister(fd)`

폴링 객체가 추적하는 파일 기술자를 제거합니다. `register()` 메서드와 마찬가지로, *fd*는 정수이거나 정수를 반환하는 `fileno()` 메서드가 있는 객체일 수 있습니다.

등록되지 않은 파일 기술자를 제거하려고 하면 안전하게 무시됩니다.

`devpoll.poll([timeout])`

등록된 파일 기술자 집합을 폴링하고, 보고할 이벤트나 에러가 있는 기술자에 대한 `(fd, event)` 2-튜플이 포함된 비어있을 수 있는 리스트를 반환합니다. *fd*는 파일 기술자이고, *event*는 해당 기술자에 대해 보고된 이벤트에 대해 설정된 비트가 있는 비트 마스크입니다 — 대기 중인 입력은 `POLLIN`, 기술자에 쓸 수 있음을 나타내는 데는 `POLLOUT`, 등등. 빈 리스트는 시간제한을 초과하였고 보고할 이벤트가 있는 파일 기술자가 없음을 나타냅니다. *timeout*이 제공되면, 시스템이 반환하기 전에 이벤트를 기다리는 시간을 밀리초로 지정합니다. *timeout*을 생략하거나 -1이거나 `None`이면, 이 폴링 객체에 대한 이벤트가 있을 때까지 호출이 블록 됩니다.

버전 3.5에서 변경: 시그널 처리기가 `InterruptedError`를 발생시키는 대신 예외를 발생시키는 경우 (이유는 **PEP 475**를 참조하십시오)를 제외하고, 시그널에 의해 인터럽트 될 때 다시 계산된 시간제한으로 함수가 다시 시도됩니다.

19.4.2 에지와 레벨 트리거 폴링 (epoll) 객체

https://linux.die.net/man/4/epoll

eventmask

상수	의미
`EPOLLIN`	읽기 가능
`EPOLLOUT`	쓰기 가능
`EPOLLPRI`	읽을 긴급 데이터
`EPOLLERR`	연관된 fd에서 에러 조건이 발생했습니다
`EPOLLHUP`	연관된 fd에서 끊어짐(hang up)이 발생했습니다
`EPOLLET`	에지 트리거 동작을 설정합니다, 기본값은 레벨 트리거 동작입니다
`EPOLLONE`	원샷(one-shot) 동작을 설정합니다. 하나의 이벤트를 꺼낸 후에, fd는 내부적으로 비활성화됩니다.
`EPOLLEXC`	연관된 fd에 이벤트가 있을 때 하나의 epoll 객체만 깨웁니다. 기본값(이 플래그가 설정되지 않으면)은 fd를 폴링하는 모든 epoll 객체를 깨우는 것입니다.
`EPOLLRDH`	스트림 소켓 반대편이 연결을 닫았거나 연결의 쓰기 절반을 종료했습니다.
`EPOLLRDN`	`EPOLLIN`과 동등합니다
`EPOLLRDB`	우선순위가 높은 데이터 대역을 읽을 수 있습니다.
`EPOLLWRN`	`EPOLLOUT`과 동등합니다
`EPOLLWRB`	우선순위가 높은 데이터를 쓸 수 있습니다.
`EPOLLMSG`	무시됩니다.
`EPOLLWAK`	Prevents sleep during event waiting.

Added in version 3.6: `EPOLLEXCLUSIVE`가 추가되었습니다. 리눅스 커널 4.5 이상에서만 지원됩니다.

Added in version 3.14: `EPOLLWAKEUP` was added. It' s only supported by Linux Kernel 3.5 or later.

`epoll.close()`

epoll 객체의 제어 파일 기술자를 닫습니다.

`epoll.closed`

epoll 객체가 닫혔으면 `True`.

epoll.**fileno**()

제어 fd의 파일 기술자 번호를 반환합니다.

epoll.**fromfd**(*fd*)

주어진 파일 기술자에서 epoll 객체를 만듭니다.

epoll.**register**(*fd*[, *eventmask*])

epoll 객체에 fd 기술자를 등록합니다.

epoll.**modify**(*fd*, *eventmask*)

등록된 파일 기술자를 수정합니다.

epoll.**unregister**(*fd*)

epoll 객체에서 등록된 파일 기술자를 제거합니다.

버전 3.9에서 변경: 이 메서드는 더는 *EBADF* 에러를 무시하지 않습니다.

epoll.**poll**(*timeout=None*, *maxevents=-1*)

이벤트를 기다립니다. 시간제한은 초 단위입니다 (float)

버전 3.5에서 변경: 시그널 처리기가 *InterruptedError*를 발생시키는 대신 예외를 발생시키는 경우 (이유는 **PEP 475**를 참조하십시오)를 제외하고, 시그널에 의해 인터럽트 될 때 다시 계산된 시간제한으로 함수가 다시 시도됩니다.

19.4.3 폴링 객체

대부분의 유닉스 시스템에서 지원되는 `poll()` 시스템 호출은 동시에 많은 클라이언트에게 서비스를 제공하는 네트워크 서버에 더 나은 확장성을 제공합니다. `select()` 는 비트맵을 빌드하고, 관심 있는 fd에 대한 비트를 켠 후, 전체 비트 맵을 다시 선형으로 스캔해야 하지만, 이 시스템 호출은 관심 있는 파일 기술자만 나열하면 되기 때문에 `poll()` 은 더 잘 확장됩니다. `select()` 는 *O*(가장 높은 파일 기술자)이고 `poll()` 은 *O*(파일 기술자 수)입니다.

poll.**register**(*fd*[, *eventmask*])

폴링 객체에 파일 기술자를 등록합니다. 이후 *poll()* 메서드에 대한 호출은 파일 기술자에 계류 중인 I/O 이벤트가 있는지 확인합니다. *fd*는 정수이거나, 정수를 반환하는 *fileno()* 메서드가 있는 객체일 수 있습니다. 파일 객체는 `fileno()` 를 구현하므로, 인자로도 사용할 수 있습니다.

*eventmask*는 확인할 이벤트 유형을 설명하는 선택적 비트 마스크이고, 아래 표에 설명된 상수 `POLLIN`, `POLLPRI` 및 `POLLOUT`의 조합일 수 있습니다. 지정하지 않으면, 사용되는 기본값은 3가지 유형의 이벤트를 모두 확인합니다.

상수	의미
POLLIN	읽을 데이터가 있습니다
POLLPRI	읽을 긴급한 데이터가 있습니다
POLLOUT	출력 준비: 쓰기가 블록 되지 않을 것입니다
POLLERR	어떤 종류의 에러 조건
POLLHUP	끊어졌습니다(Hung up)
POLLRDHUP	스트림 소켓 반대편이 연결을 닫았거나, 연결의 쓰기 절반을 종료했습니다
POLLNVAL	잘못된 요청: 기술자가 열리지 않았습니다

이미 등록된 파일 기술자를 등록하는 것은 에러가 아니며, 기술자를 정확히 한 번 등록하는 것과 같은 효과가 있습니다.

poll.**modify**(*fd*, *eventmask*)

이미 등록된 fd를 수정합니다. 이것은 `register(fd, eventmask)` 와 같은 효과가 있습니다. 등록되지 않은 파일 기술자를 수정하려고 하면 errno `ENOENT`로 *OSError* 예외가 발생합니다.

poll.**unregister**(*fd*)

폴링 객체가 추적하는 파일 기술자를 제거합니다. `register()` 메서드와 마찬가지로, *fd*는 정수이거나 정수를 반환하는 `fileno()` 메서드가 있는 객체일 수 있습니다.

등록되지 않은 파일 기술자를 제거하려고 하면 `KeyError` 예외가 발생합니다.

poll.**poll**([*timeout*])

등록된 파일 기술자 집합을 폴링하고, 보고할 이벤트나 에러가 있는 기술자에 대한 `(fd, event)` 2-튜플이 포함된 비어있을 수 있는 리스트를 반환합니다. *fd*는 파일 기술자이고, *event*는 해당 기술자에 대해 보고된 이벤트에 대해 설정된 비트가 있는 비트 마스크입니다 — 대기 중인 입력은 `POLLIN`, 기술자에 쓸 수 있음을 나타내는 데는 `POLLOUT`, 등등. 빈 리스트는 시간제한을 초과하였고 보고할 이벤트가 있는 파일 기술자가 없음을 나타냅니다. *timeout*이 제공되면, 시스템이 반환하기 전에 이벤트를 기다리는 시간을 밀리초로 지정합니다. *timeout*을 생략하거나 음수이거나 `None`이면, 이 폴링 객체에 대한 이벤트가 있을 때까지 호출이 블록 됩니다.

버전 3.5에서 변경: 시그널 처리기가 `InterruptedError`를 발생시키는 대신 예외를 발생시키는 경우 (이유는 **PEP 475**를 참조하십시오)를 제외하고, 시그널에 의해 인터럽트 될 때 다시 계산된 시간제한으로 함수가 다시 시도됩니다.

19.4.4 Kqueue 객체

kqueue.**close**()

kqueue 객체의 제어 파일 기술자를 닫습니다.

kqueue.**closed**

kqueue 객체가 닫혔으면 `True`.

kqueue.**fileno**()

제어 fd의 파일 기술자 번호를 반환합니다.

kqueue.**fromfd**(*fd*)

주어진 파일 기술자에서 kqueue 객체를 만듭니다.

kqueue.**control**(*changelist*, *max_events*[, *timeout*]) → eventlist

kevent에 대한 저수준 인터페이스

- changelist는 kevent 객체의 이터러블 이거나 `None`이어야 합니다
- max_events는 0이거나 양의 정수여야 합니다.
- timeout은 초 단위 (float 가능); 기본값은 `None`이고 무한히 대기합니다

버전 3.5에서 변경: 시그널 처리기가 `InterruptedError`를 발생시키는 대신 예외를 발생시키는 경우 (이유는 **PEP 475**를 참조하십시오)를 제외하고, 시그널에 의해 인터럽트 될 때 다시 계산된 시간제한으로 함수가 다시 시도됩니다.

19.4.5 Kevent 객체

https://man.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

kevent.**ident**

이벤트를 식별하는 데 사용되는 값. 해석은 필터에 따라 다르지만, 일반적으로 파일 기술자입니다. 생성자에서 ident 는 정수이거나 `fileno()` 메서드가 있는 객체일 수 있습니다. kevent는 내부적으로 정수를 저장합니다.

kevent.**filter**

커널 필터의 이름.

상수	의미
KQ_FILTER_READ	기술자를 취하고 읽을 수 있는 데이터가 있을 때마다 반환합니다
KQ_FILTER_WRITE	기술자를 취하고 쓸 수 있는 데이터가 있을 때마다 반환합니다
KQ_FILTER_AIO	AIO 요청
KQ_FILTER_VNODE	*fflag*에서 감시 중인 요청된 이벤트 중 하나 이상이 발생할 때 반환합니다
KQ_FILTER_PROC	프로세스 id에서 이벤트를 감시합니다
KQ_FILTER_NETDEV	네트워크 장치에서 이벤트를 감시합니다 [맥 OS에서는 사용할 수 없습니다]
KQ_FILTER_SIGNAL	감시하는 시그널이 프로세스에 전달될 때마다 반환합니다
KQ_FILTER_TIMER	임의의 타이머를 설정합니다

kevent.**flags**

필터 액션.

상수	의미
KQ_EV_ADD	이벤트를 추가하거나 수정합니다
KQ_EV_DELETE	큐에서 이벤트를 제거합니다
KQ_EV_ENABLE	이벤트를 반환하도록 Permitscontrol() 합니다
KQ_EV_DISABLE	이벤트 비활성화
KQ_EV_ONESHOT	처음 발생한 후 이벤트를 제거합니다
KQ_EV_CLEAR	이벤트를 꺼낸 후 상태를 재설정합니다
KQ_EV_SYSFLAGS	내부 이벤트
KQ_EV_FLAG1	내부 이벤트
KQ_EV_EOF	필터 특정 EOF 조건
KQ_EV_ERROR	반환 값을 봅니다

kevent.**fflags**

필터 특정 플래그.

KQ_FILTER_READ와 KQ_FILTER_WRITE 필터 플래그:

상수	의미
KQ_NOTE_LOWAT	소켓 버퍼의 낮은 수위 (low water mark)

KQ_FILTER_VNODE 필터 플래그:

상수	의미
KQ_NOTE_DELETE	*unlink*() 가 호출되었습니다
KQ_NOTE_WRITE	쓰기가 발생했습니다
KQ_NOTE_EXTEND	파일이 확장되었습니다
KQ_NOTE_ATTRIB	속성이 변경되었습니다
KQ_NOTE_LINK	링크 수가 변경되었습니다
KQ_NOTE_RENAME	파일 이름이 변경되었습니다
KQ_NOTE_REVOKE	파일에 대한 액세스가 취소되었습니다

KQ_FILTER_PROC 필터 플래그:

상수	의미
KQ_NOTE_EXIT	프로세스가 종료되었습니다
KQ_NOTE_FORK	프로세스가 *fork*()를 호출했습니다
KQ_NOTE_EXEC	프로세스가 새로운 프로세스를 실행했습니다
KQ_NOTE_PCTRLMASK	내부 필터 플래그
KQ_NOTE_PDATAMASK	내부 필터 플래그
KQ_NOTE_TRACK	*fork*()를 가로질러 프로세스를 추적합니다
KQ_NOTE_CHILD	*NOTE_TRACK*의 경우 자식 프로세스에서 반환됩니다
KQ_NOTE_TRACKERR	자식에게 연결할 수 없습니다

KQ_FILTER_NETDEV 필터 플래그 (맥 OS에서는 사용할 수 없습니다):

상수	의미
KQ_NOTE_LINKUP	링크가 올라갔습니다
KQ_NOTE_LINKDOWN	링크가 내려갔습니다
KQ_NOTE_LINKINV	링크 상태가 유효하지 않습니다

kevent.**data**

필터 특정 데이터.

kevent.**udata**

사용자 정의 값.

19.5 selectors — 고수준 I/O 다중화

Added in version 3.4.

소스 코드: Lib/selectors.py

19.5.1 소개

이 모듈은 *select* 모듈 프리미티브에 기반하여 고수준의 효율적인 I/O 다중화를 가능하게 합니다. 사용되는 OS 수준 프리미티브를 정확하게 제어하고 싶지 않으면, 사용자는 이 모듈을 대신 사용하는 것이 좋습니다.

여러 파일 객체에 대한 I/O 준비 알림을 기다리는 데 사용할 수 있는 몇 가지 구체적인 구현 (*KqueueSelector*, *EpollSelector*…)과 함께 *BaseSelector* 추상 베이스 클래스를 정의합니다. 다음에서 "파일 객체"는 *fileno()* 메서드가 있는 모든 객체나 날 파일 기술자를 가리킵니다. 파일 객체를 참조하십시오.

*DefaultSelector*는 현재 플랫폼에서 사용할 수 있는 가장 효율적인 구현의 별칭입니다: 대부분 사용자는 기본적으로 이것을 선택해야 합니다.

> **참고**
>
> 지원되는 파일 객체의 유형은 플랫폼에 따라 다릅니다: 윈도우에서는 소켓은 지원되지만, 파이프는 지원되지 않습니다. 반면에, 유닉스에서는 둘 다 지원됩니다 (fifo나 특수 파일 장치와 같은 다른 유형도 지원될 수 있습니다).

> **더 보기**

select
저수준 I/O 다중화 모듈.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

19.5.2 클래스

클래스 계층 구조:

```
BaseSelector
+-- SelectSelector
+-- PollSelector
+-- EpollSelector
+-- DevpollSelector
+-- KqueueSelector
```

다음에서, *events*는 주어진 파일 객체에서 어떤 I/O 이벤트를 기다려야 하는지를 나타내는 비트 마스크입니다. 다음 모듈 상수의 조합일 수 있습니다:

상수	의미
selectors.**EVENT_READ**	읽기 가능
selectors.**EVENT_WRITE**	쓰기 가능

class selectors.**SelectorKey**

*SelectorKey*는 파일 객체에 그것의 하부 파일 기술자, 선택한 이벤트 마스크 및 첨부된 데이터를 연결하는 데 사용되는 *namedtuple*입니다. 여러 *BaseSelector* 메서드에 의해 반환됩니다.

fileobj
등록된 파일 객체.

fd
하부 파일 기술자.

events
이 파일 객체에서 기다려야 하는 이벤트.

data
이 파일 객체에 연결된 선택적인 불투명한 데이터: 예를 들어, 이것은 클라이언트별 세션 ID를 저장하는 데 사용될 수 있습니다.

class selectors.**BaseSelector**

*BaseSelector*는 여러 파일 객체에 대한 I/O 이벤트 준비를 기다리는 데 사용됩니다. 파일 스트림 등록, 등록 취소 및 선택적 제한 시간과 함께 해당 스트림에서 I/O 이벤트를 기다리는 메서드를 지원합니다. 추상 베이스 클래스이므로, 인스턴스를 만들 수 없습니다. *DefaultSelector*를 대신 사용하거나, 특정 구현을 사용하고 싶고, 플랫폼에서 지원한다면 *SelectSelector*, *KqueueSelector* 등을 사용하십시오. *BaseSelector*와 그것의 구상 구현은 컨텍스트 관리자 프로토콜을 지원합니다.

abstractmethod register(*fileobj*, *events*, *data=None*)
I/O 이벤트를 셀렉트하고 감시하기 위한 파일 객체를 등록합니다.

*fileobj*는 감시할 파일 객체입니다. 정수 파일 기술자이거나 `fileno()` 메서드를 가진 객체일 수 있습니다. *events*는 감시할 이벤트의 비트 마스크입니다. *data*는 불투명한 객체입니다.

새로운 `SelectorKey` 인스턴스를 반환하거나, 유효하지 않은 이벤트 마스크나 파일 기술자의 경우 `ValueError`를, 파일 객체가 이미 등록된 경우 `KeyError`를 발생시킵니다.

abstractmethod unregister(*fileobj*)

셀렉트로부터 파일 객체를 등록 해지하고, 감시에서 삭제합니다. 파일 객체는 닫히기 전에 등록 해지되어야 합니다.

*fileobj*는 이전에 등록된 파일 객체여야 합니다.

연결된 `SelectorKey` 인스턴스를 반환하거나, *fileobj*가 등록되어 있지 않으면 `KeyError`를 발생시킵니다. *fileobj*가 유효하지 않으면 (예를 들어, `fileno()` 메서드가 없거나 `fileno()` 메서드가 유효하지 않은 반환 값을 가지면) `ValueError`가 발생합니다.

modify(*fileobj*, *events*, *data=None*)

등록된 파일 객체의 감시되는 이벤트나 첨부된 데이터를 변경합니다.

이것은 더 효율적으로 구현될 수 있다는 점을 제외하면, `BaseSelector.unregister(fileobj)` 다음에 `BaseSelector.register(fileobj, events, data)` 하는 것과 동등합니다.

새로운 `SelectorKey` 인스턴스를 반환하거나, 유효하지 않은 이벤트 마스크나 파일 기술자의 경우 `ValueError`를, 파일 객체가 등록되지 않았으면 `KeyError`를 발생시킵니다.

abstractmethod select(*timeout=None*)

등록된 파일 객체의 일부가 준비될 때까지 기다리거나, 제한 시간이 만료됩니다.

`timeout > 0` 이면, 최대 대기 시간을 초로 지정합니다. `timeout <= 0`이면, 호출은 블록하지 않고, 현재 준비된 파일 객체를 보고합니다. *timeout*이 `None`이면, 감시되는 파일 객체가 준비될 때까지 호출이 블록 됩니다.

각 준비된 파일 객체마다 하나씩, `(key, events)` 튜플의 리스트를 반환합니다.

*key*는 준비된 파일 객체에 해당하는 `SelectorKey` 인스턴스입니다. *events*는 이 파일 객체에서 준비된 이벤트의 비트 마스크입니다.

> **참고**
>
> 현재 프로세스가 시그널을 받으면, 이 메서드는 파일 객체가 준비되거나 제한 시간이 지나기 전에 반환할 수 있습니다: 이때는 빈 리스트가 반환됩니다.

버전 3.5에서 변경: 시그널에 의해 인터럽트 되었을 때, 셀렉터는 이제 시그널 처리기가 예외를 발생시키지 않으면, 제한 시간 이전에 빈 이벤트 리스트를 반환하는 대신, 재계산 된 제한 시간으로 재 시도됩니다 (근거는 **PEP 475**를 참조하세요).

close()

셀렉터를 닫습니다.

모든 하부 자원을 해제하기 위해 이 메서드를 호출해야 합니다. 일단 닫은 후에는 셀렉터를 더는 사용하지 않아야 합니다.

get_key(*fileobj*)

등록된 파일 객체에 연결된 키를 반환합니다.

이 파일 객체에 연결된 `SelectorKey` 인스턴스를 반환하거나, 파일 객체가 등록되지 않았으면 `KeyError`를 발생시킵니다.

abstractmethod get_map()

파일 객체에서 셀렉터로의 매핑을 반환합니다.

등록된 파일 객체를 연결된 `SelectorKey` 인스턴스로 매핑하는 `Mapping` 인스턴스를 반환합니다.

class selectors.**DefaultSelector**

현재의 플랫폼에서 사용할 수 있는 가장 효율적인 구현을 사용하는 기본 셀렉터 클래스입니다. 대부분 사용자는 기본적으로 이것을 선택해야 합니다.

class selectors.**SelectSelector**

select.select() 기반 셀렉터.

class selectors.**PollSelector**

select.poll() 기반 셀렉터.

class selectors.**EpollSelector**

select.epoll() 기반 셀렉터.

fileno()

하부 *select.epoll()* 객체에서 사용하는 파일 기술자를 반환합니다.

class selectors.**DevpollSelector**

select.devpoll() 기반 셀렉터.

fileno()

하부 *select.devpoll()* 객체에서 사용하는 파일 기술자를 반환합니다.

Added in version 3.5.

class selectors.**KqueueSelector**

select.kqueue() 기반 셀렉터.

fileno()

하부 *select.kqueue()* 객체에서 사용하는 파일 기술자를 반환합니다.

19.5.3 예제

다음은 간단한 메아리 서버 구현입니다:

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # 준비돼 있어야 합니다
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000)  # 준비돼 있어야 합니다
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # 블록하지 않기를 바랍니다
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

19.6 signal — 비동기 이벤트에 대한 처리기 설정

소스 코드: Lib/signal.py

이 모듈은 파이썬에서 시그널 처리기를 사용하는 메커니즘을 제공합니다.

19.6.1 일반 규칙

signal.signal() 함수는 시그널이 수신될 때 실행될 사용자 정의 처리기를 정의하도록 합니다. 소수의 기본 처리기가 설치됩니다: *SIGPIPE*는 무시되고 (그래서 파이프와 소켓에 대한 쓰기 에러는 일반 파이썬 예외로 보고될 수 있습니다) *SIGINT*는 부모 프로세스가 변경하지 않았다면 *KeyboardInterrupt* 예외로 번역됩니다.

일단 설정되면, 특정 시그널에 대한 처리기는 명시적으로 재설정 될 때까지 (파이썬은 하부 구현과 관계없이 BSD 스타일 인터페이스를 흉내 냅니다) 설치된 상태로 유지됩니다. *SIGCHLD*에 대한 처리기는 예외인데, 하부 구현을 따릅니다.

On WebAssembly platforms, signals are emulated and therefore behave differently. Several functions and signals are not available on these platforms.

파이썬 시그널 처리기의 실행

파이썬 시그널 처리기는 저수준 (C) 시그널 처리기 내에서 실행되지 않습니다. 대신, 저수준 시그널 처리기는 가상 기계에게 나중에 (예를 들어 다음 바이트 코드 명령에서) 해당 파이썬 시그널 처리기를 실행하도록 지시하는 플래그를 설정합니다. 결과는 다음과 같습니다:

- C 코드에서의 유효하지 않은 연산으로 인해 발생하는 *SIGFPE*나 *SIGSEGV*와 같은 동기 에러를 잡는 것은 그리 의미가 없습니다. 파이썬은 시그널 처리기에서 C 코드로 돌아오는데, 같은 시그널을 다시 발생시켜서, 파이썬이 멈출 것입니다. 파이썬 3.3부터는, *faulthandler* 모듈을 사용하여 동기 에러를 보고할 수 있습니다.
- C로만 구현된 오래 실행되는 계산(가령 커다란 텍스트 본문에 대한 정규식 일치)은 수신된 시그널에 상관없이 임의의 시간 동안 중단없이 실행될 수 있습니다. 계산이 끝나면 파이썬 시그널 처리기가 호출됩니다.
- If the handler raises an exception, it will be raised "out of thin air" in the main thread. See the *note below* for a discussion.

시그널과 스레드

파이썬 시그널 처리기는 시그널이 다른 스레드에서 수신될 때도 항상 메인 인터프리터의 메인 파이썬 스레드에서 실행됩니다. 이는 시그널을 스레드 간 통신 수단으로 사용할 수 없음을 의미합니다. 대신 *threading* 모듈의 동기화 프리미티브를 사용할 수 있습니다.

게다가, 메인 인터프리터의 메인 스레드만 새로운 시그널 처리기를 설정할 수 있습니다.

19.6.2 모듈 내용

버전 3.5에서 변경: 이하에 열거된 시그널 (SIG*), 처리기 (*SIG_DFL*, *SIG_IGN*) 및 sigmask (*SIG_BLOCK*, *SIG_UNBLOCK*, *SIG_SETMASK*) 관련 상수는 **열거형**으로 바뀌었습니다 (각각 *Signals*, *Handlers* 및

Sigmasks). *getsignal()*, *pthread_sigmask()*, *sigpending()* 및 *sigwait()* 함수는 사람이 읽을 수 있는 **열거형**을 *Signals* 객체로 반환합니다.

signal 모듈은 세개의 열거형을 정의합니다:

class signal.**Signals**

enum.IntEnum collection of SIG* constants and the CTRL_* constants.

Added in version 3.5.

class signal.**Handlers**

enum.IntEnum collection the constants *SIG_DFL* and *SIG_IGN*.

Added in version 3.5.

class signal.**Sigmasks**

enum.IntEnum collection the constants *SIG_BLOCK*, *SIG_UNBLOCK* and *SIG_SETMASK*.

가용성: Unix.

자세한 내용은 매뉴얼 페이지 *sigprocmask(2)*과 *pthread_sigmask(3)*을 참조하십시오.

Added in version 3.5.

signal 모듈에 정의된 변수는 다음과 같습니다:

signal.**SIG_DFL**

이것은 두 가지 표준 시그널 처리 옵션 중 하나입니다; 단순히 시그널의 기본 기능을 수행합니다. 예를 들어, 대부분의 시스템에서 SIGQUIT의 기본 동작은 코어를 덤프하고 종료하는 것이지만, *SIGCHLD*의 기본 동작은 단순히 무시하는 것입니다.

signal.**SIG_IGN**

이것은 주어진 시그널을 무시하는 또 다른 표준 시그널 처리기입니다.

signal.**SIGABRT**

*abort(3)*로 부터의 중단 시그널.

signal.**SIGALRM**

*alarm(2)*로 부터의 타이머 시그널.

가용성: Unix.

signal.**SIGBREAK**

키보드 인터럽트 (CTRL + BREAK).

가용성: Windows.

signal.**SIGBUS**

버스 에러 (메모리 액세스 불량).

가용성: Unix.

signal.**SIGCHLD**

자식 프로세스가 중지되었거나 종료되었습니다.

가용성: Unix.

signal.**SIGCLD**

*SIGCHLD*에 대한 별칭.

가용성: not macOS.

signal.**SIGCONT**

현재 중지되었으면 프로세스를 재개합니다.

가용성: Unix.

signal.**SIGFPE**

부동 소수점 예외. 예를 들어, 0으로 나누기.

> **더 보기**
>
> 나누기나 모듈로 연산의 두 번째 인자가 0이면 `ZeroDivisionError`가 발생합니다.

signal.**SIGHUP**

제어 터미널이 끊어졌거나 제어 프로세스가 죽었습니다.

가용성: Unix.

signal.**SIGILL**

잘못된 명령어.

signal.**SIGINT**

키보드 인터럽트 (CTRL + C).

기본 액션은 `KeyboardInterrupt`를 발생시키는 것입니다.

signal.**SIGKILL**

킬 시그널.

잡거나, 차단하거나, 무시할 수 없습니다.

가용성: Unix.

signal.**SIGPIPE**

끊어진 파이프: 판독기가 없는 파이프에 쓰기.

기본 동작은 시그널을 무시하는 것입니다.

가용성: Unix.

signal.**SIGSEGV**

세그먼테이션 오류: 유효하지 않은 메모리 참조.

signal.**SIGSTKFLT**

Stack fault on coprocessor. The Linux kernel does not raise this signal: it can only be raised in user space.

가용성: Linux.

시그널을 사용할 수 있는 아키텍처에서. 자세한 내용은 매뉴얼 페이지 `signal(7)`를 참조하십시오.

Added in version 3.11.

signal.**SIGTERM**

종료 시그널.

signal.**SIGUSR1**

사용자 정의 시그널 1.

가용성: Unix.

signal.**SIGUSR2**

사용자 정의 시그널 2.

가용성: Unix.

signal.**SIGWINCH**

창 크기 조정 시그널.

가용성: Unix.

SIG*

모든 시그널 번호는 기호적으로 정의됩니다. 예를 들어, 행업(hangup) 시그널은 *signal.SIGHUP*으로 정의됩니다; 변수 이름은 `<signal.h>`에 있는 C 프로그램에서 사용되는 이름과 동일합니다. '`signal()`'에 대한 유닉스 매뉴얼 페이지는 존재하는 시그널을 나열합니다 (일부 시스템에서는 *signal(2)*이고, 다른 시스템에서는 *signal(7)*입니다). 모든 시스템이 같은 시그널 이름 집합을 정의하는 것은 아님에 유의하십시오; 시스템에서 정의한 이름만 이 모듈에서 정의합니다.

signal.**CTRL_C_EVENT**

`Ctrl+C` 키 입력 이벤트에 해당하는 시그널. 이 시그널은 *os.kill()*에서만 사용할 수 있습니다.

가용성: Windows.

Added in version 3.2.

signal.**CTRL_BREAK_EVENT**

`Ctrl+Break` 키 입력 이벤트에 해당하는 시그널. 이 시그널은 *os.kill()*에서만 사용할 수 있습니다.

가용성: Windows.

Added in version 3.2.

signal.**NSIG**

가장 높은 시그널 번호보다 하나 큰 값. 유효한 시그널 번호를 얻으려면 *valid_signals()*를 사용하십시오.

signal.**ITIMER_REAL**

간격 타이머(interval timer)를 실시간으로 감소시키고, 만료 시 *SIGALRM*을 전달합니다.

signal.**ITIMER_VIRTUAL**

프로세스가 실행 중일 때만 간격 타이머(interval timer)를 감소시키고, 만료 시 SIGVTALRM을 전달합니다.

signal.**ITIMER_PROF**

프로세스가 실행될 때와 시스템이 프로세스를 대신하여 실행될 때 간격 타이머(interval timer)를 감소시킵니다. ITIMER_VIRTUAL과 함께 사용되어, 이 타이머는 일반적으로 사용자와 커널 공간에서 응용 프로그램이 소비한 시간을 프로파일링하는 데 사용됩니다. 만료 시 SIGPROF를 전달합니다.

signal.**SIG_BLOCK**

*pthread_sigmask()*의 *how* 매개 변수에 가능한 값으로 시그널이 차단됨을 나타냅니다.

Added in version 3.3.

signal.**SIG_UNBLOCK**

*pthread_sigmask()*의 *how* 매개 변수에 가능한 값으로 시그널이 차단 해제됨을 나타냅니다.

Added in version 3.3.

signal.**SIG_SETMASK**

*pthread_sigmask()*의 *how* 매개 변수에 가능한 값으로 시그널 마스크가 교체됨을 나타냅니다.

Added in version 3.3.

signal 모듈은 하나의 예외를 정의합니다:

exception signal.**ItimerError**

하부 *setitimer()*나 *getitimer()* 구현으로부터의 에러를 알리기 위해 발생합니다. 유효하지 않은 간격 타이머나 음의 시간이 *setitimer()*에 전달되면 이 에러가 예상됩니다. 이 에러는 *OSError*의 서브 형입니다.

Added in version 3.3: 이 에러는 *IOError*의 서브 형이었습니다, 이제는 *OSError*의 별칭입니다.

signal 모듈은 다음 함수를 정의합니다:

signal.**alarm**(*time*)

*time*이 0이 아니면, 이 함수는 `SIGALRM` 시그널이 *time* 초 내에 프로세스로 전송되도록 요청합니다. 이전에 예약된 알람은 취소됩니다 (임의의 시간에 오직 하나의 알람만 예약될 수 있습니다). 반환된 값은 이전에 설정된 알람이 전달되기까지 남은 초(seconds) 입니다. *time*이 0이면, 알람이 예약되지 않고, 예약된 알람이 취소됩니다. 반환 값이 0이면, 현재 예약된 알람이 없습니다.

가용성: Unix.

자세한 내용은 매뉴얼 페이지 `alarm(2)`를 참조하십시오.

signal.**getsignal**(*signalnum*)

시그널 *signalnum*에 대한 현재 시그널 처리기를 반환합니다. 반환된 값은 콜러블 파이썬 객체이거나, 특수 값 `signal.SIG_IGN`, `signal.SIG_DFL` 중 하나이거나 `None`일 수 있습니다. 여기서 `signal.SIG_IGN`은 시그널이 이전에 무시되었음을 의미하고, `signal.SIG_DFL`은 시그널을 처리하는 기본 방법이 이전에 사용 중임을 의미하고, `None`은 이전 시그널 처리기가 파이썬에서 설치되지 않았음을 의미합니다.

signal.**strsignal**(*signalnum*)

시그널 *signalnum*의 설명을 반환합니다, 가령 `SIGINT`의 경우 "Interrupt". *signalnum* 에 설명이 없으면 `None`을 반환합니다. *signalnum*이 유효하지 않으면 `ValueError`를 발생시킵니다.

Added in version 3.8.

signal.**valid_signals**()

이 플랫폼에서 유효한 시그널 번호 집합을 반환합니다. 일부 시그널이 시스템에서 내부 용으로 예약되었으면 `range(1, NSIG)` 보다 작을 수 있습니다.

Added in version 3.8.

signal.**pause**()

시그널이 수신될 때까지 프로세스를 휴면 상태로 만듭니다; 그런 다음 적절한 처리기가 호출됩니다. 아무것도 반환하지 않습니다.

가용성: Unix.

자세한 내용은 매뉴얼 페이지 `signal(2)`를 참조하십시오.

`sigwait()`, `sigwaitinfo()`, `sigtimedwait()` 및 `sigpending()`도 참조하십시오.

signal.**raise_signal**(*signum*)

호출하는 프로세스에 시그널을 보냅니다. 아무것도 반환하지 않습니다.

Added in version 3.8.

signal.**pidfd_send_signal**(*pidfd, sig, siginfo=None, flags=0*)

파일 기술자 *pidfd*가 참조하는 프로세스로 시그널 *sig*를 보냅니다. 파이썬은 현재 *siginfo* 매개 변수를 지원하지 않습니다; `None`이어야 합니다. *flags* 인자는 향후 확장을 위해 제공됩니다; 현재는 플래그 값이 정의되어 있지 않습니다.

자세한 내용은 `pidfd_send_signal(2)` 매뉴얼 페이지를 참조하십시오.

가용성: Linux >= 5.1, Android >= `build-time` API level 31

Added in version 3.9.

signal.**pthread_kill**(*thread_id, signalnum*)

시그널 *signalnum*을 호출자와 같은 프로세스의 다른 스레드인 스레드 *thread_id*로 보냅니다. 대상 스레드는 임의의 (파이썬이거나 아닌) 코드를 실행 중일 수 있습니다. 그러나, 대상 스레드가 파이썬 인터프리터를 실행 중이면, 파이썬 시그널 처리기는 메인 인터프리터의 메인 스레드에서 실행됩니다. 따라서, 특정 파이썬 스레드에 시그널을 보내는 것의 유일한 용도는 실행 중인 시스템 호출이 `InterruptedError`로 실패하도록 하는 것입니다.

*thread_id*에 적합한 값을 얻으려면 `threading.get_ident()` 나 `threading.Thread` 객체의 `ident` 어트리뷰트를 사용하십시오.

*signalnum*이 0이면, 시그널이 전송되지 않지만, 여전히 에러 검사가 수행됩니다; 대상 스레드가 여전히 실행 중인지 확인하는 데 사용할 수 있습니다.

인자 `thread_id`, `signalnum`으로 감사 이벤트 `signal.pthread_kill`을 발생시킵니다.

가용성: Unix.

자세한 내용은 매뉴얼 페이지 *pthread_kill(3)*을 참조하십시오.

*os.kill()*도 참조하십시오.

Added in version 3.3.

`signal.`**`pthread_sigmask`**(*how*, *mask*)

호출하는 스레드의 시그널 마스크를 가져오거나 변경하거나 가져오면서 변경합니다. 시그널 마스크는 호출자에게 현재 배달이 차단된 시그널 집합입니다. 이전 시그널 마스크를 시그널 집합으로 반환합니다.

호출의 동작은 다음과 같이 *how* 값에 따라 다릅니다.

- *SIG_BLOCK*: 차단된 시그널 집합은 현재 집합과 *mask* 인자의 합집합입니다.
- *SIG_UNBLOCK*: *mask*에 있는 시그널이 차단된 시그널의 현재 집합에서 제거됩니다. 차단되지 않은 시그널을 차단 해제하려고 시도할 수 있습니다.
- *SIG_SETMASK*: 차단된 시그널 집합이 *mask* 인자로 설정됩니다.

*mask*는 시그널 번호 집합입니다 (예를 들어 {*signal.SIGINT*, *signal.SIGTERM*}). 모든 시그널을 포함한 전체 마스크를 얻으려면 *valid_signals()*를 사용하십시오.

예를 들어, `signal.pthread_sigmask(signal.SIG_BLOCK, [])`은 호출하는 스레드의 시그널 마스크를 읽습니다.

*SIGKILL*과 `SIGSTOP`은 차단할 수 없습니다.

가용성: Unix.

자세한 내용은 매뉴얼 페이지 *sigprocmask(2)*과 *pthread_sigmask(3)*을 참조하십시오.

pause(), *sigpending()* 및 *sigwait()*도 참조하십시오.

Added in version 3.3.

`signal.`**`setitimer`**(*which*, *seconds*, *interval=0.0*)

seconds(*alarm()*과 달리 float가 허용됩니다) 이후에 그리고 (*interval*이 0이 아니면) 그 후로 *interval* 초마다 발사(fire)하도록 *which*로 지정된 간격 타이머(*signal.ITIMER_REAL*, *signal.ITIMER_VIRTUAL* 또는 *signal.ITIMER_PROF* 중 하나)를 설정합니다. *which*로 지정된 간격 타이머는 *seconds*를 0으로 설정하여 지울 수 있습니다.

간격 타이머가 발사(fire)하면, 시그널이 프로세스로 전송됩니다. 전송된 시그널은 사용 중인 타이머에 따라 다릅니다; *signal.ITIMER_REAL*은 *SIGALRM*을, *signal.ITIMER_VIRTUAL*은 `SIGVTALRM`을, *signal.ITIMER_PROF*는 `SIGPROF`를 전달합니다.

이전 값은 튜플로 반환됩니다: (지연, 간격).

유효하지 않은 간격 타이머를 전달하려고 하면 *ItimerError*가 발생합니다.

가용성: Unix.

`signal.`**`getitimer`**(*which*)

*which*로 지정된 주어진 간격 타이머의 현재 값을 반환합니다.

가용성: Unix.

`signal.`**`set_wakeup_fd`**(*fd*, ***, *warn_on_full_buffer=True*)

웨이크업 파일 기술자를 *fd*로 설정합니다. 프로그램에서 시그널 처리기를 등록한 시그널이 수신되면, 시그널 번호는 단일 바이트로 fd에 기록됩니다. 관심 있는 시그널에 대한 시그널 처리기를 등록하지 않았으면, 웨이크업 fd에 아무 것도 기록되지 않습니다. 라이브러리에서 poll이나 select 호출을 깨워서, 시그널을 완전히 처리하는 데 사용될 수 있습니다.

이전 웨이크업 fd가 반환됩니다 (또는 파일 기술자 웨이크업이 활성화되지 않았으면 -1). *fd*가 -1이면, 파일 기술자 웨이크업이 비활성화됩니다. -1이 아니면, *fd*는 비 블로킹이어야 합니다. poll이나 select를 다시 호출하기 전에 *fd*에서 바이트를 제거하는 것은 라이브러리의 책임입니다.

스레드가 활성화되었을 때, 이 함수는 메인 인터프리터의 메인 스레드에서만 호출할 수 있습니다; 다른 스레드에서 호출하려고 하면 `ValueError` 예외가 발생합니다.

이 함수를 사용하는 일반적인 두 가지 방법이 있습니다. 두 방법 모두, 시그널이 도착할 때 깨어나기 위해 fd를 사용하지만, 어떤 시그널이나 시그널들이 도착했는지 판단하는 방법이 다릅니다.

첫 번째 방법에서는, fd의 버퍼에서 데이터를 읽고, 바이트 값이 시그널 번호를 제공합니다. 이것은 간단합니다만, 드물게 문제가 될 수 있습니다: 일반적으로 fd에는 제한된 버퍼 공간이 있으며, 너무 많은 시그널이 너무 빨리 도착하면, 버퍼가 가득 차고, 일부 시그널이 손실될 수 있습니다. 이 방법을 사용하면, 시그널이 손실될 때 최소한 stderr에 경고가 인쇄되도록 `warn_on_full_buffer=True`를 설정해야 합니다.

두 번째 방법에서는, 오직 웨이크업만을 위해 웨이크업 fd를 사용하고, 실제 바이트 값은 무시합니다. 이 경우, 우리가 신경 쓰는 것은 fd의 버퍼가 비어 있는지 비어 있지 않은지 입니다; 가득 찬 버퍼는 전혀 문제를 가리키지 않습니다. 이 방법을 사용하면, 사용자가 가짜 경고 메시지로 혼동되지 않도록 `warn_on_full_buffer=False`를 설정해야 합니다.

버전 3.5에서 변경: 윈도우에서, 이 함수는 이제 소켓 핸들도 지원합니다.

버전 3.7에서 변경: `warn_on_full_buffer` 매개 변수를 추가했습니다.

`signal.`**`siginterrupt`**(*signalnum*, *flag*)

시스템 호출 재시작 동작을 변경합니다: *flag*가 `False`이면, 시그널 *signalnum*에 의해 인터럽트 될 때 시스템 호출이 다시 시작되고, 그렇지 않으면 시스템 호출이 중단됩니다. 아무것도 반환하지 않습니다.

가용성: Unix.

자세한 내용은 매뉴얼 페이지 `siginterrupt(3)`을 참조하십시오.

`signal()`로 시그널 처리기를 설치하면 주어진 시그널에 대해 *flag* 값을 참으로 `siginterrupt()`를 묵시적으로 호출하여 재시작 동작을 인터럽트 가능으로 재설정합니다.

`signal.`**`signal`**(*signalnum*, *handler*)

시그널 *signalnum*의 처리기를 함수 *handler*로 설정합니다. *handler*는 두 개의 인자(아래를 참조하십시오)를 취하는 콜러블 파이썬 객체, 또는 특수 값 `signal.SIG_IGN`이나 `signal.SIG_DFL` 중 하나일 수 있습니다. 이전 시그널 처리기가 반환됩니다 (위의 `getsignal()` 설명을 참조하십시오). (자세한 내용은 유닉스 매뉴얼 페이지 `signal(2)`를 참조하십시오.)

스레드가 활성화되었을 때, 이 함수는 메인 인터프리터의 메인 스레드에서만 호출할 수 있습니다; 다른 스레드에서 호출하려고 하면 `ValueError` 예외가 발생합니다.

*handler*는 두 개의 인자로 호출됩니다: 시그널 번호와 현재 스택 프레임 (`None`이나 프레임 객체; 프레임 객체에 대한 설명은, 형 계층에 있는 설명을 참조하거나 `inspect` 모듈의 어트리뷰트 설명을 참조하십시오).

윈도우에서, `signal()`은 `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, `SIGTERM`또는 `SIGBREAK`로만 호출 할 수 있습니다. 다른 경우에는 `ValueError`가 발생합니다. 모든 시스템이 같은 시그널 이름 집합을 정의하는 것은 아님에 유의하십시오; 시그널 이름이 `SIG*` 모듈 수준 상수로 정의되지 않으면 `AttributeError`가 발생합니다.

`signal.`**`sigpending`**()

호출하는 스레드로 전달 계류 중인 시그널 집합을 검사합니다 (즉, 차단된 동안 발생한 시그널). 계류 중인 시그널 집합을 반환합니다.

가용성: Unix.

자세한 내용은 매뉴얼 페이지 `sigpending(2)`를 참조하십시오.

`pause()`, `pthread_sigmask()` 및 `sigwait()`도 참조하십시오.

Added in version 3.3.

signal.**sigwait**(*sigset*)

시그널 집합 *sigset*에 지정된 시그널 중 하나가 전달될 때까지 호출하는 스레드의 실행을 일시 중단합니다. 이 함수는 시그널을 받아들이고 (계류 중인 시그널 목록에서 제거합니다), 시그널 번호를 반환합니다.

가용성: Unix.

자세한 내용은 매뉴얼 페이지 `sigwait(3)`을 참조하십시오.

`pause()`, `pthread_sigmask()`, `sigpending()`, `sigwaitinfo()` 및 `sigtimedwait()`도 참조하십시오.

Added in version 3.3.

signal.**sigwaitinfo**(*sigset*)

시그널 집합 *sigset*에 지정된 시그널 중 하나가 전달될 때까지 호출하는 스레드의 실행을 일시 중단합니다. 이 함수는 시그널을 받아들이고 계류 중인 시그널 목록에서 제거합니다. *sigset*의 시그널 중 하나가 이미 호출하는 스레드에 대해 계류 중이면, 함수는 해당 시그널에 대한 정보와 함께 즉시 반환합니다. 전달된 시그널에 대해 시그널 처리기가 호출되지 않습니다. 이 함수는 *sigset*에 없는 시그널에 의해 중단되면 `InterruptedError`를 발생시킵니다.

반환 값은 `siginfo_t` 구조체에 포함된 데이터, 즉 `si_signo`, `si_code`, `si_errno`, `si_pid`, `si_uid`, `si_status`, `si_band`를 표현하는 객체입니다.

가용성: Unix.

자세한 내용은 매뉴얼 페이지 `sigwaitinfo(2)`를 참조하십시오.

`pause()`, `sigwait()` 및 `sigtimedwait()`도 참조하십시오.

Added in version 3.3.

버전 3.5에서 변경: 이 함수는 이제 *sigset*에 없는 시그널에 의해 중단되고 시그널 처리기가 예외를 발생시키지 않으면 재시도됩니다 (이유는 **PEP 475**를 참조하십시오).

signal.**sigtimedwait**(*sigset*, *timeout*)

`sigwaitinfo()`와 유사하지만, 시간제한을 지정하는 추가 *timeout* 인자를 취합니다. *timeout*이 `0`으로 지정되면, 폴링이 수행됩니다. 시간제한 초과가 발생하면 `None`을 반환합니다.

가용성: Unix.

자세한 내용은 매뉴얼 페이지 `sigtimedwait(2)`를 참조하십시오.

`pause()`, `sigwait()` 및 `sigwaitinfo()`도 참조하십시오.

Added in version 3.3.

버전 3.5에서 변경: 이 함수는 이제 *sigset*에 없는 시그널에 의해 중단되고 시그널 처리기가 예외를 발생시키지 않으면 다시 계산된 *timeout*으로 재시도됩니다 (이유는 **PEP 475**를 참조하십시오).

19.6.3 예제

다음은 최소한의 예제 프로그램입니다. `alarm()` 함수를 사용하여 파일을 여는 데 대기하는 시간을 제한합니다; 이것은 파일이 켜져 있지 않을 수 있는 직렬 장치를 위한 파일일 때 유용하며, 일반적으로 `os.open()`이 무기한 정지됩니다. 해결책은 파일을 열기 전에 5초 알람을 설정하는 것입니다; 작업이 너무 오래 걸리면, 알람 시그널이 전송되고, 처리기가 예외를 발생시킵니다.

```
import signal, os

def handler(signum, frame):
    signame = signal.Signals(signum).name
    print(f'Signal handler called with signal {signame} ({signum})')
    raise OSError("Couldn't open device!")

# 시그널 처리기와 5초 알람을 설정합니다
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# 이 open()은 무기한 멈출 수 있습니다
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # 알람을 비활성화합니다
```

19.6.4 SIGPIPE에 대한 참고 사항

프로그램의 출력을 *head(1)*와 같은 도구로 파이핑 하면 표준 출력의 수신기가 일찍 닫힐 때 여러분의 프로세스로 *SIGPIPE* 시그널이 전송됩니다. 이것은 `BrokenPipeError: [Errno 32] Broken pipe`와 같은 예외를 일으킵니다. 이 경우를 처리하려면, 다음과 같이 이 예외를 포착하도록 진입점을 감싸십시오:

```
import os
import sys

def main():
    try:
        # 큰 출력을 시뮬레이션합니다 (여러분의 코드가 이 루프를 대체합니다)
        for x in range(10000):
            print("y")
        # 이 try 블록 내부에 있는 동안 SIGPIPE를 강제로 트리거 하기 위해
        # 여기에서 출력을 플러시 합니다.
        sys.stdout.flush()
    except BrokenPipeError:
        # 파이썬은 종료 시 표준 스트림을 플러시 합니다; 종료 시 또 다른 BrokenPipeError
를
        # 피하고자 나머지 출력을 devnull 로 리디렉션 합니다
        devnull = os.open(os.devnull, os.O_WRONLY)
        os.dup2(devnull, sys.stdout.fileno())
        sys.exit(1)  # EPIPE에 대해 파이썬은 에러 코드 1로 종료합니다

if __name__ == '__main__':
    main()
```

*BrokenPipeError*를 피하고자 *SIGPIPE*의 배치를 *SIG_DFL*로 설정하지 마십시오. 그렇게 하면 프로그램이 여전히 쓰고 있는 동안 소켓 연결이 중단될 때마다 프로그램이 예기치 않게 종료되도록 합니다.

19.6.5 신호 처리기와 예외에 대한 참고 사항

If a signal handler raises an exception, the exception will be propagated to the main thread and may be raised after any *bytecode* instruction. Most notably, a *KeyboardInterrupt* may appear at any point during execution. Most Python code, including the standard library, cannot be made robust against this, and so a *KeyboardInterrupt* (or any other exception resulting from a signal handler) may on rare occasions put the program in an unexpected state.

To illustrate this issue, consider the following code:

```
class SpamContext:
    def __init__(self):
        self.lock = threading.Lock()

    def __enter__(self):
        # 여기에서 KeyboardInterrupt 가 발생하면, 괜찮습니다
        self.lock.acquire()
        # 여기에서 KeyboardInterrupt 가 발생하면, __exit__ 가 호출되지 않습니다
        ...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        # KeyboardInterrupt 는 함수 반환 직전에 발생할 수 있습니다

    def __exit__(self, exc_type, exc_val, exc_tb):
        ...
        self.lock.release()
```

For many programs, especially those that merely want to exit on *KeyboardInterrupt*, this is not a problem, but applications that are complex or require high reliability should avoid raising exceptions from signal handlers. They should also avoid catching *KeyboardInterrupt* as a means of gracefully shutting down. Instead, they should install their own *SIGINT* handler. Below is an example of an HTTP server that avoids *KeyboardInterrupt*:

```
import signal
import socket
from selectors import DefaultSelector, EVENT_READ
from http.server import HTTPServer, SimpleHTTPRequestHandler

interrupt_read, interrupt_write = socket.socketpair()

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    interrupt_write.send(b'\0')
signal.signal(signal.SIGINT, handler)

def serve_forever(httpd):
    sel = DefaultSelector()
    sel.register(interrupt_read, EVENT_READ)
    sel.register(httpd, EVENT_READ)

    while True:
        for key, _ in sel.select():
            if key.fileobj == interrupt_read:
                interrupt_read.recv(1)
                return
            if key.fileobj == httpd:
                httpd.handle_request()

print("Serving on port 8000")
httpd = HTTPServer(('', 8000), SimpleHTTPRequestHandler)
serve_forever(httpd)
print("Shutdown...")
```

19.7 mmap — 메모리 맵 파일 지원

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

메모리 맵 파일 객체는 동시에 *bytearray* 와 파일 객체처럼 작동합니다. *bytearray*를 기대하는 대부분 장소에서 mmap 객체를 사용할 수 있습니다. 예를 들어, *re* 모듈을 사용하여 메모리 맵 파일을 검색할 수 있습니다. `obj[index] = 97`를 사용해서 한 바이트를 변경하거나, 슬라이스에 대입하여 서브 시퀀스를 변경할 수도 있습니다: `obj[i1:i2] = b'...'`. 또한 현재 파일 위치에서 시작하여 데이터를 읽고 쓸 수 있고, 다른 위치로 파일을 `seek()` 할 수 있습니다.

메모리 맵 파일은 *mmap* 생성자로 만드는데, 유닉스와 윈도우에서 다릅니다. 두 경우 모두 갱신을 위해 열린 파일에 대한 파일 기술자를 제공해야 합니다. 기존 파이썬 파일 객체를 매핑하려면, *fileno()* 메서드를

사용하여 *fileno* 매개 변수에 대한 올바른 값을 가져오십시오. 그렇지 않으면, 파일 기술자를 직접 반환하는 `os.open()` 함수를 사용하여 파일을 열 수 있습니다 (완료되면 파일을 닫아야 합니다).

> **참고**
>
> 쓰기 가능하고 버퍼링 되는 파일에 대한 메모리 맵을 만들려면, 먼저 파일을 `flush()`해야 합니다. 버퍼에 대한 지역 변경 사항이 실제로 매핑에 반영되게 하는 데 필요합니다.

유닉스와 윈도우 버전의 생성자 모두에서, *access*는 선택적 키워드 매개 변수로 지정될 수 있습니다. *access*는 `ACCESS_READ`, `ACCESS_WRITE` 또는 `ACCESS_COPY` 중 하나의 값을 받아, 읽기 전용, 동시 기록(write-through) 또는 쓸 때 복사(copy-on-write) 메모리를 각각 지정하거나, `ACCESS_DEFAULT`를 사용하여 *prot*로 위임합니다. *access*는 유닉스와 윈도우에서 모두 사용할 수 있습니다. *access*를 지정하지 않으면, 윈도우 mmap은 동시 기록(write-through) 매핑을 반환합니다. 세 가지 액세스 유형 모두에서 초기 메모리값은 지정된 파일에서 가져옵니다. `ACCESS_READ` 메모리 맵에 대입하면 `TypeError` 예외가 발생합니다. `ACCESS_WRITE` 메모리 맵에 대입하면 메모리와 하부 파일에 모두 영향을 줍니다. `ACCESS_COPY` 메모리 맵에 대입하면 메모리에는 영향을 미치지만, 하부 파일은 변경되지 않습니다.

버전 3.7에서 변경: `ACCESS_DEFAULT` 상수가 추가되었습니다.

익명 메모리를 매핑하려면, length와 함께 -1을 fileno로 전달해야 합니다.

class `mmap.`**`mmap`**(*fileno, length, tagname=None, access=ACCESS_DEFAULT, offset=0*)

(윈도우 버전) 파일 핸들 *fileno*로 지정된 파일의 *length* 바이트를 매핑하고, mmap 객체를 만듭니다. *length*가 파일의 현재 크기보다 크면, 파일은 *length* 바이트를 포함하도록 확장됩니다. *length*가 `0` 이면, 맵의 최대 길이는 파일의 현재 길이입니다. 단, 파일이 비어 있으면 윈도우에서 예외가 발생합니다 (윈도우에는 빈 매핑을 만들 수 없습니다).

*tagname*가 지정되고 `None`이 아니면, 매핑의 태그 이름을 제공하는 문자열입니다. 윈도우에서는 같은 파일에 대해 여러 가지 다른 매핑을 사용할 수 있게 합니다. 기존 태그의 이름을 지정하면, 해당 태그가 열리고, 그렇지 않으면 이 이름의 새 태그가 만들어집니다. 이 매개 변수가 생략되거나 `None`이면, 매핑은 이름 없이 만들어집니다. *tagname* 매개 변수의 사용을 피하면 코드를 유닉스와 윈도우 사이에서 이식성 있게 유지할 수 있습니다.

*offset*은 음이 아닌 정수 오프셋으로 지정할 수 있습니다. mmap 참조는 파일 시작 부분으로부터의 오프셋에 상대적입니다. *offset*의 기본값은 0입니다. *offset*은 `ALLOCATIONGRANULARITY`의 배수여야 합니다.

인자 `fileno`, `length`, `access`, `offset`로 감사 이벤트 (*auditing event*) `mmap.__new__`를 발생시킵니다.

class `mmap.`**`mmap`**(*fileno, length, flags=MAP_SHARED, prot=PROT_WRITE | PROT_READ, access=ACCESS_DEFAULT, offset=0, *, trackfd=True*)

(유닉스 버전) 파일 기술자 *fileno*로 지정된 파일의 *length* 바이트를 매핑하고, mmap 객체를 반환합니다. *length*가 `0` 이면, 맵의 최대 길이는 `mmap`가 호출될 때 파일의 현재 길이입니다.

*flags*는 매핑의 특성을 지정합니다. `MAP_PRIVATE`는 비공개 쓸 때 복사 (copy-on-write) 매핑을 생성하므로, mmap 객체의 내용에 대한 변경 사항은 이 프로세스에만 적용되고, `MAP_SHARED`는 파일의 같은 영역을 매핑하는 다른 모든 프로세스와 공유되는 매핑을 만듭니다. 기본값은 `MAP_SHARED`입니다. 일부 시스템에는 *MAP_* 상수에 지정된 전체 목록과 함께 추가적으로 가능한 플래그가 있습니다.

*prot*가 지정되면 원하는 메모리 보호를 제공합니다; 가장 유용한 두 값은 페이지를 읽거나 쓰도록 지정할 수 있는 `PROT_READ`와 `PROT_WRITE`입니다. *prot*의 기본값은 `PROT_READ | PROT_WRITE`입니다.

*access*는 선택적 키워드 매개 변수로 *flags* 와 *prot* 대신 지정 될 수 있습니다. *flags*, *prot* 와 *access*를 모두 지정하는 것은 에러입니다. 이 매개 변수를 사용하는 방법에 대한 정보는 위의 *access* 설명을 참조하십시오.

*offset*은 음이 아닌 정수 오프셋으로 지정할 수 있습니다. mmap 참조는 파일 시작 부분으로부터의 오프셋에 상대적입니다. *offset*의 기본값은 0입니다. *offset*은 유닉스 시스템에서 `PAGESIZE`와 같은 `ALLOCATIONGRANULARITY`의 배수여야 합니다.

If *trackfd* is `False`, the file descriptor specified by *fileno* will not be duplicated, and the resulting `mmap` object will not be associated with the map' s underlying file. This means that the `size()` and `resize()` methods will fail. This mode is useful to limit the number of open file descriptors.

생성된 메모리 매핑의 유효성을 보장하기 위해, 기술자 *fileno*로 지정된 파일은 맥 OS에서 물리적 저장 장치와 내부적으로 자동 동기화됩니다.

버전 3.13에서 변경: *trackfd* 매개 변수를 추가했습니다.

이 예제는 *mmap*을 사용하는 간단한 방법을 보여줍니다:

```
import mmap

# 간단한 예제 파일을 작성합니다
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # 파일을 메모리 맵 합니다, 크기 0은 전체 파일을 뜻합니다.
    mm = mmap.mmap(f.fileno(), 0)
    # 표준 파일 메서드로 내용을 읽습니다
    print(mm.readline())  # b"Hello Python!\n" 를 인쇄합니다
    # 슬라이스 표기법으로 내용을 읽습니다
    print(mm[:5])  # b"Hello" 를 인쇄합니다
    # 슬라이스 표기법을 사용하여 내용을 갱신합니다;
    # 새 내용의 크기가 같아야 합니다
    mm[6:] = b" world!\n"
    # ... 표준 파일 메서드로 다시 읽습니다
    mm.seek(0)
    print(mm.readline())  # b"Hello  world!\n" 를 인쇄합니다
    # 맵을 닫습니다
    mm.close()
```

*mmap*은 `with` 문에서 컨텍스트 관리자로 사용할 수도 있습니다:

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

Added in version 3.2: 컨텍스트 관리자 지원.

다음 예제는 익명 맵을 만들고 부모와 자식 프로세스 간에 데이터를 교환하는 방법을 보여줍니다:

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")

pid = os.fork()

if pid == 0:  # 자식 프로세스에서
    mm.seek(0)
    print(mm.readline())

    mm.close()
```

인자 `fileno`, `length`, `access`, `offset`로 감사 이벤트 (*auditing event*) `mmap.__new__`를 발생시킵니다.

메모리 맵 파일 객체는 다음 메서드를 지원합니다:

close()

mmap를 닫습니다. 이후에 객체의 다른 메서드를 호출하면 ValueError 예외가 발생합니다. 열려 있는 파일을 닫지 않습니다.

closed

파일이 닫혔으면 `True` 입니다.

Added in version 3.2.

find(*sub*[, *start*[, *end*]])

서브 시퀀스 *sub*가 발견되는 객체에서 가장 낮은 인덱스를 반환합니다. *sub*는 [*start*, *end*] 범위에 포함되어야 합니다. 선택적 인자 *start* 와 *end*는 슬라이스 표기법처럼 해석됩니다. 실패하면 `-1` 를 반환합니다.

버전 3.5에서 변경: 이제 쓰기 가능한 바이트열류 객체를 받아들입니다.

flush()

flush(*offset*, *size*, /)

파일의 메모리 내 복사본에 대한 변경 사항을 디스크로 플러시 합니다. 이 호출을 사용하지 않으면, 객체가 파괴되기 전에 변경 내용이 기록된다고 보장할 수 없습니다. *offset*과 *size*가 지정되면, 지정된 바이트 범위의 변경 사항만 디스크로 플러시 됩니다; 그렇지 않으면, 매핑의 전체 범위가 플러시 됩니다. *offset*은 `PAGESIZE` 나 `ALLOCATIONGRANULARITY`의 배수여야 합니다.

성공을 나타내기 위해 `None`이 반환됩니다. 호출이 실패하면 예외가 발생합니다.

버전 3.8에서 변경: 이전에는, 성공 시 0이 아닌 값이 반환되었습니다; 윈도우에서 에러 시 0이 반환되었습니다. 성공 시 0 값이 반환되었습니다; 유닉스에서 에러 시 예외가 발생했습니다.

madvise(*option*[, *start*[, *length*]])

*start*에서 시작하고 *length* 바이트만큼 확장하는 메모리 영역에 대해 커널에 조언 *option*을 보냅니다. *option*은 시스템에서 사용할 수 있는 *MADV_** 상수 중 하나여야 합니다. *start*와 *length*가 생략되면, 전체 매핑으로 확장됩니다. 일부 시스템(리눅스 포함)에서, *start*는 `PAGESIZE`의 배수여야 합니다.

가용성: `madvise()` 시스템 호출이 있는 시스템.

Added in version 3.8.

move(*dest*, *src*, *count*)

오프셋 *src*에서 시작하는 *count* 바이트를 대상 인덱스 *dest*로 복사합니다. mmap이 `ACCESS_READ` 로 만들어졌으면, move를 호출하면 `TypeError` 예외가 발생합니다.

read([*n*])

현재의 파일 위치로부터 최대 *n* 바이트를 포함하는 `bytes`를 반환합니다. 인자가 생략되거나 `None` 이거나 음수면, 현재 파일 위치에서 매핑의 끝까지 모든 바이트를 반환합니다. 파일 위치는 반환된 바이트의 뒤를 가리키도록 갱신됩니다.

버전 3.3에서 변경: 인자는 생략되거나 `None` 일 수 있습니다.

read_byte()

현재 파일 위치의 한 바이트를 정수로 반환하고, 파일 위치를 1 증가시킵니다.

readline()

현재 파일 위치에서 시작하여 다음 줄 바꿈까지 한 줄을 반환합니다. 반환된 바이트 뒤를 가리키도록 파일 위치가 갱신됩니다.

resize(*newsize*)

Resizes the map and the underlying file, if any.

Resizing a map created with *access* of `ACCESS_READ` or `ACCESS_COPY`, will raise a `TypeError` exception. Resizing a map created with *trackfd* set to `False`, will raise a `ValueError` exception.

On Windows: Resizing the map will raise an `OSError` if there are other maps against the same named file. Resizing an anonymous map (ie against the pagefile) will silently create a new map with the original data copied over up to the length of the new size.

버전 3.11에서 변경: Correctly fails if attempting to resize when another map is held Allows resize against an anonymous map on Windows

rfind(*sub*[, *start*[, *end*]])

서브 시퀀스 *sub*가 발견되는 객체에서 가장 높은 인덱스를 반환합니다. *sub*는 [*start*, *end*] 범위에 포함되어야 합니다. 선택적 인자 *start* 와 *end*는 슬라이스 표기법처럼 해석됩니다. 실패하면 `-1`를 반환합니다.

버전 3.5에서 변경: 이제 쓰기 가능한 바이트열류 객체를 받아들입니다.

seek(*pos*[, *whence*])

파일의 현재 위치를 설정합니다. *whence* 인자는 선택적이며 기본값은 `os.SEEK_SET` 또는 `0` (절대 파일 위치) 입니다; 다른 값은 `os.SEEK_CUR` 또는 `1` (현재 위치를 기준으로 seek) 과 `os.SEEK_END` 또는 `2` (파일의 끝을 기준으로 seek) 입니다.

버전 3.13에서 변경: Return the new absolute position instead of `None`.

seekable()

Return whether the file supports seeking, and the return value is always `True`.

Added in version 3.13.

size()

파일의 길이를 반환합니다. 메모리 매핑된 영역의 크기보다 클 수 있습니다.

tell()

파일 포인터의 현재 위치를 반환합니다.

write(*bytes*)

*bytes*의 바이트를 파일 포인터의 현재 위치에 있는 메모리에 기록하고 기록된 바이트 수를 반환합니다 (쓰기가 실패하면 *ValueError*가 발생하기 때문에 결코 `len(bytes)` 보다 작지 않습니다). 파일 위치는 기록된 바이트 뒤를 가리 키도록 갱신됩니다. mmap이 `ACCESS_READ`로 만들어졌으면, 기록할 때 *TypeError* 예외가 발생합니다.

버전 3.5에서 변경: 이제 쓰기 가능한 바이트열류 객체를 받아들입니다.

버전 3.6에서 변경: 이제 기록한 바이트 수가 반환됩니다.

write_byte(*byte*)

정수 *byte*를 파일 포인터의 현재 위치에 있는 메모리에 기록합니다; 파일 위치가 `1` 증가합니다. mmap이 `ACCESS_READ`로 만들어졌으면, 기록할 때 *TypeError* 예외가 발생합니다.

19.7.1 MADV_* 상수

mmap.**MADV_NORMAL**

mmap.**MADV_RANDOM**

mmap.**MADV_SEQUENTIAL**

mmap.**MADV_WILLNEED**

mmap.**MADV_DONTNEED**

mmap.**MADV_REMOVE**

mmap.**MADV_DONTFORK**

mmap.**MADV_DOFORK**

mmap.**MADV_HWPOISON**

mmap.**MADV_MERGEABLE**

mmap.**MADV_UNMERGEABLE**

mmap.**MADV_SOFT_OFFLINE**

mmap.**MADV_HUGEPAGE**

mmap.**MADV_NOHUGEPAGE**

mmap.**MADV_DONTDUMP**

mmap.**MADV_DODUMP**

mmap.**MADV_FREE**

mmap.**MADV_NOSYNC**

mmap.**MADV_AUTOSYNC**

mmap.**MADV_NOCORE**

mmap.**MADV_CORE**

mmap.**MADV_PROTECT**

mmap.**MADV_FREE_REUSABLE**

mmap.**MADV_FREE_REUSE**

이 옵션은 *mmap.madvise()*로 전달될 수 있습니다. 모든 시스템에서 모든 옵션이 제공되는 것은 아닙니다.

가용성: madvise() 시스템 호출이 있는 시스템.

Added in version 3.8.

19.7.2 MAP_* 상수

mmap.**MAP_SHARED**

mmap.**MAP_PRIVATE**

mmap.**MAP_32BIT**

mmap.**MAP_ALIGNED_SUPER**

mmap.**MAP_ANON**

mmap.**MAP_ANONYMOUS**

mmap.**MAP_CONCEAL**

mmap.**MAP_DENYWRITE**

mmap.**MAP_EXECUTABLE**

mmap.**MAP_HASSEMAPHORE**

mmap.**MAP_JIT**

mmap.**MAP_NOCACHE**

mmap.**MAP_NOEXTEND**

mmap.**MAP_NORESERVE**

mmap.**MAP_POPULATE**

mmap.**MAP_RESILIENT_CODESIGN**

mmap.**MAP_RESILIENT_MEDIA**

mmap.**MAP_STACK**

mmap.**MAP_TPRO**

mmap.**MAP_TRANSLATED_ALLOW_EXECUTE**

mmap.**MAP_UNIX03**

These are the various flags that can be passed to *mmap.mmap()*. *MAP_ALIGNED_SUPER* is only available at FreeBSD and *MAP_CONCEAL* is only available at OpenBSD. Note that some options might not be present on some systems.

버전 3.10에서 변경: *MAP_POPULATE* 상수가 추가되었습니다.

Added in version 3.11: *MAP_STACK* 상수가 추가되었습니다.

Added in version 3.12: *MAP_ALIGNED_SUPER* 와 *MAP_CONCEAL* 상수를 추가했습니다.

Added in version 3.13: *MAP_32BIT*, *MAP_HASSEMAPHORE*, *MAP_JIT*, *MAP_NOCACHE*, *MAP_NOEXTEND*, *MAP_NORESERVE*, *MAP_RESILIENT_CODESIGN*, *MAP_RESILIENT_MEDIA*, *MAP_TPRO*, *MAP_TRANSLATED_ALLOW_EXECUTE* 및 *MAP_UNIX03* 상수를 추가했습니다.

CHAPTER 20

인터넷 데이터 처리

이 장에서는 인터넷에서 일반적으로 사용되는 데이터 형식 처리를 지원하는 모듈에 관해 설명합니다.

20.1 email — 전자 메일과 MIME 처리 패키지

소스 코드: Lib/email/__init__.py

email 패키지는 전자 메일 메시지를 관리하기 위한 라이브러리입니다. 특히 SMTP (**RFC 2821**), NNTP 또는 다른 서버로 전자 메일 메시지를 보내도록 설계되지 않았습니다; 그런 것들은 smtplib 같은 모듈의 기능입니다. email 패키지는 **RFC 5322**와 **RFC 6532**뿐만 아니라, **RFC 2045**, **RFC 2046**, **RFC 2047**, **RFC 2183** 및 **RFC 2231**와 같은 MIME 관련 RFC를 지원하여 가능한 최대로 RFC를 준수하려고 시도합니다.

email 패키지의 전체 구조는 세 가지 주요 구성 요소와 다른 구성 요소의 동작을 제어하는 네 번째 구성 요소로 나눌 수 있습니다.

패키지의 중심 구성 요소는 전자 메일 메시지를 나타내는 "객체 모델" 입니다. 응용 프로그램은 주로 message 서브 모듈에 정의된 객체 모델 인터페이스를 통해 패키지와 상호 작용합니다. 응용 프로그램은 이 API를 사용하여 기존 전자 메일에 대해 질문을 하거나, 새 전자 메일을 작성하거나, 같은 객체 모델 인터페이스를 사용하는 전자 메일 하위 구성 요소를 추가하거나 제거할 수 있습니다. 즉, 전자 메일 메시지와 MIME 하위 구성 요소의 특성에 따라, 전자 메일 객체 모델은 모두 EmailMessage API를 제공하는 객체의 트리 구조입니다.

패키지의 다른 두 가지 주요 구성 요소는 parser와 generator입니다. 구문 분석기(parser)는 직렬화된 전자 메일 메시지(바이트 스트림)를 가져와 EmailMessage 객체의 트리로 변환합니다. 생성기(generator)는 EmailMessage를 받아서 직렬화된 바이트 스트림으로 다시 변환합니다. (구문 분석기와 생성기는 텍스트 문자의 스트림도 처리하지만, 이 사용법은 유효하지 않은 메시지로 끝나기 쉬우므로 사용하지 않는 것이 좋습니다.)

제어 구성 요소는 policy 모듈입니다. 모든 EmailMessage, 모든 generator 및 모든 parser에는 그것의 동작을 제어하는 연관된 policy 객체가 있습니다. 일반적으로 응용 프로그램은 EmailMessage가 만들어질 때 정책을 지정하기만 하면 되는데, EmailMessage를 직접 인스턴스로 만들어서 새 전자 메일을 만들거나, parser를 사용하여 입력 스트림을 구문 분석할 때입니다. 그러나 메시지가 generator를 사용하여 직렬화될 때 정책을 변경할 수 있습니다. 이것은, 예를 들어, 범용 전자 메일 메시지를 디스크에서 구문 분석하지만, 전자 메일 서버로 보낼 때 표준 SMTP 설정을 사용하여 직렬화할 수 있도록 합니다.

email 패키지는 응용 프로그램으로부터 각종 관리적인 RFC의 세부 사항을 숨기기 위해 최선을 다합니다. 개념적으로 응용 프로그램은 전자 메일 메시지를 유니코드 텍스트와 바이너리 첨부 파일의 구조화 된 트리로 처리할 수 있어야 하며, 직렬화될 때 이것들이 어떻게 표시되는지 걱정할 필요가 없어야 합니다.

하지만, 실제로는, MIME 메시지와 그 구조, 특히 MIME "콘텐츠 형식(content type)"의 이름과 특성, 그리고 다중 부분 문서를 식별하는 방법을 관리하는 규칙 중 적어도 일부를 신경 쓸 필요가 종종 있습니다. 대부분, 이 지식은 더욱 복잡한 응용 프로그램에만 필요하며, 그럴 때도 그 구조가 어떻게 표현되는지에 대한 세부 사항이 아닌, 문제가 되는 고수준 구조에 관한 것이어야 합니다. MIME 콘텐츠 유형은 최신 인터넷 소프트웨어(전자 메일뿐만 아니라)에서 널리 사용되므로, 많은 프로그래머에게 익숙한 개념입니다.

다음 절에서는 `email` 패키지의 기능에 관해 설명합니다. 응용 프로그램에서 사용할 기본 인터페이스인 `message` 객체 모델부터 시작하여, `parser`와 `generator` 구성 요소를 다룹니다. 그런 다음 `policy` 제어를 다뤄서, 라이브러리의 주요 구성 요소를 마무리합니다.

다음 세 절에서는 패키지에서 발생할 수 있는 예외와 `parser`가 감지할 수 있는 결함(RFC를 준수하지 않는)에 관해 설명합니다. 그런 다음 `headerregistry`와 `contentmanager` 하위 구성 요소를 다룹니다. 이것들은 각각 헤더와 페이로드를 보다 자세하게 조작할 수 있는 도구를 제공합니다. 이 두 구성 요소는 모두 단순하지 않은 메시지를 소비하고 생성하는 것과 관련된 기능을 포함하지만, 고급 응용 프로그램이 관심을 가질 확장 API를 설명하기도 합니다.

그다음은 이전 절에서 다룬 API의 기본 부분들을 사용하는 일련의 예제입니다.

앞의 내용은 email 패키지의 최신(유니코드 친화적인) API를 나타냅니다. `Message` 클래스로 시작하는 나머지 절에서는 전자 메일 메시지가 표현되는 방법에 대한 세부 사항을 훨씬 직접 다루는 레거시 `compat32` API를 다룹니다. `compat32` API는 응용 프로그램으로부터 RFC 세부 사항을 숨기지 않습니다만, 그 수준에서 작동해야 하는 응용 프로그램에는 유용한 도구가 될 수 있습니다. 이 설명서는 과거 호환성을 위해 여전히 `compat32` API를 사용하는 응용 프로그램과도 관련이 있습니다.

버전 3.6에서 변경: 새로운 `EmailMessage`/`EmailPolicy` API를 홍보하기 위해 설명서가 재구성되고 다시 작성되었습니다.

`email` 패키지 설명서의 목차:

20.1.1 email.message: 전자 메일 메시지 표현

소스 코드: Lib/email/message.py

Added in version 3.6:[1]

`email` 패키지의 중심 클래스는 `email.message` 모듈에서 임포트 되는 `EmailMessage` 클래스입니다. `email` 객체 모델의 베이스 클래스입니다. `EmailMessage`는 헤더 필드 설정과 조회, 메시지 본문 액세스 및 구조화된 메시지 작성이나 수정을 위한 핵심 기능을 제공합니다.

전자 메일 메시지는 헤더(*headers*)와 페이 로드(*payload*)(내용(*content*)이라고도 합니다)로 구성됩니다. 헤더는 **RFC 5322**나 **RFC 6532** 스타일 필드 이름과 값이며, 필드 이름과 값은 콜론으로 구분됩니다. 콜론은 필드 이름이나 필드 값의 일부가 아닙니다. 페이 로드는 간단한 텍스트 메시지, 바이너리 객체 또는 각각 자신만의 헤더 집합과 자신만의 페이 로드를 갖는 서브 메시지들의 구조화된 시퀀스일 수 있습니다. 마지막 유형의 페이 로드는 `multipart/*`나 `message/rfc822`와 같은 MIME 유형을 가진 메시지로 표시됩니다.

`EmailMessage` 객체가 제공하는 개념적 모델은 메시지의 **RFC 5322** 본문을 나타내는 *payload*와 결합 된 헤더들의 순서 있는 딕셔너리이며, 본문은 서브-`EmailMessage` 객체의 리스트일 수 있습니다. 헤더 이름과 값에 액세스하기 위한 일반적인 딕셔너리 메서드 외에도, 헤더에서 특수 정보(예를 들어 MIME 콘텐츠 유형)에 액세스하고, 페이 로드를 다루고, 메시지의 직렬화된 버전을 생성하고, 객체 트리를 재귀적으로 탐색하는 메서드가 있습니다.

`EmailMessage` 딕셔너리류 인터페이스는 ASCII 값이어야 하는 헤더 이름으로 인덱싱됩니다. 딕셔너리의 값은 몇 가지 추가 메서드가 있는 문자열입니다. 헤더는 대소 문자를 유지하면서 저장되고 반환되지만, 필드 이름은 대소 문자를 구분하지 않고 일치합니다. 키 순서가 있습니다. 하지만 실제 딕셔너리와 달리 중복된 키를 가질 수 있습니다. 중복 키가 있는 헤더로 작업하기 위한 추가 메서드가 제공됩니다.

페이 로드는 간단한 메시지 객체의 경우 문자열이나 바이트열 객체이고, `multipart/*`와 `message/rfc822` 메시지 객체와 같은 MIME 컨테이너 문서에서는 `EmailMessage` 객체의 리스트입니다.

[1] 원래 3.4에서 잠정 모듈로 추가되었습니다. 레거시 메시지 클래스를 위한 설명서는 *email.message.Message: compat32 API를 사용하여 이메일 메시지 표현하기*로 옮겼습니다.

class email.message.**EmailMessage**(*policy=default*)

*policy*가 지정되면, 그것이 지정하는 규칙을 사용하여 메시지 표현을 갱신하고 직렬화합니다. *policy*가 설정되지 않으면, 줄 종료를 제외하고는 전자 메일 RFC 규칙을 따르는 *default* 정책을 사용합니다 (RFC가 요구하는 \r\n 대신에, 파이썬 표준 \n 줄 종료를 사용합니다). 자세한 내용은 *policy* 설명서를 참조하십시오.

as_string(*unixfrom=False, maxheaderlen=None, policy=None*)

전체 메시지를 평평하게 만든 문자열을 반환합니다. 선택적 *unixfrom*이 참이면, 봉투 헤더 (envelope header)가 반환된 문자열에 포함됩니다. *unixfrom*의 기본값은 False입니다. 베이스 *Message* 클래스와의 과거 호환성을 위해 *maxheaderlen*이 허용되지만, 기본값은 None이고, 이는 기본적으로 줄 길이가 정책의 *max_line_length*에 의해 제어됨을 의미합니다. *policy* 인자는 메시지 인스턴스에서 얻은 기본 정책을 대체하는 데 사용될 수 있습니다. 지정된 *policy*가 *Generator*로 전달되기 때문에, 메서드가 생성하는 포매팅의 일부를 제어하는 데 사용할 수 있습니다.

문자열로의 변환을 완료하기 위해 기본값을 채워야 하면 메시지를 평평하게 만들 때 *EmailMessage*에 대한 변경이 발생할 수 있습니다 (예를 들어, MIME 경계(boundaries)가 생성되거나 수정될 수 있습니다).

이 메서드는 편의상 제공되며, 응용 프로그램에서 메시지를 직렬화하는 가장 유용한 방법은 아닐 수 있습니다, 특히 여러 메시지를 다룬다면 그렇습니다. 메시지 직렬화를 위한 더 유연한 API는 *email.generator.Generator*를 참조하십시오. 또한 이 메서드는 *utf8*가 False(기본값)일 때 "7비트 클린"으로 직렬화된 메시지를 생성하는 것으로 제한됩니다.

버전 3.6에서 변경: *maxheaderlen*이 지정되지 않았을 때의 기본 동작은 기본값을 0으로 하는 것에서 정책(policy)의 *max_line_length* 값을 기본값으로 사용하는 것으로 변경되었습니다.

__str__()

as_string(policy=self.policy.clone(utf8=True))와 동등합니다. str(msg)가 직렬화된 메시지를 포함하는 문자열을 읽을 수 있는 형식으로 생성할 수 있도록 합니다.

버전 3.4에서 변경: 이 메서드는 utf8=True를 사용하도록 변경되어, *as_string()*의 직접적인 별칭인 대신, **RFC 6531**과 유사한 메시지 표현을 생성합니다.

as_bytes(*unixfrom=False, policy=None*)

전체 메시지를 평평하게 만든 바이트열 객체를 반환합니다. 선택적 *unixfrom*이 참이면, 봉투 헤더(envelope header)가 반환된 문자열에 포함됩니다. *unixfrom*의 기본값은 False입니다. *policy* 인자는 메시지 인스턴스에서 얻은 기본 정책을 대체하는 데 사용될 수 있습니다. 지정된 *policy*가 *BytesGenerator*로 전달되기 때문에, 메서드가 생성하는 포매팅의 일부를 제어하는 데 사용할 수 있습니다.

문자열로의 변환을 완료하기 위해 기본값을 채워야 하면 메시지를 평평하게 만들 때 *EmailMessage*에 대한 변경이 발생할 수 있습니다 (예를 들어, MIME 경계(boundaries)가 생성되거나 수정될 수 있습니다).

이 메서드는 편의상 제공되며, 응용 프로그램에서 메시지를 직렬화하는 가장 유용한 방법은 아닐 수 있습니다, 특히 여러 메시지를 다룬다면 그렇습니다. 메시지 직렬화를 위한 더 유연한 API는 *email.generator.BytesGenerator*를 참조하십시오.

__bytes__()

*as_bytes()*와 동등합니다. bytes(msg)가 직렬화된 메시지를 포함하는 바이트열 객체를 생성할 수 있도록 합니다.

is_multipart()

메시지의 페이로드가 서브-*EmailMessage* 객체의 리스트면 True를, 그렇지 않으면 False를 반환합니다. *is_multipart()*가 False를 반환할 때, 페이로드는 문자열 객체(CTE 인코딩된 바이너리 페이로드일 수 있습니다)여야 합니다. True를 반환하는 *is_multipart()*가 반드시 "msg.get_content_maintype() == 'multipart '"가 True를 반환한다는 것을 의미하지는 않습니다. 예를 들어, *EmailMessage*가 message/rfc822 유형일 때 is_multipart는 True를 반환합니다.

set_unixfrom(*unixfrom*)

메시지의 봉투 헤더(envelope header)를 문자열이어야 하는 *unixfrom*으로 설정합니다. (이 헤더에 대한 간단한 설명은 *mboxMessage*를 참조하십시오.)

get_unixfrom()

메시지의 봉투 헤더(envelope header)를 반환합니다. 봉투 헤더가 설정되지 않았으면 기본값은 `None`입니다.

다음 메서드는 메시지 헤더에 액세스하기 위한 매핑류 인터페이스를 구현합니다. 이 메서드들과 일반 매핑(즉, 딕셔너리) 인터페이스 사이에는 의미상 차이가 있습니다. 예를 들어, 딕셔너리에는 중복 키가 없지만, 여기서는 중복 메시지 헤더가 있을 수 있습니다. 또한 딕셔너리에서는 *keys()*가 반환한 키의 순서가 보장되지 않지만, *EmailMessage* 객체에서는 헤더가 항상 원래 메시지에 나타난 순서대로, 또는 나중에 메시지에 추가된 순서대로 반환됩니다. 삭제한 후 다시 추가된 헤더는 항상 헤더 리스트의 끝에 추가됩니다.

이러한 의미적 차이는 의도적이며 가장 흔한 사용 사례에서의 편의를 추구하는 쪽으로 기울어져 있습니다.

모든 경우에, 메시지에 존재하는 봉투 헤더는 매핑 인터페이스에 포함되지 않습니다.

__len__()

중복을 포함하여, 총 헤더 수를 반환합니다.

__contains__(*name*)

메시지 객체에 *name*이라는 필드가 있으면 `True`를 반환합니다. 대소 문자를 구분하지 않고 일치하며, *name*은 후행 콜론을 포함하지 않습니다. `in` 연산자에 사용됩니다. 예를 들면:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)

이름으로 지정된 헤더 필드의 값을 반환합니다. *name*은 콜론 필드 구분자를 포함하지 않습니다. 헤더가 없으면 `None`이 반환됩니다; *KeyError*가 발생하지 않습니다.

이름이 지정된 필드가 메시지 헤더에 두 번 이상 나타나면, 해당 필드 값 중 정확히 어떤 필드 값이 반환되는지 정의되지 않습니다. *get_all()* 메서드를 사용하여 *name*으로 이름이 지정된 모든 기존 헤더의 값을 가져오십시오.

표준 (`compat32`가 아닌) 정책을 사용하여, 반환된 값은 *email.headerregistry.BaseHeader*의 서브 클래스 인스턴스입니다.

__setitem__(*name*, *val*)

필드 이름이 *name*이고 값이 *val*인 헤더를 메시지에 추가합니다. 필드는 메시지의 기존 헤더들 끝에 추가됩니다.

같은 이름을 가진 기존 헤더를 덮어쓰거나 삭제하지 않습니다. 새 헤더가 메시지에서 필드 이름이 *name*인 유일한 것이 되도록 하려면, 먼저 필드를 삭제하십시오. 예를 들어:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

정책이 (표준 정책에서처럼) 특정 헤더를 고유한(unique) 것으로 정의하면, 이 메서드는 헤더가 이미 존재할 때 해당 헤더에 값을 할당하려고 시도하면 *ValueError*를 발생시킬 수 있습니다. 이 동작은 일관성을 위해 의도적이지만, 향후 이러한 대입이 기존 헤더를 자동으로 삭제하도록 선택할 수 있기 때문에, 이것에 의존하지 마십시오.

__delitem__(*name*)

메시지 헤더에서 이름이 *name*인 모든 필드를 삭제합니다. 해당 이름의 필드가 헤더에 없어도 예외가 발생하지 않습니다.

keys()

메시지의 모든 헤더 필드 이름의 리스트를 반환합니다.

values()

메시지의 모든 필드 값의 리스트를 반환합니다.

items()

메시지의 모든 필드 헤더와 값을 담은 2-튜플의 리스트를 반환합니다.

get(*name*, *failobj=None*)

명명된 헤더 필드의 값을 반환합니다. 명명된 헤더가 없을 때 선택적 *failobj*가 반환된다는 점을 제외하면 `__getitem__()`와 같습니다 (*failobj*의 기본값은 `None`).

추가적인 유용한 헤더 관련 메서드는 다음과 같습니다:

get_all(*name*, *failobj=None*)

*name*으로 명명된 필드의 모든 값의 리스트를 반환합니다. 메시지에 그런 이름의 헤더가 없으면 *failobj*가 반환됩니다 (기본값은 `None`).

add_header(*_name*, *_value*, ***_params*)

확장된 헤더 설정. 이 메서드는 추가 헤더 파라미터가 키워드 인자로 제공될 수 있다는 점을 제외하고는 *`__setitem__()`*과 유사합니다. *_name*은 추가할 헤더 필드이고 *_value*는 헤더의 기본(*primary*) 값입니다.

키워드 인자 딕셔너리 *_params*의 각 항목에 대해, 키는 파라미터 이름으로 사용되며 밑줄은 대시로 변환됩니다 (대시는 파이썬 식별자에서 유효하지 않기 때문입니다). 일반적으로, 값이 `None`이 아니면 파라미터가 `key="value"`로 추가되며, None이면 키만 추가됩니다.

값에 ASCII가 아닌 문자가 포함되면, 값을 `(CHARSET, LANGUAGE, VALUE)` 형식의 3-튜플로 지정하여 문자 집합과 언어를 명시적으로 제어 할 수 있습니다. 여기서 `CHARSET`은 값을 인코딩하는 데 사용할 문자 집합의 이름을 지정하는 문자열이고, `LANGUAGE`는 보통 `None`이나 빈 문자열 (다른 가능성은 **RFC 2231**을 참조하십시오)로 설정되고, `VALUE`는 비 ASCII 코드 포인트를 포함하는 문자열 값입니다. 3-튜플이 전달되지 않고 값에 ASCII가 아닌 문자가 포함되면, `CHARSET`으로 `utf-8`, `LANGUAGE`로 `None`을 사용하여 **RFC 2231** 형식으로 자동 인코딩됩니다.

예를 들면 다음과 같습니다:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

이것은 다음과 같은 헤더를 추가합니다

```
Content-Disposition: attachment; filename="bud.gif"
```

비 ASCII 문자가 있는 확장 인터페이스의 예:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

replace_header(*_name*, *_value*)

헤더를 교체합니다. 메시지에서 발견된 *_name*과 일치하는 첫 번째 헤더를 교체하고, 원래 헤더의 헤더 순서와 필드 이름 케이스(case)를 유지합니다. 일치하는 헤더가 없으면 *`KeyError`*를 발생시킵니다.

get_content_type()

`maintype/subtype` 형식의 소문자로 강제 변환된 메시지의 콘텐츠 유형을 반환합니다. 메시지에 *`Content-Type`* 헤더가 없으면 *`get_default_type()`*이 반환하는 값을 반환합니다. *`Content-Type`* 헤더가 유효하지 않으면 `text/plain`을 반환합니다.

(**RFC 2045**에 따라, 메시지는 항상 기본 유형을 가지며, *`get_content_type()`*은 항상 값을 반환합니다. **RFC 2045**는 *`multipart/digest`* 컨테이너 내에 등장하면 기본 유형이 *`message/rfc822`*이고, 그렇지 않으면 기본 유형을 *`text/plain`*으로 정의합니다. *`Content-Type`* 헤더가 유효하지 않은 유형 지정이면, **RFC 2045**는 기본 유형을 *`text/plain`*으로 강제합니다.)

get_content_maintype()

메시지의 메인 콘텐츠 유형을 반환합니다. 이것은 *`get_content_type()`*이 반환한 문자열의 *`maintype`* 부분입니다.

get_content_subtype()

메시지의 서브 콘텐츠 유형을 반환합니다. 이것은 *get_content_type()*이 반환한 문자열의 *subtype* 부분입니다.

get_default_type()

기본 콘텐츠 유형을 반환합니다. *multipart/digest* 컨테이너의 서브 파트인 메시지를 제외하고 대부분의 메시지는 기본 콘텐츠 유형이 *text/plain*입니다. 이러한 서브 파트는 기본 콘텐츠 유형이 *message/rfc822*입니다.

set_default_type(*ctype*)

기본 콘텐츠 유형을 설정합니다. *ctype*은 *text/plain*이나 *message/rfc822*여야 하지만, 이것을 강제하지는 않습니다. 기본 콘텐츠 유형은 *Content-Type* 헤더에 저장되지 않기 때문에, 메시지에 *Content-Type* 헤더가 없을 때 `get_content_type` 메서드의 반환 값에만 영향을 줍니다.

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)

Content-Type 헤더에 파라미터를 설정합니다. 파라미터가 이미 헤더에 존재하면, 해당 값을 *value*로 바꿉니다. *header*가 `Content-Type`(기본값)이고 헤더가 메시지에 아직 없으면, 헤더를 추가하고 값을 *text/plain*으로 설정한 다음 새 파라미터 값을 추가합니다. 선택적 *header*는 *Content-Type*의 대체 헤더를 지정합니다.

값에 ASCII가 아닌 문자가 포함되면, 선택적 *charset*과 *language* 매개 변수를 사용하여 문자 집합과 언어를 명시적으로 지정할 수 있습니다. 선택적 *language*는 **RFC 2231** 언어를 지정하며, 기본값은 빈 문자열입니다. *charset*과 *language*는 모두 문자열이어야 합니다. 기본값은 `utf8` *charset*과 `None` *language*를 사용하는 것입니다.

*replace*가 `False`(기본값)이면 헤더는 헤더 리스트의 끝으로 이동합니다. *replace*가 `True`이면, 헤더는 제자리에서 갱신됩니다.

EmailMessage 객체에 *requote* 매개 변수를 사용하는 것은 폐지되었습니다.

헤더의 기존 파라미터값은 헤더 값의 *params* 어트리뷰트를 통해 액세스 할 수 있음에 유의하십시오 (예를 들어, `msg['Content-Type'].params['charset']`).

버전 3.4에서 변경: `replace` 키워드가 추가되었습니다.

del_param(*param*, *header='content-type'*, *requote=True*)

Content-Type 헤더에서 지정된 파라미터를 완전히 제거합니다. 헤더는 해당 파라미터나 그 값 없이 제자리에서 다시 작성됩니다. 선택적 *header*는 *Content-Type*의 대체 헤더를 지정합니다.

EmailMessage 객체에 *requote* 매개 변수를 사용하는 것은 폐지되었습니다.

get_filename(*failobj=None*)

메시지의 *Content-Disposition* 헤더의 `filename` 파라미터값을 반환합니다. 헤더에 `filename` 파라미터가 없으면, 이 메서드는 *Content-Type* 헤더에서 `name` 파라미터를 찾는 것으로 폴백(fallback)합니다. 둘 다 없거나 헤더가 없으면, *failobj*가 반환됩니다. 반환된 문자열은 항상 *email.utils.unquote()*에 따라 unquote 됩니다.

get_boundary(*failobj=None*)

메시지의 *Content-Type* 헤더의 `boundary` 파라미터값, 또는 헤더가 없거나 `boundary` 파라미터가 없으면 *failobj*를 반환합니다. 반환된 문자열은 항상 *email.utils.unquote()*에 따라 unquote 됩니다.

set_boundary(*boundary*)

Content-Type 헤더의 `boundary` 파라미터를 *boundary*로 설정합니다. 필요하면 *set_boundary()*는 항상 *boundary*를 quote 합니다. 메시지 객체에 *Content-Type* 헤더가 없으면 *HeaderParseError*가 발생합니다.

*set_boundary()*가 헤더 리스트에서 *Content-Type* 헤더의 순서를 유지하기 때문에, 이 메서드를 사용하는 것은 이전 *Content-Type* 헤더를 삭제하고 *add_header()*를 통해 새 boundary로 새 헤더를 추가하는 것과는 미묘한 차이가 있음에 유의하십시오.

get_content_charset(*failobj=None*)

Content-Type 헤더의 charset 파라미터를 소문자로 강제 변환하여 반환합니다. *Content-Type* 헤더가 없거나 헤더에 charset 파라미터가 없으면 *failobj*가 반환됩니다.

get_charsets(*failobj=None*)

메시지 내의 문자 집합 이름들을 포함하는 리스트를 반환합니다. 메시지가 *multipart*이면, 리스트는 페이로드의 각 서브 파트마다 하나의 요소를 포함하며, 그렇지 않으면 길이 1인 리스트가 됩니다.

리스트의 각 항목은 표현된 서브 파트에 대한 *Content-Type* 헤더의 charset 파라미터의 값인 문자열입니다. 서브 파트에 *Content-Type* 헤더가 없거나, charset 파라미터가 없거나, *text* 메인 MIME 유형이 아니면, 반환된 리스트의 해당 항목은 *failobj*입니다.

is_attachment()

Content-Disposition 헤더가 있고 (대소 문자를 구분하지 않는) 값이 attachment이면 True를, 그렇지 않으면 False를 반환합니다.

버전 3.4.2에서 변경: *is_multipart()*와 일관성을 유지하기 위해, is_attachment는 이제 프로퍼티 대신 메서드입니다.

get_content_disposition()

메시지의 *Content-Disposition* 헤더가 있으면 소문자로 변환된 (파라미터 없는) 값을, 그렇지 않으면 None을 반환합니다. 메시지가 **RFC 2183**을 따르면, 이 메서드의 가능한 값은 *inline*, *attachment* 또는 None입니다.

Added in version 3.5.

다음 메서드는 메시지의 내용(페이로드)을 조사하고 조작하는 것과 관련이 있습니다.

walk()

walk() 메서드는 메시지 객체 트리의 모든 파트와 서브 파트를 깊이 우선 탐색 순서로 이터레이트 하는 데 사용할 수 있는 범용 제너레이터입니다. 일반적으로 *walk()*를 for 루프에서 이터레이터로 사용합니다; 각 이터레이션은 다음 서브 파트를 반환합니다.

다음은 멀티 파트 메시지 구조의 모든 파트의 MIME 유형을 인쇄하는 예입니다:

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

msg.get_content_maintype() == 'multipart'가 False를 반환하더라도, walk는 *is_multipart()*가 True를 반환하는 모든 파트의 서브 파트를 이터레이트 합니다. _structure 디버그 도우미 함수를 사용하여 예제에서 이를 확인할 수 있습니다:

```
>>> from email.iterators import _structure
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

여기서 message 파트는 multiparts는 아니지만, 서브 파트를 포함합니다. is_multipart() 는 True를 반환하고 walk는 서브 파트로 내려갑니다.

get_body(*preferencelist=('related', 'html', 'plain')*)

메시지의 "본문"이 될 수 있는 가장 적합한 후보인 MIME 파트를 반환합니다.

*preferencelist*는 related, html 및 plain 집합의 문자열 시퀀스이어야 하며, 반환된 파트의 콘텐츠 유형에 대한 선호 순서를 나타냅니다.

get_body 메서드가 호출된 객체와 일치하는 후보를 찾기 시작합니다.

related가 *preferencelist*에 포함되지 않으면, (서브) 파트가 선호와 일치하면 후보로 만난 모든 관련(related)의 루트 파트(또는 루트 파트의 서브 파트)를 고려합니다.

multipart/related와 만날 때, start 파라미터를 확인하고 일치하는 *Content-ID*가 있는 파트를 발견하면, 일치하는 후보를 찾을 때만 고려합니다. 그렇지 않으면 multipart/related의 첫 번째 (기본 루트) 파트만 고려합니다.

파트에 *Content-Disposition* 헤더가 있으면, 헤더 값이 inline일 때만 해당 파트를 후보로 간주합니다.

*preferencelist*의 선호 어느 것과도 일치하는 후보가 없으면, None을 반환합니다.

참고: (1) 대부분의 응용 프로그램에서 실제로 의미가 있는 *preferencelist* 조합은 ('plain',), ('html', 'plain') 및 기본 ('related', 'html', 'plain')뿐입니다. (2) get_body가 호출되는 객체에서 일치가 시작되므로, *preferencelist*가 기본값이면 multipart/related에서 get_body를 호출하면 객체 자신을 반환합니다. (3) *Content-Type*을 지정하지 않거나 *Content-Type* 헤더가 유효하지 않은 메시지(또는 메시지 파트)는 마치 text/plain 유형인 것처럼 처리되어, 간혹 get_body가 예기치 않은 결과를 반환하도록 합니다.

iter_attachments()

"본문" 파트 후보가 아닌 메시지의 모든 직접적인 서브 파트에 대한 이터레이터를 반환합니다. 즉, text/plain, text/html, multipart/related 또는 multipart/alternative 각각의 첫 번째 등장을 건너뛰고 (*Content-Disposition: attachment*를 통해 첨부 파일로 명시적으로 표시되지 않은 한), 나머지 모든 파트를 반환합니다. multipart/related에 직접 적용될 때, 루트 파트(즉: start 파라미터가 가리키는 파트나 start 파라미터가 없거나 start 파라미터가 파트들의 *Content-ID*와 일치하지 않으면 첫 번째 파트)를 제외한 모든 관련 파트에 대한 이터레이터를 반환합니다. multipart/alternative 또는 비-multipart에 직접 적용되면 빈 이터레이터를 반환합니다.

iter_parts()

메시지의 모든 직계 서브 파트에 대한 이터레이터를 반환합니다. multipart가 아니면 비어있게 됩니다. (*walk()*도 참조하십시오.)

get_content(**args, content_manager=None, **kw*)

*content_manager*의 *get_content()* 메서드를 호출합니다. 추가 인자로 제공되는 인자나 키워드와 함께 self를 메시지 객체로 전달합니다. *content_manager*가 지정되지 않으면, 현재 *policy*가 지정하는 content_manager를 사용합니다.

set_content(**args, content_manager=None, **kw*)

*content_manager*의 *set_content()* 메서드를 호출합니다. 추가 인자로 제공되는 인자나 키워드와 함께 self를 메시지 객체로 전달합니다. *content_manager*가 지정되지 않으면, 현재 *policy*가 지정하는 content_manager를 사용합니다.

make_related(*boundary=None*)

비 multipart 메시지를 multipart/related 메시지로 변환합니다. 기존 *Content-* 헤더와 페이 로드를 multipart의 (새로운) 첫 파트로 옮깁니다. *boundary*가 지정되면, multipart에서 경계 문자열로 사용하고, 그렇지 않으면 필요할 때 (예를 들어, 메시지가 직렬화될 때) 경계가 자동으로 만들어지도록 둡니다.

make_alternative(*boundary=None*)

비 multipart나 multipart/related를 multipart/alternative로 변환합니다. 기존 *Content-* 헤더와 페이 로드를 multipart의 (새로운) 첫 파트로 옮깁니다. *boundary*가 지정되면, multipart에서 경계 문자열로 사용하고, 그렇지 않으면 필요할 때 (예를 들어, 메시지가 직렬화될 때) 경계가 자동으로 만들어지도록 둡니다.

make_mixed(*boundary=None*)

비 multipart, multipart/related 또는 multipart-alternative를 multipart/mixed로 변환합니다. 기존 *Content-* 헤더와 페이 로드를 multipart의 (새로운) 첫 파트로 옮깁니다. *boundary*가 지정되면, multipart에서 경계 문자열로 사용하고, 그렇지 않으면 필요할 때 (예를 들어, 메시지가 직렬화될 때) 경계가 자동으로 만들어지도록 둡니다.

add_related(**args, content_manager=None, **kw*)

메시지가 multipart/related이면, 새 메시지 객체를 만들고, 모든 인자를 그것의 *set_content()* 메서드에 전달하고, 그것을 multipart에 *attach()*합니다. 메시지가 multipart가 아니면, *make_related()*를 호출한 다음 위에서처럼 진행합니다. 메시지가 다른 유형의 multipart이면, *TypeError*를 발생시킵니다. *content_manager*가 지정되지 않으면, 현재 *policy*가 지정하는 content_manager를 사용합니다. 추가된 파트에 *Content-Disposition* 헤더가 없으면, 값 inline으로 추가합니다.

add_alternative(**args, content_manager=None, **kw*)

메시지가 multipart/alternative이면, 새 메시지 객체를 만들고, 모든 인자를 그것의 *set_content()* 메서드에 전달하고, 그것을 multipart에 *attach()*합니다. 메시지가 multipart가 아니거나 multipart/related이면, *make_alternative()*를 호출한 다음 위에서처럼 진행합니다. 메시지가 다른 유형의 multipart이면, *TypeError*를 발생시킵니다. *content_manager*가 지정되지 않으면, 현재 *policy*가 지정하는 content_manager를 사용합니다.

add_attachment(**args, content_manager=None, **kw*)

메시지가 multipart/mixed이면, 새 메시지 객체를 만들고, 모든 인자를 그것의 *set_content()* 메서드에 전달하고, 그것을 multipart에 *attach()*합니다. 메시지가 multipart가 아니거나, multipart/related나 multipart/alternative이면, *make_mixed()*를 호출한 다음 위에서처럼 진행합니다. *content_manager*가 지정되지 않으면, 현재 *policy*가 지정하는 content_manager를 사용합니다. 추가된 파트에 *Content-Disposition* 헤더가 없으면, 값 attachment로 추가합니다. 이 메서드는 content_manager에 적절한 옵션을 전달하여 명시적 첨부(*Content-Disposition: attachment*)와 inline 첨부(*Content-Disposition: inline*)에 모두 사용할 수 있습니다.

clear()

페이 로드와 모든 헤더를 제거합니다.

clear_content()

페이 로드와 모든 *!Content-* 헤더를 제거하고, 다른 모든 헤더는 원래 순서대로 그대로 둡니다.

EmailMessage 객체에는 다음과 같은 인스턴스 어트리뷰트가 있습니다:

preamble

MIME 문서의 형식은 헤더 다음의 빈 줄과 첫 번째 멀티 파트 경계 문자열 사이에 어떤 텍스트를 허용합니다. 일반적으로 이 텍스트는 표준 MIME 방어구를 벗어나기 때문에 MIME을 인식하는 메일 리더에서 보이지 않습니다. 그러나, 메시지의 원시 텍스트를 보거나, MIME을 인식하지 않는 리더에서 메시지를 볼 때 이 텍스트가 나타날 수 있습니다.

preamble 어트리뷰트는 MIME 문서에 있는 이 선행 방어구 밖 텍스트를 포함합니다. *Parser*가 헤더 다음이지만 첫 번째 경계 문자열 이전에 있는 어떤 텍스트를 발견하면, 이 텍스트를 메시지의 *preamble* 어트리뷰트에 대입합니다. *Generator*가 MIME 메시지의 일반 텍스트(plain

text) 표현을 기록할 때, 메시지가 *preamble* 어트리뷰트를 가진 것을 발견하면, 헤더와 첫 번째 경계 사이의 영역에 이 텍스트를 씁니다. 자세한 내용은 `email.parser`와 `email.generator`를 참조하십시오.

메시지 객체에 preamble이 없으면, *preamble* 어트리뷰트는 `None` 입니다.

epilogue

epilogue 어트리뷰트는 메시지의 마지막 경계와 끝 사이에 나타나는 텍스트를 포함한다는 점을 제외하고 *preamble* 어트리뷰트와 같은 방식으로 작동합니다. `preamble`과 마찬가지로 epilog 텍스트가 없으면, 이 어트리뷰트는 `None` 입니다.

defects

defects 어트리뷰트는 이 메시지를 구문 분석할 때 발견된 모든 문제점의 리스트를 포함합니다. 가능한 구문 분석 결함에 대한 자세한 설명은 `email.errors`를 참조하십시오.

class email.message.**MIMEPart**(*policy=default*)

이 클래스는 MIME 메시지의 서브 파트를 나타냅니다. 서브 파트에는 자체 `MIME-Version` 헤더가 필요하지 않아서, `set_content()`를 호출할 때 `MIME-Version` 헤더가 추가되지 않는다는 점을 제외하면 `EmailMessage`와 같습니다.

20.1.2 email.parser: 전자 메일 메시지 구문 분석

소스 코드: Lib/email/parser.py

메시지 객체 구조는 두 가지 방법으로 만들 수 있습니다: `EmailMessage` 객체를 만들고, 딕셔너리 인터페이스를 사용하여 헤더를 추가하고, `set_content()`와 관련 메서드를 사용하여 페이 로드를 추가하여 아예 새로 만들 수 있습니다. 또는 전자 메일 메시지의 직렬화된 표현을 구문 분석해서 만들 수 있습니다.

`email` 패키지는 MIME 문서를 포함한 대부분의 전자 우편 문서 구조를 이해하는 표준 구문 분석기를 제공합니다. 구문 분석기에 바이트열, 문자열 또는 파일 객체를 전달하면 구문 분석기가 객체 구조의 루트 `EmailMessage` 인스턴스를 반환합니다. MIME이 아닌 간단한 메시지의 경우 이 루트 객체의 페이 로드는 메시지의 텍스트를 포함하는 문자열일 수 있습니다. MIME 메시지의 경우 루트 객체는 `is_multipart()` 메서드가 `True`를 반환하고, `get_body()`, `iter_parts()` 및 `walk()`와 같은 페이 로드 조작 메서드를 통해 서브 파트에 액세스 할 수 있습니다.

실제로 사용 가능한 두 가지 구문 분석기 인터페이스가 있습니다, `Parser` API와 증분 `FeedParser` API. `Parser` API는 메시지의 전체 텍스트가 메모리에 있거나 전체 메시지가 파일 시스템의 파일에 있을 때 가장 유용합니다. 더 많은 입력을 기다리기 위해 블록할 수 있는 스트림에서 메시지를 읽을 때는 `FeedParser`가 더 적합합니다 (가령 소켓에서 전자 메일 메시지를 읽을 때). `FeedParser`는 메시지를 증분 적으로 소비하고 구문 분석 할 수 있으며, 구문 분석기를 닫을 때만 루트 객체를 반환합니다.

구문 분석기는 제한적인 방식으로 확장될 수 있음에 유의하십시오. 물론 구문 분석기는 처음부터 새로 구현할 수 있습니다. `email` 패키지에 포함된 구문 분석기와 `EmailMessage` 클래스를 연결하는 모든 로직은 `Policy` 클래스에 내장되므로, 사용자 정의 구문 분석기는 적절한 `Policy` 메서드의 사용자 정의 버전을 구현하여 필요한 방식으로 메시지 객체 트리를 만들 수 있습니다.

FeedParser API

`email.feedparser` 모듈에서 임포트 한 `BytesFeedParser`는 전자 메일 메시지의 증분 구문 분석에 도움이 되는 API를 제공합니다. 블록할 수 있는 소스(가령 소켓)에서 전자 메일 메시지의 텍스트를 읽을 때 필요합니다. 물론 `BytesFeedParser`는 바이트열류 객체, 문자열 또는 파일에 완전히 포함된 전자 메일 메시지를 구문 분석하는 데 사용될 수 있지만, `BytesParser` API가 이러한 사용 사례에서는 더 편리 할 수 있습니다. 두 구문 분석기 API의 의미와 결과는 같습니다.

`BytesFeedParser`의 API는 간단합니다; 인스턴스를 만들고, 더는 공급할 것이 없을 때까지 바이트열을 공급한 다음 구문 분석기를 닫아 루트 메시지 객체를 얻습니다. `BytesFeedParser`는 표준 호환 메시지를 구문 분석할 때 매우 정확하고, 미준수 메시지를 구문 분석하는 데 매우 효과적이며 메시지를 어떻게 손상되었다고 간주하는지에 대한 정보를 제공합니다. 메시지에서 발견된 문제점 리스트로 메시지 객체의 `defects` 어트리뷰트를 채웁니다. 찾을 수 있는 결함 목록은 `email.errors` 모듈을 참조하십시오.

`BytesFeedParser` API는 다음과 같습니다:

class email.parser.**BytesFeedParser**(*_factory=None, *, policy=policy.compat32*)

`BytesFeedParser` 인스턴스를 만듭니다. 선택적 *_factory*는 인자 없는 콜러블입니다; 지정되지 않으면 *policy*의 `message_factory`를 사용합니다. 새로운 메시지 객체가 필요할 때마다 *_factory*를 호출합니다.

*policy*가 지정되면, 그것이 지정하는 규칙을 사용하여 메시지 표시를 갱신합니다. *policy*가 설정되지 않으면, `compat32` 정책을 사용합니다. 이 정책은 파이썬 3.2 버전의 email 패키지와의 호환성을 유지하고 `Message`를 기본 팩토리로 제공합니다. 다른 모든 정책은 `EmailMessage`를 기본 *_factory*로 제공합니다. *policy*가 제어하는 다른 기능에 대한 자세한 내용은 `policy` 설명서를 참조하십시오.

참고: **policy 키워드는 항상 지정해야 합니다**; 이후 버전의 파이썬에서는 기본값이 `email.policy.default`로 변경됩니다.

Added in version 3.2.

버전 3.3에서 변경: *policy* 키워드를 추가했습니다.

버전 3.6에서 변경: *_factory*는 기본적으로 정책 `message_factory`입니다.

feed(*data*)

구문 분석기에 데이터를 더 공급합니다. *data*는 하나 이상의 줄을 포함하는 바이트열류 객체여야 합니다. 줄은 부분적일 수 있고 구문 분석기는 그러한 부분적인 줄을 올바르게 이어붙입니다. 줄은 세 가지 일반 줄 종료 중 어느 것이라도 될 수 있습니다: 캐리지 리턴(carriage return), 줄 바꿈(newline) 또는 캐리지 리턴과 줄 바꿈 (이것들을 혼합할 수도 있습니다).

close()

이전에 제공된 모든 데이터의 구문 분석을 완료하고 루트 메시지 객체를 반환합니다. 이 메서드가 호출된 후 `feed()`가 호출되면 어떻게 되는지는 정의되지 않습니다.

class email.parser.**FeedParser**(*_factory=None, *, policy=policy.compat32*)

Works like `BytesFeedParser` except that the input to the `feed()` method must be a string. This is of limited utility, since the only way for such a message to be valid is for it to contain only ASCII text or, if `utf8` is `True`, no binary attachments.

버전 3.3에서 변경: *policy* 키워드를 추가했습니다.

Parser API

`email.parser` 모듈에서 임포트 한 `BytesParser` 클래스는 메시지의 전체 내용이 바이트열류 객체나 파일로 있을 때 메시지를 구문 분석하는 데 사용할 수 있는 API를 제공합니다. `email.parser` 모듈은 또한 문자열 구문 분석을 위한 `Parser`와 메시지 헤더에만 관심이 있을 때 사용할 수 있는 헤더 전용 구문 분석기인 `BytesHeaderParser`와 `HeaderParser`를 제공합니다. `BytesHeaderParser`와 `HeaderParser`는 메시지 본문을 구문 분석하지 않고 페이 로드를 원시 본문으로 설정하기 때문에 이러한 상황에서 훨씬 더 빠를 수 있습니다.

class email.parser.**BytesParser**(*_class=None, *, policy=policy.compat32*)

`BytesParser` 인스턴스를 만듭니다. *_class*와 *policy* 인자는 `BytesFeedParser` 의 *_factory*와 *policy* 인자와 같은 의미입니다.

참고: **policy 키워드는 항상 지정해야 합니다**; 이후 버전의 파이썬에서는 기본값이 `email.policy.default`로 변경됩니다.

버전 3.3에서 변경: 2.4에서 폐지된 *strict* 인자를 제거했습니다. *policy* 키워드를 추가했습니다.

버전 3.6에서 변경: *_class*는 기본적으로 정책 `message_factory`입니다.

parse(*fp, headersonly=False*)

바이너리 파일류 객체 *fp*에서 모든 데이터를 읽고, 결과 바이트열을 구문 분석한 후, 메시지 객체를 반환합니다. *fp*는 `readline()`과 `read()` 메서드를 모두 지원해야 합니다.

The bytes contained in *fp* must be formatted as a block of **RFC 5322** (or, if `utf8` is `True`, **RFC 6532**) style headers and header continuation lines, optionally preceded by an envelope header. The header block is terminated either by the end of the data or by a blank line. Following the header block is the body of the message (which may contain MIME-encoded subparts, including subparts with a `Content-Transfer-Encoding` of `8bit`).

선택적 *headersonly*는 헤더를 읽은 후에 구문 분석을 중지할지를 지정하는 플래그입니다. 기본값은 `False`이며 파일의 전체 내용을 구문 분석합니다.

parsebytes(*bytes*, *headersonly=False*)

파일류 객체 대신 바이트열류 객체를 취한다는 점을 제외하고는 `parse()` 메서드와 유사합니다. 바이트열류 객체로 이 메서드를 호출하는 것은 `BytesIO` 인스턴스로 *bytes*를 먼저 감싸고 `parse()`를 호출하는 것과 동등합니다.

선택적 *headersonly*는 `parse()` 메서드와 같습니다.

Added in version 3.2.

class email.parser.**BytesHeaderParser**(*_class=None*, **, policy=policy.compat32*)

*headersonly*의 기본값이 `True`라는 점을 제외하고는 `BytesParser`와 정확히 같습니다.

Added in version 3.3.

class email.parser.**Parser**(*_class=None*, **, policy=policy.compat32*)

이 클래스는 `BytesParser`와 유사하지만, 문자열 입력을 처리합니다.

버전 3.3에서 변경: *strict* 인자를 제거했습니다. *policy* 키워드를 추가했습니다.

버전 3.6에서 변경: *_class*는 기본적으로 정책 `message_factory`입니다.

parse(*fp*, *headersonly=False*)

텍스트 모드 파일류 객체 *fp*에서 모든 데이터를 읽고, 결과 텍스트를 구문 분석한 후, 루트 메시지 객체를 반환합니다. *fp*는 파일류 객체의 `readline()`과 `read()` 메서드를 모두 지원해야 합니다.

텍스트 모드 요구 사항 외에, 이 메서드는 `BytesParser.parse()`처럼 작동합니다.

parsestr(*text*, *headersonly=False*)

파일류 객체 대신 문자열 객체를 취한다는 점을 제외하고는 `parse()` 메서드와 유사합니다. 문자열로 이 메서드를 호출하는 것은 `StringIO` 인스턴스로 *text*를 먼저 감싸고 `parse()`를 호출하는 것과 동등합니다.

선택적 *headersonly*는 `parse()` 메서드와 같습니다.

class email.parser.**HeaderParser**(*_class=None*, **, policy=policy.compat32*)

*headersonly*의 기본값이 `True`라는 점을 제외하고는 `Parser`와 정확히 같습니다.

문자열이나 파일 객체로부터 메시지 객체 구조를 만드는 것이 일반적인 작업이기 때문에, 편의상 4가지 함수가 제공됩니다. 최상위 `email` 패키지 이름 공간에 있습니다.

email.**message_from_bytes**(*s*, *_class=None*, **, policy=policy.compat32*)

바이트열류 객체로부터 메시지 객체 구조를 반환합니다. 이것은 `BytesParser().parsebytes(s)`와 동등합니다. 선택적 *_class*와 *policy*는 `BytesParser` 클래스 생성자에서처럼 해석됩니다.

Added in version 3.2.

버전 3.3에서 변경: *strict* 인자를 제거했습니다. *policy* 키워드를 추가했습니다.

email.**message_from_binary_file**(*fp*, *_class=None*, **, policy=policy.compat32*)

열린 바이너리 파일 객체로부터 메시지 객체 구조 트리를 반환합니다. 이것은 `BytesParser().parse(fp)`와 동등합니다. *_class*와 *policy*는 `BytesParser` 클래스 생성자에서처럼 해석됩니다.

Added in version 3.2.

버전 3.3에서 변경: *strict* 인자를 제거했습니다. *policy* 키워드를 추가했습니다.

email.**message_from_string**(*s*, *_class=None*, **, policy=policy.compat32*)

문자열로부터 메시지 객체 구조 트리를 반환합니다. 이것은 `Parser().parsestr(s)`와 동등합니다. *_class*와 *policy*는 `Parser` 클래스 생성자에서처럼 해석됩니다.

버전 3.3에서 변경: *strict* 인자를 제거했습니다. *policy* 키워드를 추가했습니다.

email.**message_from_file**(*fp, _class=None, *, policy=policy.compat32*)

열린 파일 객체로부터 메시지 객체 구조 트리를 반환합니다. 이것은 Parser().parse(fp) 와 동등합니다. *_class*와 *policy*는 Parser 클래스 생성자에서처럼 해석됩니다.

버전 3.3에서 변경: *strict* 인자를 제거했습니다. *policy* 키워드를 추가했습니다.

버전 3.6에서 변경: *_class*는 기본적으로 정책 message_factory 입니다.

대화식 파이썬 프롬프트에서 message_from_bytes() 를 사용하는 방법의 예는 다음과 같습니다:

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

추가 사항

구문 분석 의미에 대한 참고 사항은 다음과 같습니다:

- 대부분의 multipart 가 아닌 유형의 메시지는 문자열 페이 로드가 있는 단일 메시지 객체로 구문 분석됩니다. 이 객체는 is_multipart() 가 False 를 반환하고 iter_parts() 는 빈 목록을 산출합니다.
- 모든 multipart 유형 메시지는 서브 메시지 객체 리스트 페이 로드가 있는 컨테이너 메시지 객체로 구문 분석됩니다. 바깥 컨테이너 메시지는 is_multipart() 가 True 를 반환하고 iter_parts() 는 서브 파트 목록을 산출합니다.
- 콘텐츠 유형이 message/* (가령 message/delivery-status 와 message/rfc822) 인 대부분의 메시지는 길이가 1 인 리스트 페이 로드를 포함하는 컨테이너 객체로 구문 분석됩니다. is_multipart() 메서드는 True 를 반환합니다. iter_parts() 가 산출하는 단일 요소가 서브 메시지 객체입니다.
- 일부 표준을 준수하지 않는 메시지는 multipart 처리에 대해 내부적으로 일관성이 없을 수 있습니다. 이러한 메시지는 multipart 유형의 Content-Type 헤더를 가지면서도 is_multipart() 메서드가 False 를 반환할 수 있습니다. 이러한 메시지가 FeedParser 로 구문 분석되었다면, *defects* 어트리뷰트 리스트에 MultipartInvariantViolationDefect 클래스의 인스턴스가 있습니다. 자세한 내용은 email.errors 를 참조하십시오.

20.1.3 email.generator: MIME 문서 생성

소스 코드: Lib/email/generator.py

가장 일반적인 작업 중 하나는 메시지 객체 구조로 표현되는 전자 우편 메시지의 평평한 (직렬화된) 버전을 생성하는 것입니다. smtplib.SMTP.sendmail() 을 통해 메시지를 보내거나 콘솔에서 메시지를 인쇄하려면 이 작업을 수행해야 합니다. 메시지 객체 구조를 취하고 직렬화된 표현을 생성하는 것은 제너레이터 클래스의 작업입니다.

email.parser 모듈과 마찬가지로 번들 제너레이터의 기능으로 제한되지 않습니다; 처음부터 직접 작성할 수 있습니다. 그러나 번들 제너레이터는 표준 호환 방식으로 대부분 전자 우편을 생성하는 방법을 알고 있고, MIME과 비 MIME 전자 우편 메시지를 잘 처리하며, 변환 없는 같은 policy 가 사용된다고 가정할 때 바이트열 지향 구문 분석과 생성 연산이 역이 되도록 설계되었습니다. 즉, BytesParser 클래스로 직렬화된 바이트 스트림을 구문 분석한 다음, BytesGenerator 를 사용하여 직렬화된 바이트 스트림을 재생성하면 입력과 동일한 출력이 생성됩니다[1]. (반면에, 프로그램에서 구축한 EmailMessage 에 제너레이터를 사용하면 기본값이 채워지기 때문에 EmailMessage 객체가 변경될 수 있습니다.)

Generator 클래스를 사용하면 메시지를 (바이너리가 아닌) 텍스트 직렬화 표현으로 펼칠 수 있지만, 유니코드는 바이너리 데이터를 직접 표현할 수 없기 때문에, "8 비트 클린" 하지 않은 채널을 통한 전송을 위한 전자 우편 메시지를 인코딩하기 위한 표준 전자 우편 RFC 콘텐츠 전송 인코딩 (Content Transfer Encoding) 기술을 사용하여 메시지를 ASCII 문자만 포함된 것으로 변환해야 합니다.

[1] 이 문장은 unixfrom 에 적절한 설정을 사용하고, 자동 조정을 요구하는 email.policy 설정이 없다고 가정합니다 (예를 들어, refold_source 는 none 이어야 하며, 이는 기본값이 아닙니다). 메시지가 RFC 표준을 준수하지 않으면 때때로 구문 분석 에러 복구 중에 정확한 원본 텍스트에 대한 정보가 손실되므로 100% 사실이 아니기도 합니다. 가능하다면 이 후자의 경계 사례를 해결하는 것이 목표입니다.

SMIME 서명된 메시지의 재현성 있는 처리를 위해 *Generator*는 `multipart/signed` 유형의 메시지 파트와 모든 서브 부분에 대해 헤더 접기를 비활성화합니다.

class email.generator.**BytesGenerator**(*outfp, mangle_from_=None, maxheaderlen=None, *, policy=None*)

flatten() 메서드에 제공된 모든 메시지나 *write()* 메서드에 제공된 모든 서로게이트 이스케이프 인코딩된 텍스트를 파일류 객체 *outfp*에 쓰는 *BytesGenerator* 객체를 반환합니다. *outfp*는 바이너리 데이터를 받아들이는 `write` 메서드를 지원해야 합니다.

선택적인 *mangle_from_*이 `True`인 경우, 정확한 문자열 `"From "`으로 시작하는(즉 줄의 시작에 `From`이 오고 스페이스가 뒤따르는) 본문의 모든 줄 앞에 `>` 문자를 넣습니다. *mangle_from_*의 기본값은 *policy*의 *mangle_from_* 설정값입니다 (*compat32* 정책의 경우 `True`, 다른 모든 경우 `False`입니다). *mangle_from_*은 메시지가 유닉스 mbox 형식으로 저장될 때 사용하기 위한 것입니다 (*mailbox*와 WHY THE CONTENT-LENGTH FORMAT IS BAD를 참조하십시오).

*maxheaderlen*이 `None`이 아니면, *maxheaderlen*보다 긴 헤더 줄을 다시 접거나, `0`이면 헤더를 다시 접지 않습니다. *manheaderlen*이 `None`(기본값)이면, *policy* 설정에 따라 헤더와 기타 메시지 줄을 줄 바꿈합니다.

*policy*가 지정되면, 해당 정책을 사용하여 메시지 생성을 제어합니다. *policy*가 `None`(기본값)이면 `flatten`에 전달된 *Message*나 *EmailMessage* 객체와 연관된 정책을 사용하여 메시지 생성을 제어합니다. *policy*가 제어하는 것에 대한 자세한 내용은 *email.policy*를 참조하십시오.

Added in version 3.2.

버전 3.3에서 변경: *policy* 키워드를 추가했습니다.

버전 3.6에서 변경: *mangle_from_*과 *maxheaderlen* 매개 변수의 기본 동작은 정책을 따르는 것입니다.

flatten(*msg, unixfrom=False, linesep=None*)

*msg*를 루트로 하는 메시지 객체 구조의 텍스트 표현을 *BytesGenerator* 인스턴스가 만들어질 때 지정된 출력 파일에 인쇄합니다.

policy 옵션 *cte_type*이 `8bit`(기본값)이면, 하이 비트가 설정된 바이트들이 원본에서와같이 재생성되도록 출력이 수정되지 않은 원본 구문 분석된 메시지의 헤더를 복사하고, 비 ASCII *Content-Transfer-Encoding*을 이것을 갖는 모든 본문 파트에서 보존합니다. `cte_type`이 `7bit`이면, 하이 비트가 설정된 바이트들을 ASCII 호환 *Content-Transfer-Encoding*을 사용하여 필요에 따라 변환합니다. 즉, 비 ASCII *Content-Transfer-Encoding*(*Content-Transfer-Encoding: 8bit*)을 갖는 파트를 ASCII 호환 *Content-Transfer-Encoding*으로 변환하고, 헤더에 있는 RFC 유효하지 않은 비 ASCII 바이트를 MIME `unknown-8bit` 문자 집합을 사용하여 인코딩하여, RFC 호환되게 만듭니다.

*unixfrom*이 `True`이면, 루트 메시지 객체의 첫 번째 **RFC 5322** 헤더 앞에 유닉스 mailbox 형식 (*mailbox*를 참조하십시오)에서 사용되는 봉투 헤더 구분자를 인쇄합니다. 루트 객체에 봉투 헤더가 없으면, 표준 헤더를 만듭니다. 기본값은 `False`입니다. 서브 파트의 경우 봉투 헤더가 인쇄되지 않음에 유의하십시오.

*linesep*이 `None`이 아니면, 펼쳐진 메시지의 모든 줄 사이의 구분자 문자로 사용합니다. *linesep*이 `None`(기본값)이면, *policy*에 지정된 값을 사용합니다.

clone(*fp*)

정확히 같은 옵션 설정이고 *fp*를 새 *outfp*로 사용하는, 이 *BytesGenerator* 인스턴스의 독립 클론을 반환합니다.

write(*s*)

`ASCII` 코덱과 `surrogateescape` 에러 처리기를 사용하여 *s*를 인코딩하고, *BytesGenerator*의 생성자에 전달된 *outfp*의 *write* 메서드로 전달합니다.

편의상, *EmailMessage*는 *as_bytes()* 메서드와 `bytes(aMessage)`(일명 *__bytes__()*)를 제공하여 메시지 객체의 직렬화된 바이너리 표현 생성을 단순화합니다. 자세한 내용은 *email.message*를 참조하십시오.

문자열은 바이너리 데이터를 나타낼 수 없어서, *Generator* 클래스는 펼쳐지는 모든 메시지의 바이너리 데이터를 ASCII 호환 *Content-Transfer_Encoding*으로 변환하여 ASCII 호환 형식으로 변환해야 합니다. 전자 우편 RFC의 용어를 사용하면, 이를 "8비트 클린"이 아닌 I/O 스트림으로 직렬화하는 *Generator*로

생각할 수 있습니다. 즉, 대부분 응용 프로그램은 `Generator`가 아닌 `BytesGenerator`를 사용하려고 합니다.

class email.generator.**Generator**(*outfp, mangle_from_=None, maxheaderlen=None, *, policy=None*)

`flatten()` 메서드에 제공된 모든 메시지나 `write()` 메서드에 제공된 텍스트를 파일류 객체 *outfp*에 쓰는 `Generator` 객체를 반환합니다. *outfp*는 문자열 데이터를 받아들이는 `write` 메서드를 지원해야 합니다.

선택적인 *mangle_from_*이 `True`인 경우, 정확한 문자열 `"From "`으로 시작하는(즉 줄의 시작에 `From`이 오고 스페이스가 뒤따르는) 본문의 모든 줄 앞에 `>` 문자를 넣습니다. *mangle_from_*의 기본값은 *policy*의 `mangle_from_` 설정값입니다 (`compat32` 정책의 경우 `True`, 다른 모든 경우 `False`입니다). *mangle_from_*은 메시지가 유닉스 mbox 형식으로 저장될 때 사용하기 위한 것입니다 (`mailbox`와 WHY THE CONTENT-LENGTH FORMAT IS BAD를 참조하십시오).

*maxheaderlen*이 `None`이 아니면, *maxheaderlen*보다 긴 헤더 줄을 다시 접거나, `0`이면 헤더를 다시 접지 않습니다. *manheaderlen*이 `None`(기본값)이면, *policy* 설정에 따라 헤더와 기타 메시지 줄을 줄 바꿈합니다.

*policy*가 지정되면, 해당 정책을 사용하여 메시지 생성을 제어합니다. *policy*가 `None`(기본값)이면 `flatten`에 전달된 `Message`나 `EmailMessage` 객체와 연관된 정책을 사용하여 메시지 생성을 제어합니다. *policy*가 제어하는 것에 대한 자세한 내용은 `email.policy`를 참조하십시오.

버전 3.3에서 변경: *policy* 키워드를 추가했습니다.

버전 3.6에서 변경: *mangle_from_*과 *maxheaderlen* 매개 변수의 기본 동작은 정책을 따르는 것입니다.

flatten(*msg, unixfrom=False, linesep=None*)

*msg*를 루트로 하는 메시지 객체 구조의 텍스트 표현을 `Generator` 인스턴스가 만들어질 때 지정된 출력 파일에 인쇄합니다.

`policy` 옵션 `cte_type`이 `8bit`이면, 옵션이 `7bit`로 설정된 것처럼 메시지를 생성합니다. (문자열은 비 ASCII 바이트를 나타낼 수 없기 때문에 필요합니다.) 하이 비트가 설정된 모든 바이트를 ASCII 호환 `Content-Transfer-Encoding`을 사용하여 필요에 따라 변환합니다. 즉, 비 ASCII `Content-Transfer-Encoding`(`Content-Transfer-Encoding: 8bit`)을 갖는 파트를 ASCII 호환 `Content-Transfer-Encoding`으로 변환하고, 헤더에 있는 RFC 유효하지 않은 비 ASCII 바이트를 MIME `unknown-8bit` 문자 집합을 사용하여 인코딩하여, RFC 호환되게 만듭니다.

*unixfrom*이 `True`이면, 루트 메시지 객체의 첫 번째 **RFC 5322** 헤더 앞에 유닉스 mailbox 형식 (`mailbox`를 참조하십시오)에서 사용되는 봉투 헤더 구분자를 인쇄합니다. 루트 객체에 봉투 헤더가 없으면, 표준 헤더를 만듭니다. 기본값은 `False`입니다. 서브 파트의 경우 봉투 헤더가 인쇄되지 않음에 유의하십시오.

*linesep*이 `None`이 아니면, 펼쳐진 메시지의 모든 줄 사이의 구분자 문자로 사용합니다. *linesep*이 `None`(기본값)이면, *policy*에 지정된 값을 사용합니다.

버전 3.2에서 변경: `8bit` 메시지 본문을 다시 인코딩하기 위한 지원과 *linesep* 인자가 추가되었습니다.

clone(*fp*)

정확히 같은 옵션을 갖고 *fp*를 새 *outfp*로 사용하는, 이 `Generator` 인스턴스의 독립 클론을 반환합니다.

write(*s*)

`Generator`의 생성자에 전달된 *outfp*의 *write* 메서드로 *s*를 씁니다. 이것은 `print()` 함수에서 사용될 `Generator` 인스턴스를 위해 딱 필요한 만큼의 파일류 API를 제공합니다.

편의상, `EmailMessage`는 `as_string()` 메서드와 `str(aMessage)`(일명 `__str__()`)를 제공하여 메시지 객체의 포맷된 문자열 표현 생성을 단순화합니다. 자세한 내용은 `email.message`를 참조하십시오.

`email.generator` 모듈은 또한 파생 클래스인 `DecodedGenerator`를 제공하는데, `Generator` 베이스 클래스와 유사하지만, 비 `text` 파트는 직렬화되지 않고, 대신에 파트에 대한 정보로 채워진 템플릿에서 파생된 문자열로 출력 스트림에 표시됩니다.

class email.generator.**DecodedGenerator**(*outfp, mangle_from_=None, maxheaderlen=None, fmt=None, *, policy=None*)

`Generator`와 같이 작동하지만, `Generator.flatten()`에 전달된 메시지의 서브 파트에 대해, 서브 파트가 메인 유형이 `text`이면, 서브 파트의 디코딩된 페이 로드를 인쇄하고, 메인 유형이 `text`가 아니면, 그것을 인쇄하지 않고 파트 정보를 사용하여 문자열 *fmt*를 채운 후에 그 문자열을 인쇄합니다.

*fmt*를 채우기 위해, `fmt % part_info`를 실행하는데, 여기서 `part_info`는 다음 키와 값으로 구성된 딕셔너리입니다:

- `type` – `text`가 아닌 파트의 전체 MIME 유형
- `maintype` – `text`가 아닌 파트의 메인 MIME 유형
- `subtype` – `text`가 아닌 파트의 서브 MIME 유형
- `filename` – `text`가 아닌 파트의 파일명
- `description` – `text`가 아닌 파트와 관련된 설명
- `encoding` – `text`가 아닌 파트의 콘텐츠 전송 인코딩(Content transfer encoding)

*fmt*가 `None`이면, 다음 기본 *fmt*를 사용합니다:

"[Non-text (%(type)s) part of message omitted, filename %(filename)s]"

선택적 *_mangle_from_* 및 *maxheaderlen*은 `Generator` 베이스 클래스와 같습니다.

20.1.4 email.policy: 정책 객체

Added in version 3.3.

소스 코드: Lib/email/policy.py

`email` 패키지의 주요 초점은 다양한 전자 우편과 MIME RFC에 설명된 대로 전자 우편 메시지를 처리하는 것입니다. 그러나 전자 우편 메시지의 일반적인 형식(각각 이름, 콜론, 값 순으로 구성된 헤더 필드의 블록, 빈 줄과 임의의 '본문' 이 뒤따르는 전체 블록)은 전자 우편 영역 밖에서도 용도가 발견되는 형식입니다. 이러한 용도 중 일부는 메인 전자 우편 RFC와 상당히 유사하지만, 일부는 그렇지 않습니다. 전자 우편으로 작업할 때에도, RFC의 엄격한 준수를 위반하는 것이 바람직할 때가 있습니다. 가령 표준을 따르지 않는 전자 우편 서버와 상호 운영되거나 표준을 위반하는 방식으로 사용하기 원하는 확장을 구현하는 전자 우편을 생성하는 경우가 그렇습니다.

정책 객체는 email 패키지에 이러한 개별 사용 사례를 모두 처리 할 수 있는 유연성을 제공합니다.

`Policy` 객체는 사용 중에 email 패키지의 다양한 구성 요소 동작을 제어하는 일련의 어트리뷰트와 메서드를 캡슐화합니다. `Policy` 인스턴스는 email 패키지의 다양한 클래스와 메서드로 전달되어 기본 동작을 변경할 수 있습니다. 설정 가능한 값과 기본값은 아래에 설명되어 있습니다.

email 패키지의 모든 클래스에서 사용되는 기본 정책이 있습니다. 모든 `parser` 클래스와 관련 편의 함수 및 `Message` 클래스의 경우, 이것은 사전 정의된 인스턴스 `compat32`를 통한 `Compat32` 정책입니다. 이 정책은 파이썬 3.3 이전 버전의 email 패키지와 완전한 과거 호환성(어떤 경우에는, 버그 호환성을 포함합니다)을 제공합니다.

`EmailMessage`에 대한 *policy* 키워드의 기본값은 사전 정의된 인스턴스 `default`를 통한 `EmailPolicy` 정책입니다.

`Message`나 `EmailMessage` 객체가 만들어질 때, 정책을 획득합니다. 메시지가 `parser`로 만들어지면, 구문 분석기에 전달된 정책이 만들어지는 메시지가 사용하는 정책이 됩니다. 프로그램이 메시지를 만들면, 만들 때 정책을 지정할 수 있습니다. 메시지가 `generator`에 전달될 때, 제너레이터는 기본적으로 메시지의 정책을 사용하지만, 특정 정책을 제너레이터에 전달하여 메시지 객체에 저장된 정책을 재정의할 수도 있습니다.

`email.parser` 클래스와 구문 분석기 편의 함수에 대한 *policy* 키워드의 기본값은 이후 버전의 파이썬에서 **변경될 예정**입니다. 따라서 `parser` 모듈에서 설명된 클래스와 함수를 호출할 때는 **항상 사용할 정책을 명시적으로 지정**해야 합니다.

이 설명서의 첫 부분은 *compat32*를 포함한 모든 정책 객체에 공통적인 기능을 정의하는 추상 베이스 클래스 인 *Policy*의 기능을 다룹니다. 여기에는 email 패키지에 의해 내부적으로 호출되는 특정 훅 메서드가 포함되며, 사용자 정의 정책은 다른 동작을 얻기 위해 재정의할 수 있습니다. 두 번째 부분에서는 표준 동작과 이전 버전과 호환되는 동작과 기능을 각각 제공하는 훅을 구현하는 구상 클래스 *EmailPolicy*와 *Compat32*를 설명합니다.

Policy 인스턴스는 불변이지만, 복제할 수 있는데, 클래스 생성자와 같은 키워드 인자를 받아들이고 원본의 사본이지만 지정된 어트리뷰트 값이 변경된 새 *Policy* 인스턴스를 반환합니다.

예를 들어, 다음 코드를 사용하여 디스크의 파일에서 전자 우편 메시지를 읽고 유닉스 시스템의 시스템 `sendmail` 프로그램으로 전달할 수 있습니다:

```
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
...     msg = message_from_binary_file(f, policy=policy.default)
...
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone(linesep='\r\n'))
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

여기서 우리는 *BytesGenerator*에게 `sendmail`의 `stdin`으로 공급할 바이너리 문자열을 만들 때 RFC 올바른 줄 구분자 문자를 사용하도록 지시합니다. 기본 정책은 `\n` 줄 구분자를 사용합니다.

일부 email 패키지 메서드는 *policy* 키워드 인자를 받아들여, 해당 메서드에 대한 정책을 대체 할 수 있도록 합니다. 예를 들어, 다음 코드는 이전 예제의 *msg* 객체의 *as_bytes()* 메서드를 사용하고 실행 중인 플랫폼의 기본 줄 구분자를 사용하여 메시지를 파일에 씁니다:

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linesep)))
17
```

더하기 연산자를 사용하여 정책 객체를 결합하여, 기본값이 아닌 설정이 합쳐진 조합으로 설정된 정책 객체를 생성할 수도 있습니다:

```
>>> compat_SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

이 연산은 교환적(commutative)이지 않습니다; 즉, 객체가 더해지는 순서가 중요합니다. 예시하면 이렇습니다:

```
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

class email.policy.**Policy**(***kw*)

모든 정책 클래스의 추상 베이스 클래스입니다. 불변 속성, *clone()* 메서드 및 생성자 시맨틱의 구현뿐만 아니라 몇 가지 간단한 메서드에 대한 기본 구현을 제공합니다.

정책 클래스의 생성자에는 다양한 키워드 인자가 전달될 수 있습니다. 지정할 수 있는 인자는 이 클래스의 메서드 이외의 프로퍼티, 그리고 구상 클래스의 메서드 이외의 프로퍼티입니다. 생성자에 지정된 값은 해당 어트리뷰트의 기본값을 재정의합니다.

이 클래스는 다음 프로퍼티를 정의합니다. 따라서, 모든 정책 클래스의 생성자에 다음에 대한 값이 전달될 수 있습니다:

max_line_length

줄 종료 문자를 포함하지 않고, 직렬화된 출력에서 줄의 최대 길이. **RFC 5322**에 따라 기본값은 78입니다. `0`이나 *`None`* 값은 줄 바꿈을 전혀 수행하지 않아야 함을 나타냅니다.

linesep

직렬화된 출력에서 줄을 종료하는 데 사용되는 문자열입니다. RFC는 `\r\n`을 요구하지만, 기본값은 파이썬에서 사용하는 내부 줄 종료 규칙을 따라 `\n`입니다.

cte_type

사용해야 하는 콘텐츠 전송 인코딩(Content Transfer Encoding) 유형을 제어합니다. 가능한 값은 다음과 같습니다:

`7bit`	모든 데이터는 "7비트 클린"(ASCII 전용)이어야 합니다. 즉, 필요하면 quoted-printable이나 base64 인코딩을 사용하여 데이터를 인코딩합니다.
`8bit`	데이터는 7비트 클린으로 제한되지 않습니다. 헤더의 데이터는 여전히 ASCII 전용이어야 하므로 인코딩되지만 (예외는 아래 *`fold_binary()`*와 *`utf8`*을 참조하십시오), 본문 부분은 `8bit` CTE를 사용할 수 있습니다.

`8bit` `cte_type` 값은 문자열이 바이너리 데이터를 포함할 수 없어서 `Generator`가 아닌 `BytesGenerator`에서만 작동합니다. `Generator`가 `cte_type=8bit`를 지정하는 정책에 따라 작동하면, `cte_type`이 `7bit`인 것처럼 동작합니다.

raise_on_defect

*`True`*이면, 만나는 모든 결함을 에러로 발생시킵니다. *`False`*(기본값)이면, 결함은 *`register_defect()`* 메서드로 전달됩니다.

mangle_from_

*`True`*이면, 본문에서 "*From* "으로 시작하는 줄은 `>`를 앞에 배치하여 이스케이프 됩니다. 이 매개변수는 제너레이터가 메시지를 직렬화할 때 사용됩니다. 기본값: *`False`*.

Added in version 3.5.

message_factory

새로운 빈 메시지 객체를 생성하기 위한 팩토리 함수. 메시지를 구축할 때 구분 분석기가 사용합니다. 기본값은 `None`이며, 이때 *`Message`*가 사용됩니다.

Added in version 3.6.

verify_generated_headers

If `True` (the default), the generator will raise *`HeaderWriteError`* instead of writing a header that is improperly folded or delimited, such that it would be parsed as multiple headers or joined with adjacent data. Such headers can be generated by custom header classes or bugs in the `email` module.

As it' s a security feature, this defaults to `True` even in the *`Compat32`* policy. For backwards compatible, but unsafe, behavior, it must be set to `False` explicitly.

Added in version 3.13.

다음 *`Policy`* 메서드는 email 라이브러리를 사용하여 사용자 정의 설정으로 정책 인스턴스를 만드는 코드가 호출하기 위한 것입니다:

clone(***kw*)

키워드 인자로 새로운 값이 부여되는 어트리뷰트를 제외하고, 어트리뷰트가 현재 인스턴스와 같은 값을 갖는 새로운 *`Policy`* 인스턴스를 반환합니다.

나머지 *Policy* 메서드는 email 패키지 코드에 의해 호출되며, email 패키지를 사용하는 응용 프로그램에 의해 호출되는 용도가 아닙니다. 사용자 정의 정책은 이 모든 메서드를 구현해야 합니다.

handle_defect(*obj*, *defect*)

*obj*에서 찾은 *defect*를 처리합니다. email 패키지가 이 메서드를 호출할 때, *defect*는 항상 *MessageDefect*의 서브 클래스입니다.

기본 구현은 *raise_on_defect* 플래그를 확인합니다. `True`이면, *defect*가 예외로 발생합니다. `False`(기본값)이면 *obj*와 *defect*가 *register_defect()*로 전달됩니다.

register_defect(*obj*, *defect*)

*obj*에 *defect*를 등록합니다. email 패키지에서, *defect*는 항상 *MessageDefect*의 서브 클래스입니다.

기본 구현은 *obj*의 `defects` 어트리뷰트의 `append` 메서드를 호출합니다. email 패키지가 *handle_defect*를 호출할 때, *obj*는 일반적으로 `append` 메서드가 있는 `defects` 어트리뷰트가 있습니다. email 패키지와 함께 사용되는 사용자 정의 객체 형(예를 들어, 사용자 정의 `Message` 객체)도 이러한 어트리뷰트를 제공해야 합니다, 그렇지 않으면 구문 분석된 메시지의 결함이 예기치 않은 에러를 발생시킵니다.

header_max_count(*name*)

*name*이라는 헤더의 최대 허용 개수를 반환합니다.

헤더가 *EmailMessage*나 *Message* 객체에 추가될 때 호출됩니다. 반환 값이 `0`이나 `None`이 아니고, 반환 값보다 크거나 같은 수의 이름이 *name*인 헤더가 이미 있으면 *ValueError*가 발생합니다.

`Message.__setitem__`의 기본 동작은 값을 헤더 리스트에 추가하는 것이므로, 깨닫지 못하는 사이에 중복 헤더를 만들기 쉽습니다. 이 메서드는 특정 헤더를 `Message`에 프로그래밍 방식으로 추가할 수 있는 인스턴스의 수를 제한할 수 있도록 합니다. (이 제약은 구문 분석기가 보지 않습니다, 구문 분석기는 구문 분석 중인 메시지에 존재하는 수 만큼 헤더를 충실하게 생성합니다.)

기본 구현은 모든 헤더 이름에 대해 `None`을 반환합니다.

header_source_parse(*sourcelines*)

email 패키지는 문자열 리스트로 이 메서드를 호출하며, 각 문자열은 구문 분석 중인 소스에서 발견된 줄 구분 문자로 끝납니다. 첫 번째 줄에는 필드 헤더 이름과 구분자가 포함됩니다. 소스의 모든 공백이 유지됩니다. 이 메서드는 구문 분석된 헤더를 나타내기 위해 `Message`에 저장될 `(name, value)` 튜플을 반환해야 합니다.

구현이 기존 email 패키지 정책과의 호환성을 유지하기 원한다면, *name*은 대소 문자를 유지한 이름('`:`' 구분자까지의 모든 문자)이어야 하지만, *value*는 선행 공백이 제거되고 펼쳐진(unfolded) 값(모든 줄 구분자 문자는 제거하지만, 공백은 그대로 유지한)이어야 합니다.

*sourcelines*는 서로게이트 이스케이프 된 바이너리 데이터를 포함할 수 있습니다.

기본 구현이 없습니다

header_store_parse(*name*, *value*)

email 패키지는 (구문 분석기가 만든 `Message`가 아니라) 응용 프로그램이 `Message`를 프로그래밍 방식으로 수정할 때, 응용 프로그램이 제공한 name과 value로 이 메서드를 호출합니다. 이 메서드는 헤더를 나타내기 위해 `Message`에 저장될 `(name, value)` 튜플을 반환해야 합니다.

구현이 기존 email 패키지 정책과의 호환성을 유지하기 원한다면, *name*과 *value*는 전달된 인자의 내용을 변경하지 않는 문자열이나 문자열의 서브 클래스여야 합니다.

기본 구현이 없습니다

header_fetch_parse(*name*, *value*)

email 패키지는 응용 프로그램이 해당 헤더를 요청할 때 `Message`에 현재 저장된 *name*과 *value*로 이 메서드를 호출하며, 메서드가 반환하는 것은 꺼내는 헤더의 값으로 응용 프로그램에 다시 전달됩니다. `Message`에 같은 이름을 가진 헤더가 두 개 이상 있을 수 있음에 유의하십시오; 이 메서드로는 응용 프로그램으로 반환될 헤더의 특정 이름과 값이 전달됩니다.

*value*는 서로게이트 이스케이프 된 바이너리 데이터를 포함할 수 있습니다. 이 메서드가 반환하는 값에는 서로게이트 이스케이프 된 바이너리 데이터가 없어야 합니다.

기본 구현이 없습니다

fold(*name*, *value*)

email 패키지는 지정된 헤더에 대해 `Message`에 현재 저장된 *name*과 *value*로 이 메서드를 호출합니다. 이 메서드는 *name*을 *value*와 합치고 적절한 위치에 `linesep` 문자를 삽입하여 (정책 설정에 따라) 올바르게 "접힌(folded)" 헤더를 나타내는 문자열을 반환해야 합니다. 전자 우편 헤더 접기 규칙에 대한 설명은 **RFC 5322**를 참조하십시오.

*value*는 서로게이트 이스케이프 된 바이너리 데이터를 포함할 수 있습니다. 메서드가 반환한 문자열에는 서로게이트 이스케이프 된 바이너리 데이터가 없어야 합니다.

fold_binary(*name*, *value*)

반환 값이 문자열이 아니라 바이트열 객체여야 한다는 점을 제외하고는 `fold()`와 같습니다.

*value*는 서로게이트 이스케이프 된 바이너리 데이터를 포함할 수 있습니다. 이들은 반환된 바이트열 객체에서 바이너리 데이터로 다시 변환될 수 있습니다.

class email.policy.**EmailPolicy**(**kw)

이 구상 `Policy`는 현재 전자 우편 RFC를 완전히 준수하기 위한 동작을 제공합니다. 여기에는 **RFC 5322**, **RFC 2047** 및 현재 MIME RFC가 포함되지만 이에 국한되지는 않습니다.

이 정책은 새로운 헤더 구문 분석과 접기(folding) 알고리즘을 추가합니다. 단순한 문자열 대신, 헤더는 필드 유형에 따라 달라지는 어트리뷰트를 가진 `str` 서브 클래스입니다. 구문 분석과 접기 알고리즘은 **RFC 2047**과 **RFC 5322**를 완전히 구현합니다.

`message_factory` 어트리뷰트의 기본값은 `EmailMessage`입니다.

모든 정책에 적용되는 위에 나열된 설정 가능 어트리뷰트 외에도, 이 정책은 다음과 같은 어트리뷰트를 추가합니다:

Added in version 3.6:[1]

utf8

`False`이면, **RFC 5322**를 따르고 헤더에서 ASCII가 아닌 문자를 "인코딩된 단어"로 인코딩하여 지원합니다. `True`이면, **RFC 6532**를 따르고 헤더에 `utf-8` 인코딩을 사용합니다. 이러한 방식으로 포맷된 메시지는 `SMTPUTF8` 확장(**RFC 6531**)을 지원하는 SMTP 서버로 전달될 수 있습니다.

refold_source

`Message` 객체의 헤더 값이 (프로그램이 설정하는 것과 대조적으로) `parser`에서 온 것이면, 이 어트리뷰트는 메시지를 직렬화된 형식으로 다시 변환할 때 제너레이터가 그 값을 다시 접어야 하는지를 나타냅니다. 가능한 값은 다음과 같습니다:

`none`	모든 소스 값은 원래 접기를 사용합니다
`long`	`max_line_length`보다 긴 줄이 있는 소스 값은 다시 접힙니다.
`all`	모든 값이 다시 접힙니다.

기본값은 `long`입니다.

header_factory

`name`과 `value` 두 개의 인자를 취하는 콜러블. 여기서 `name`은 헤더 필드 이름이고 `value`는 펼쳐진 헤더 필드 값이며 해당 헤더를 나타내는 문자열 서브 클래스를 반환합니다. 다양한 주소와 날짜 **RFC 5322** 헤더 필드 유형과 주요 MIME 헤더 필드 유형에 대한 사용자 정의 구문 분석을 지원하는 기본 `header_factory`(`headerregistry`를 참조하십시오)가 제공됩니다. 향후 추가 사용자 정의 구문 분석에 대한 지원이 추가될 것입니다.

[1] 원래 3.3에 잠정적 기능으로 추가되었습니다.

content_manager

적어도 두 개의 메서드가 있는 객체: get_content와 set_content. *EmailMessage* 객체의 *get_content()*나 *set_content()* 메서드가 호출될 때, 이 객체의 해당 메서드를 호출하는데, 메시지 객체를 첫 번째 인자로 전달하고 전달된 다른 인자와 키워드를 추가 인자로 전달합니다. 기본적으로 content_manager는 *raw_data_manager*로 설정됩니다.

Added in version 3.4.

이 클래스는 다음과 같은 *Policy*의 추상 메서드의 구상 구현을 제공합니다:

header_max_count(*name*)

지정된 이름의 헤더를 나타내는 데 사용되는 특수화된 클래스의 *max_count* 어트리뷰트 값을 반환합니다.

header_source_parse(*sourcelines*)

이름은 ':'까지의 모든 것으로 구문 분석되고 수정되지 않은 상태로 반환됩니다. 값은 첫 번째 줄의 나머지 부분에서 선행 공백을 제거한 후에 모든 후속 줄을 이어붙이고 후행 캐리지 리턴이나 줄 바꿈 문자를 제거하여 결정됩니다.

header_store_parse(*name*, *value*)

이름은 변경되지 않고 반환됩니다. 입력값에 name 어트리뷰트가 있고 대소 문자를 무시하고 *name*과 일치하면, 값은 변경되지 않고 반환됩니다. 그렇지 않으면 *name*과 *value*는 header_factory로 전달되고, 결과 헤더 객체가 값으로 반환됩니다. 이 경우 입력값에 CR이나 LF 문자가 포함되어 있으면 ValueError가 발생합니다.

header_fetch_parse(*name*, *value*)

값에 name 어트리뷰트가 있으면, 수정되지 않은 상태로 반환됩니다. 그렇지 않으면 *name*과 CR이나 LF 문자가 제거된 *value*가 header_factory로 전달되고, 결과 헤더 객체가 반환됩니다. 서로게이트 이스케이프 된 바이트열은 유니코드 알 수 없는 문자 글리프(unknown-character glyph)로 바뀝니다.

fold(*name*, *value*)

헤더 접기는 *refold_source* 정책 설정에 의해 제어됩니다. 값은 name 어트리뷰트가 없을 때, 그리고 그때만 '소스값' 으로 간주합니다 (name 어트리뷰트가 있다는 것은 헤더 객체나 그 일종이라는 뜻입니다). 정책에 따라 소스값을 다시 접어야 할 필요가 있으면, header_factory에 *name*과 CR과 LF 문자가 제거된 *value*를 전달하여 헤더 객체로 변환됩니다. 헤더 객체의 접기는 현재 정책으로 fold 메서드를 호출하여 수행됩니다.

소스값은 *splitlines()*를 사용하여 줄로 분할됩니다. 값을 다시 접지 않으면, 정책의 linesep을 사용하여 줄을 다시 이어붙인 후에 반환합니다. ASCII가 아닌 바이너리 데이터가 포함된 줄은 예외입니다. 이 경우 refold_source 설정과 관계없이 값이 다시 접히는데, unknown-8bit 문자 집합을 사용하여 바이너리 데이터가 CTE로 인코딩됩니다.

fold_binary(*name*, *value*)

*cte_type*이 7bit이면, 반환된 값이 바이트열인 것을 제외하고 *fold()*와 같습니다.

*cte_type*이 8bit이면, ASCII가 아닌 바이너리 데이터는 다시 바이트열로 변환됩니다. 바이너리 데이터가 단일 바이트 문자와 멀티 바이트 문자 중 어는 것으로 구성되어 있는지 알 방법이 없어서, refold_header 설정과 관계없이 바이너리 데이터가 있는 헤더는 다시 접히지 않습니다.

다음 *EmailPolicy* 인스턴스는 특정 응용 프로그램 도메인에 적합한 기본값을 제공합니다. 향후 이러한 인스턴스들(특히 HTTP 인스턴스)의 동작은 그 들의 도메인과 관련된 RFC에 훨씬 더 가깝게 조정될 수 있음에 유의하십시오.

email.policy.**default**

모든 기본값이 변경되지 않은 EmailPolicy 인스턴스. 이 정책은 RFC 올바른 \r\n이 아닌 표준 파이썬 \n 줄 종료를 사용합니다.

email.policy.**SMTP**

전자 우편 RFC를 준수하도록 메시지를 직렬화하는 데 적합합니다. default와 유사하지만, linesep이 \r\n으로 설정되어 RFC를 준수합니다.

email.policy.**SMTPUTF8**

*utf8*가 True라는 점을 제외하고, SMTP와 같습니다. 헤더에 인코딩된 단어를 사용하지 않고 메시지를 메시지 저장소로 직렬화하는 데 유용합니다. SMTP 전송에는 발신자나 수신자 주소에 ASCII가 아닌 문자가 있을 때만 사용해야 합니다 (*smtplib.SMTP.send_message()* 메서드는 이를 자동으로 처리합니다).

email.policy.**HTTP**

HTTP 트래픽에 사용하기 위해 헤더를 직렬화하는 데 적합합니다. max_line_length가 None(무제한)으로 설정된 것을 제외하고, SMTP와 유사합니다.

email.policy.**strict**

편의 인스턴스. raise_on_defect가 True로 설정된 것을 제외하고, default와 같습니다. 다음과 같이 작성하여 모든 정책을 엄격하게 만들 수 있도록 합니다:

```
somepolicy + policy.strict
```

이러한 모든 *EmailPolicy*를 통해, email 패키지의 효과적인 API가 다음과 같은 방식으로 파이썬 3.2 API에서 변경됩니다:

- *Message*에서 헤더를 설정하면 해당 헤더가 구문 분석되고 헤더 객체가 만들어집니다.
- *Message*에서 헤더 값을 가져오면 해당 헤더가 구문 분석되고 헤더 객체가 만들어져 반환됩니다.
- 모든 헤더 객체나 정책 설정으로 인해 다시 접힌 모든 헤더는 인코딩된 단어가 필요한 위치와 허용되는 위치를 포함하여 RFC 접기 알고리즘을 완전히 구현하는 알고리즘을 사용하여 접힙니다.

응용 프로그램의 시각에서, 이것은 *EmailMessage*를 통해 얻은 모든 헤더가 추가 어트리뷰트가 있는 헤더 객체이며, 그것의 문자열 값은 헤더의 완전히 디코딩된 유니코드 값이 됨을 뜻합니다. 마찬가지로, 유니코드 문자열을 사용하여 헤더에 새 값이나 새로 만들어진 헤더를 대입할 수 있으며, 정책은 유니코드 문자열을 올바른 RFC 인코딩 형식으로 변환합니다.

헤더 객체와 그들의 어트리뷰트는 *headerregistry*에 설명되어 있습니다.

class email.policy.**Compat32**(***kw*)

이 구상 *Policy*는 과거 호환성 정책입니다. 파이썬 3.2에 있는 email 패키지의 동작을 흉내 냅니다. *policy* 모듈은 이 클래스의 인스턴스 *compat32*도 정의하고, 기본 정책으로 사용합니다. 따라서 email 패키지의 기본 동작은 파이썬 3.2와의 호환성을 유지하는 것입니다.

다음 어트리뷰트는 *Policy* 기본값과 다른 값을 갖습니다:

mangle_from_

기본값은 True입니다.

이 클래스는 다음과 같은 *Policy*의 추상 메서드의 구상 구현을 제공합니다:

header_source_parse(*sourcelines*)

이름은 ':'까지의 모든 것으로 구문 분석되고 수정되지 않은 상태로 반환됩니다. 값은 첫 번째 줄의 나머지 부분에서 선행 공백을 제거한 후에 모든 후속 줄을 이어붙이고 후행 캐리지 리턴이나 줄 바꿈 문자를 제거하여 결정됩니다.

header_store_parse(*name*, *value*)

이름과 값은 수정되지 않은 상태로 반환됩니다.

header_fetch_parse(*name*, *value*)

값에 바이너리 데이터가 포함되어 있으면, unknown-8bit 문자 집합을 사용하여 *Header* 객체로 변환됩니다. 그렇지 않으면 수정되지 않은 상태로 반환됩니다.

fold(*name*, *value*)

Header 접기 알고리즘을 사용하여 헤더를 접습니다. 이 알고리즘은 값의 기존 줄 바꿈을 유지하고, 각 결과 줄을 max_line_length로 줄 넘김 합니다. ASCII가 아닌 바이너리 데이터는 unknown-8bit 문자 집합을 사용하여 CTE 인코딩됩니다.

fold_binary(*name*, *value*)

Header 접기 알고리즘을 사용하여 헤더를 접습니다. 이 알고리즘은 값의 기존 줄 바꿈을 유지하고, 각 결과 줄을 max_line_length로 줄 넘김 합니다. cte_type이 7bit이면, ASCII가 아닌 바이너리 데이터는 unknown-8bit 문자 집합을 사용하여 CTE 인코딩됩니다. 그렇지 않으면 원본 소스 헤더가 사용되는데, 기존 줄 바꿈과 임의의 (RFC 유효하지 않은) 바이너리 데이터가 포함될 수 있습니다.

email.policy.**compat32**

파이썬 3.2 email 패키지 동작과의 호환성을 제공하는 *Compat32*의 인스턴스.

20.1.5 email.errors: 예외와 결함 클래스

소스 코드: Lib/email/errors.py

---

email.errors 모듈에는 다음과 같은 예외 클래스가 정의되어 있습니다:

exception email.errors.**MessageError**

이것은 *email* 패키지가 발생시킬 수 있는 모든 예외의 베이스 클래스입니다. 표준 *Exception* 클래스에서 파생되며 추가 메서드를 정의하지 않습니다.

exception email.errors.**MessageParseError**

이것은 *Parser* 클래스에서 발생하는 예외의 베이스 클래스입니다. *MessageError*에서 파생됩니다. 이 클래스는 *headerregistry*에서 사용하는 구문 분석기에서도 내부적으로 사용됩니다.

exception email.errors.**HeaderParseError**

메시지의 **RFC 5322** 헤더를 구문 분석할 때 일부 에러 조건에서 발생합니다. 이 클래스는 *MessageParseError*에서 파생됩니다. 메서드가 호출될 때 콘텐츠 유형을 알 수 없으면, *set_boundary()* 메서드는 이 에러를 발생시킵니다. *Header*는 특정 base64 디코딩 에러와 내장된 헤더를 포함하는 것으로 보이는 헤더를 만들려고 할 때 (즉, 연장 줄(continuation line) 이어야 할 곳에 선행 공백이 없고 헤더처럼 보이는 것이 있을 때) 이 에러를 발생시킬 수 있습니다.

exception email.errors.**BoundaryError**

폐지되었고 더는 사용되지 않습니다.

exception email.errors.**MultipartConversionError**

*MIMENonMultipart*에서 파생된 클래스(가령 *MIMEImage*)의 인스턴스에서 *attach()* 메서드를 호출할 때 발생합니다. *MultipartConversionError*는 *MessageError*와 내장 *TypeError*에서 다중 상속됩니다.

exception email.errors.**HeaderWriteError**

Raised when an error occurs when the *generator* outputs headers.

exception email.errors.**MessageDefect**

이메일 메시지를 구문 분석할 때 발견되는 모든 결함에 대한 베이스 클래스입니다. *ValueError*에서 파생됩니다.

exception email.errors.**HeaderDefect**

이메일 헤더를 구문 분석할 때 발견되는 모든 결함에 대한 베이스 클래스입니다. *MessageDefect*에서 파생됩니다.

다음은 메시지를 구문 분석하는 동안 *FeedParser*가 찾을 수 있는 결함 목록입니다. 문제가 발견된 메시지에 결함이 추가됨에 유의하십시오. 그래서, 예를 들어, *multipart/alternative* 내에 중첩된 메시지에 잘못된 헤더가 있으면, 해당 중첩 메시지 객체가 결함을 갖게 되지만 포함하는 메시지는 그렇지 않습니다.

모든 결함 클래스는 *email.errors.MessageDefect*의 서브 클래스입니다.

exception email.errors.**NoBoundaryInMultipartDefect**

메시지가 멀티 파트라고 주장했지만, *boundary* 파라미터가 없습니다.

exception email.errors.**StartBoundaryNotFoundDefect**

Content-Type 헤더에서 주장하는 시작 경계를 찾지 못했습니다.

exception email.errors.**CloseBoundaryNotFoundDefect**

시작 경계가 발견되었지만, 해당하는 종료 경계가 발견되지 않았습니다.

Added in version 3.3.

exception email.errors.**FirstHeaderLineIsContinuationDefect**

메시지의 첫 번째 헤더 줄에 연장 줄(continuation line)이 있습니다.

exception email.errors.**MisplacedEnvelopeHeaderDefect**

헤더 블록 중간에 "Unix From" 헤더가 있습니다.

exception email.errors.**MissingHeaderBodySeparatorDefect**

헤더를 구문 분석하는 중에 선행 공백이 없지만 ':' 가 포함되지 않은 줄이 발견되었습니다. 그 줄이 본문의 첫 번째 줄을 나타내는 것으로 가정하여 구문 분석이 계속됩니다.

Added in version 3.3.

exception email.errors.**MalformedHeaderDefect**

콜론이 없거나 다른 식으로 잘못된 헤더가 발견되었습니다.

버전 3.3부터 폐지됨: 이 결함은 여러 파이썬 버전에서 사용되지 않았습니다.

exception email.errors.**MultipartInvariantViolationDefect**

메시지가 *multipart*라고 주장했지만, 서브 파트가 없습니다. 메시지에 이 결함이 있으면, 콘텐츠 유형이 *multipart*라고 주장하더라도 *is_multipart()* 메서드는 False를 반환할 수 있음에 유의하십시오.

exception email.errors.**InvalidBase64PaddingDefect**

base64로 인코딩된 바이트열 블록을 디코딩할 때, 패딩이 올바르지 않습니다. 디코딩을 수행하기 위해 충분한 패딩이 추가되지만, 바이트열을 디코딩한 결과는 유효하지 않을 수 있습니다.

exception email.errors.**InvalidBase64CharactersDefect**

base64로 인코딩된 바이트열 블록을 디코딩할 때, base64 알파벳 이외의 문자가 발견되었습니다. 문자는 무시되지만, 바이트열을 디코딩한 결과는 유효하지 않을 수 있습니다.

exception email.errors.**InvalidBase64LengthDefect**

base64로 인코딩된 바이트열 블록을 디코딩할 때, 비 패딩 base64 문자 수가 유효하지 않습니다 (4의 배수보다 1이 큽니다). 인코딩된 블록은 그대로 유지됩니다.

exception email.errors.**InvalidDateDefect**

When decoding an invalid or unparsable date field. The original value is kept as-is.

20.1.6 email.headerregistry: 사용자 정의 헤더 객체

소스 코드: Lib/email/headerregistry.py

Added in version 3.6:[1]

헤더는 *str*의 사용자 정의된 서브 클래스로 표현됩니다. 주어진 헤더를 표현하는 데 사용되는 특정 클래스는 헤더가 만들어질 때 *policy*의 *header_factory*에 의해 결정됩니다. 이 섹션에서는 **RFC 5322** 호환 전자우편 메시지를 처리하기 위해 email 패키지가 구현한 특정 header_factory에 관해 설명합니다. 다양한 헤더 유형에 대해 사용자 정의된 헤더 객체를 제공할 뿐만 아니라, 응용 프로그램에서 고유한 사용자 정의 헤더 유형을 추가할 수 있는 확장 메커니즘을 제공합니다.

*EmailPolicy*에서 파생된 정책 객체를 사용할 때, 모든 헤더는 *HeaderRegistry*가 생성하며 마지막 베이스 클래스로 *BaseHeader*를 갖습니다. 각 헤더 클래스에는 헤더 유형에 따라 결정되는 추가 베이스 클래스가 있습니다. 예를 들어, 많은 헤더에는 다른 베이스 클래스로 *UnstructuredHeader* 클래스를 갖습니다.

[1] 원래 3.3에서 잠정적 모듈로 추가되었습니다.

헤더의 특수화된 두 번째 클래스는 *HeaderRegistry*에 저장된 검색 테이블을 사용하여 헤더의 이름으로 결정됩니다. 이 모든 것은 일반적인 응용 프로그램에 대해 투명하게 관리되지만, 더욱 복잡한 응용 프로그램에서 사용할 수 있도록 기본 동작을 수정하기 위한 인터페이스가 제공됩니다.

아래 섹션은 먼저 헤더 베이스 클래스와 그들의 어트리뷰트를, 그다음으로 *HeaderRegistry*의 동작을 수정하기 위한 API를, 그리고 마지막으로 구조화된 헤더에서 구문 분석된 데이터를 나타내는 데 사용되는 지원 클래스에를 설명합니다.

class email.headerregistry.**BaseHeader**(*name*, *value*)

*name*과 *value*는 *header_factory* 호출에서 BaseHeader로 전달됩니다. 헤더 객체의 문자열 값은 유니코드로 완전히 디코딩된 *value*입니다.

이 베이스 클래스는 다음과 같은 읽기 전용 프로퍼티를 정의합니다:

name

헤더 이름('':' 앞의 필드 부분). 이것은 정확히 *header_factory* 호출에 전달된 *name*에 대한 값입니다; 즉, 대소 문자가 유지됩니다.

defects

구문 분석 중 발견된 RFC 준수 문제를 보고하는 *HeaderDefect* 인스턴스 튜플. email 패키지는 규정 준수 문제 감지에 대해 완전해지려고 합니다. 보고될 수 있는 결함 유형에 대한 설명은 *errors* 모듈을 참조하십시오.

max_count

같은 name을 가질 수 있는 이 유형의 최대 헤더 수. None 값은 무제한을 의미합니다. 이 어트리뷰트의 BaseHeader 값은 None입니다; 특수화된 헤더 클래스가 필요할 때 이 값을 재정의할 것으로 기대됩니다.

BaseHeader는 또한 email 라이브러리 코드에 의해 호출되고 일반적으로 응용 프로그램이 호출해서는 안 되는 다음 메서드를 제공합니다:

fold(*, *policy*)

*policy*에 따라 헤더를 올바르게 접는 데 필요한 *linesep* 문자를 포함하는 문자열을 반환합니다. 헤더는 임의의 바이너리 데이터를 포함할 수 없어서, 8bit의 *cte_type*은 마치 7bit인 것처럼 처리됩니다. *utf8*이 False이면, 비 ASCII 데이터는 **RFC 2047**로 인코딩됩니다.

BaseHeader 자체는 헤더 객체를 만드는 데 사용할 수 없습니다. 헤더 객체를 생성하기 위해 각 특수화된 헤더가 협력하는 프로토콜을 정의합니다. 특히 BaseHeader는 특수화된 클래스가 parse라는 *classmethod()*를 제공할 것을 요구합니다. 이 메서드는 다음과 같이 호출됩니다:

```
parse(string, kwds)
```

kwds는 하나의 미리 초기화된 키 defects를 포함하는 딕셔너리입니다. defects는 빈 리스트입니다. parse 메서드는 감지된 결함을 이 리스트에 추가해야 합니다. 반환될 때, kwds 딕셔너리는 반드시 적어도 키 decoded와 defects에 대한 값을 포함해야 합니다. decoded는 헤더의 문자열 값이어야 합니다 (즉, 유니코드로 완전히 디코딩된 헤더 값). parse 메서드는 *string*이 콘텐츠 전송 인코딩된 파트를 포함할 수 있다고 가정해야 하지만, 인코딩되지 않은 헤더 값을 구문 분석할 수 있도록 모든 유효한 유니코드 문자도 올바르게 처리해야 합니다.

BaseHeader의 __new__는 헤더 인스턴스를 만들고, init 메서드를 호출합니다. 특수화된 클래스가 BaseHeader 자체에서 제공하는 것 이상의 추가 어트리뷰트를 설정하려면 init 메서드 만 제공하면 됩니다. 이러한 init 메서드는 다음과 같아야 합니다:

```
def init(self, /, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

즉, 특수화된 클래스가 kwds 딕셔너리에 추가하는 것은 제거해야 하고 처리해야 하며, kw(및 args)의 나머지 내용은 BaseHeader init 메서드로 전달됩니다.

class email.headerregistry.**UnstructuredHeader**

"구조화되지 않은" 헤더는 **RFC 5322**의 기본 헤더 유형입니다. 지정된 문법이 없는 헤더는 구조화되지 않은 것으로 취급됩니다. 구조화되지 않은 헤더의 전형적인 예는 *Subject* 헤더입니다.

RFC 5322에서, 구조화되지 않은 헤더는 ASCII 문자 집합에서 임의의 텍스트를 나열합니다. 그러나 RFC 2047은 비 ASCII 텍스트를 헤더 값 내에서 ASCII 문자로 인코딩하기 위한 RFC 5322 호환 메커니즘을 가지고 있습니다. 인코딩된 단어를 포함하는 *value*가 생성자에 전달되면, `UnstructuredHeader` 구문 분석기는 구조화되지 않은 텍스트의 RFC 2047 규칙에 따라 인코딩된 단어를 유니코드로 변환합니다. 구문 분석기는 휴리스틱을 사용하여 특정 비 호환 인코딩된 단어를 디코딩하려고 시도합니다. 인코딩된 단어나 인코딩되지 않는 텍스트 내의 유효하지 않은 문자와 같은 문제에 대한 결함뿐만 아니라 이럴 때 결함을 등록합니다.

이 헤더 유형은 추가 어트리뷰트를 제공하지 않습니다.

class email.headerregistry.**DateHeader**

RFC 5322는 전자 우편 헤더 내의 날짜에 대해 매우 구체적인 형식을 지정합니다. `DateHeader` 구문 분석기는 이 날짜 형식을 인식할 뿐만 아니라, "야생"에서 발견되는 다양한 변종 형식을 인식합니다.

이 헤더 유형은 다음과 같은 추가 어트리뷰트를 제공합니다:

datetime

헤더 값이 한 양식이나 다른 양식의 유효한 날짜로 인식될 수 있으면, 이 어트리뷰트는 해당 날짜를 나타내는 *datetime* 인스턴스가 포함됩니다. 입력 날짜의 시간대가 `-0000`으로 지정되면 (UTC이지만 소스 시간대에 대한 정보는 포함하지 않음을 나타냅니다), *datetime*은 나이브 *datetime*이 됩니다. 특정 시간대 오프셋이 발견되면 (`+0000`을 포함합니다), *datetime*에는 *datetime.timezone*을 사용하여 시간대 오프셋을 기록하는 어웨어 `datetime`이 포함됩니다.

헤더의 `decoded` 값은 RFC 5322 규칙에 따라 `datetime`을 포매팅해서 결정됩니다; 즉, 다음과 같이 설정됩니다:

```
email.utils.format_datetime(self.datetime)
```

`DateHeader`를 만들 때, *value*는 *datetime* 인스턴스일 수 있습니다. 이것은, 예를 들어, 다음 코드가 유효하고 기대하는 것을 수행함을 뜻합니다:

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

이것은 나이브 `datetime`이므로 UTC 타임스탬프로 해석되며 결괏값의 시간대는 `-0000`입니다. *utils* 모듈의 *localtime()* 함수를 사용하는 것이 훨씬 더 유용합니다:

```
msg['Date'] = utils.localtime()
```

이 예에서는 현재 시간대 오프셋을 사용하여 날짜 헤더를 현재 시간과 날짜로 설정합니다.

class email.headerregistry.**AddressHeader**

주소 헤더는 가장 복잡한 구조화된 헤더 유형 중 하나입니다. `AddressHeader` 클래스는 모든 주소 헤더에 대한 범용 인터페이스를 제공합니다.

이 헤더 유형은 다음과 같은 추가 어트리뷰트를 제공합니다:

groups

헤더 값에서 찾은 주소와 그룹을 인코딩하는 *Group* 객체의 튜플. 그룹 일부가 아닌 주소는 이 목록에서 *display_name*이 `None`인 단일 주소 `Groups`로 표현됩니다.

addresses

헤더 값의 모든 개별 주소를 인코딩하는 *Address* 객체의 튜플. 헤더 값이 그룹을 포함하면, 그룹의 개별 주소가 값에서 그룹이 발생하는 지점에서 목록에 포함됩니다 (즉, 주소 목록은 1 차원 목록으로 "평평하게" 만들어집니다).

헤더의 `decoded` 값에서는 모든 인코딩된 단어가 유니코드로 디코딩됩니다. *idna*로 인코딩된 도메인 이름도 유니코드로 디코딩됩니다. `decoded` 값은 `groups` 어트리뷰트 요소의 *str* 값을 `', '`로 이어 붙여 설정됩니다.

*Address*와 *Group* 객체의 목록을 임의 조합한 목록을 주소 헤더의 값을 설정하는 데 사용할 수 있습니다. `display_name`이 `None`인 `Group` 객체는 단일 주소로 해석되므로, 소스 헤더의 `groups` 어트리뷰트에서 얻은 목록을 사용하여 주소 목록을 그룹과 함께 그대로 복사할 수 있습니다.

class email.headerregistry.**SingleAddressHeader**

하나의 추가 어트리뷰트를 추가하는 *AddressHeader*의 서브 클래스:

address

헤더 값으로 인코딩된 단일 주소. 헤더 값에 실제로 둘 이상의 주소가 포함될 때 (기본 *policy*에서 RFC 위반입니다), 이 어트리뷰트에 액세스하면 *ValueError*가 발생합니다.

위의 많은 클래스에는 Unique 변형도 있습니다 (예를 들어, UniqueUnstructuredHeader). 유일한 차이점은 Unique 변형에서 *max_count*가 1로 설정되어 있다는 것입니다.

class email.headerregistry.**MIMEVersionHeader**

MIME-Version 헤더에는 실제로 하나의 유효한 값만 있으며, 이는 1.0입니다. 미래에 안전하기 위해, 이 헤더 클래스는 다른 유효한 버전 번호를 지원합니다. 버전 번호가 **RFC 2045**에 따라 유효한 값을 가지면, 헤더 객체는 다음 어트리뷰트에 None이 아닌 값을 갖습니다:

version

공백 및/또는 주석이 제거된 문자열 버전 번호.

major

정수 주 버전 번호

minor

정수 부 버전 번호

class email.headerregistry.**ParameterizedMIMEHeader**

MIME 헤더는 모두 접두사 'Content-'로 시작합니다. 각 특정 헤더에는 해당 헤더의 클래스에 설명된 특정 값이 있습니다. 일부는 공통 형식을 가진 보조 파라미터 목록을 취할 수도 있습니다. 이 클래스는 파라미터를 취하는 모든 MIME 헤더의 베이스로 사용됩니다.

params

파라미터 이름을 파라미터값으로 매핑하는 딕셔너리.

class email.headerregistry.**ContentTypeHeader**

Content-Type 헤더를 처리하는 *ParameterizedMIMEHeader* 클래스.

content_type

maintype/subtype 형식의 콘텐츠 유형 문자열.

maintype

subtype

class email.headerregistry.**ContentDispositionHeader**

Content-Disposition 헤더를 처리하는 *ParameterizedMIMEHeader* 클래스.

content_disposition

inline과 attachment가 일반적으로 사용되는 유일하게 유효한 값입니다.

class email.headerregistry.**ContentTransferEncoding**

Content-Transfer-Encoding 헤더를 처리합니다.

cte

유효한 값은 7bit, 8bit, base64 및 quoted-printable입니다. 자세한 정보는 **RFC 2045**를 참조하십시오.

class email.headerregistry.**HeaderRegistry**(*base_class=BaseHeader*, *default_class=UnstructuredHeader*, *use_default_map=True*)

기본적으로 *EmailPolicy*에서 사용되는 팩토리입니다. HeaderRegistry는 *base_class*와 보유한 등록소에서 꺼낸 특수화된 클래스를 사용하여 헤더 인스턴스를 동적으로 만드는 데 사용되는 클래스를 구축합니다. 지정된 헤더 이름이 등록소에 나타나지 않으면, *default_class*에 의해 지정된 클래스가 특수화된 클래스로 사용됩니다. *use_default_map*이 True(기본값)이면, 헤더 이름에서 클래스로의 표준

매핑이 초기화 중에 등록소에 복사됩니다. *base_class*는 항상 생성된 클래스의 `__bases__` 목록에서 마지막 클래스입니다.

기본 매핑은 다음과 같습니다:

subject
UniqueUnstructuredHeader

date
UniqueDateHeader

resent-date
DateHeader

orig-date
UniqueDateHeader

sender
UniqueSingleAddressHeader

resent-sender
SingleAddressHeader

to
UniqueAddressHeader

resent-to
AddressHeader

cc
UniqueAddressHeader

resent-cc
AddressHeader

bcc
UniqueAddressHeader

resent-bcc
AddressHeader

from
UniqueAddressHeader

resent-from
AddressHeader

reply-to
UniqueAddressHeader

mime-version
MIMEVersionHeader

content-type
ContentTypeHeader

content-disposition
ContentDispositionHeader

content-transfer-encoding
ContentTransferEncodingHeader

message-id
MessageIDHeader

`HeaderRegistry`에는 다음과 같은 메서드가 있습니다:

map_to_type(*self*, *name*, *cls*)

*name*은 매핑할 헤더의 이름입니다. 등록소에서 소문자로 변환됩니다. *cls*는 *name*과 일치하는 헤더를 인스턴스 화하는 데 사용되는 클래스를 만들기 위해 *base_class*와 함께 사용되는 특수 클래스입니다.

__getitem__(*name*)

name 헤더 생성을 처리할 클래스를 구성하고 반환합니다.

__call__(*name*, *value*)

등록소에서 *name*과 연관된 특수화된 헤더를 꺼내고 (등록소에 *name*이 없으면 *default_class*를 사용해서), *base_class*와 결합하여 클래스를 생성하고, 생성된 클래스의 생성자를 같은 인자 목록을 전달해서 호출한 후, 마지막으로 이렇게 만들어진 클래스 인스턴스를 반환합니다.

다음 클래스는 구조화된 헤더에서 구문 분석된 데이터를 나타내는 데 사용되는 클래스이며 일반적으로 응용 프로그램에서 특정 헤더에 지정할 구조화된 값을 대입하는 데 사용할 수 있습니다.

class email.headerregistry.**Address**(*display_name=''*, *username=''*, *domain=''*, *addr_spec=None*)

전자 우편 주소를 나타내는 데 사용되는 클래스. 주소의 일반적인 형식은 다음과 같습니다:

```
[display_name] <username@domain>
```

또는:

```
username@domain
```

여기서 각 부분은 **RFC 5322**에 명시된 특정 문법 규칙을 준수해야 합니다.

편의상 *username*과 *domain* 대신 *addr_spec*을 지정할 수 있으며, 이 경우 *addr_spec*에서 *username*과 *domain*이 구문 분석됩니다. *addr_spec*은 올바르게 RFC 인용된 문자열(quoted string)이어야 합니다; 그렇지 않으면 `Address`는 에러를 발생시킵니다. 유니코드 문자는 허용되며 직렬화될 때 적절하게 인코딩됩니다. 그러나, RFC에 따라, 주소의 사용자 이름 부분에는 유니코드가 허용되지 않습니다.

display_name

모든 인용(quoting)이 제거된 주소의 표시 이름 부분 (있다면). 주소에 표시 이름이 없으면, 이 어트리뷰트는 빈 문자열입니다.

username

모든 인용(quoting)이 제거된 주소의 `username` 부분.

domain

주소의 `domain` 부분.

addr_spec

주소의 `username@domain` 부분, 단순 주소(위의 두 번째 형식)로 사용하기 위해 올바르게 인용(quoted)됩니다. 이 어트리뷰트는 불변입니다.

__str__()

객체의 `str` 값은 **RFC 5322** 규칙에 따라 인용된(quoted) 주소이지만, ASCII가 아닌 문자의 콘텐츠 전송 인코딩(Content Transfer Encoding)은 없습니다.

SMTP(**RFC 5321**)를 지원하기 위해, `Address`는 한가지 특별한 경우를 처리합니다: `username`과 `domain`이 모두 빈 문자열(또는 `None`)이면, `Address`의 문자열 값은 `<>`입니다.

class email.headerregistry.**Group**(*display_name=None*, *addresses=None*)

주소 그룹을 표현하는 데 사용되는 클래스. 주소 그룹의 일반적인 형식은 다음과 같습니다:

```
display_name: [address-list];
```

그룹과 단일 주소의 혼합으로 구성된 주소 목록 처리의 편의를 위해, *display_name*을 `None`으로 설정하고 단일 주소 목록을 *addresses*로 제공하여, `Group`을 그룹의 일부가 아닌 단일 주소를 나타내는 데 사용할 수도 있습니다.

display_name

그룹의 display_name. None이고 addresses에 정확히 하나의 Address가 있으면, Group은 그룹에 속하지 않은 단일 주소를 나타냅니다.

addresses

그룹에 있는 주소를 나타내는 *Address* 객체의 비어있을 수 있는 튜플.

__str__()

Group의 str 값은 **RFC 5322**에 따라 포맷되지만, ASCII가 아닌 문자의 콘텐츠 전송 인코딩 (Content Transfer Encoding)은 없습니다. display_name이 None이고 addresses 목록에 단일 Address가 있으면 str 값은 해당 단일 Address의 str과 같습니다.

20.1.7 email.contentmanager: MIME 콘텐츠 관리

소스 코드: Lib/email/contentmanager.py

Added in version 3.6:[1]

class email.contentmanager.**ContentManager**

콘텐츠 관리자를 위한 베이스 클래스. get_content와 set_content 디스패치 메서드뿐만 아니라 MIME 콘텐츠와 다른 표현 간의 변환기를 등록하는 표준 등록소 메커니즘을 제공합니다.

get_content(*msg*, **args*, ***kw*)

*msg*의 mimetype을 기반으로 처리기 함수를 찾고 (다음 단락을 참조하십시오), 모든 인자를 전달하여 그것을 호출하고, 이 호출의 결과를 반환합니다. 처리기가 *msg*에서 페이 로드를 추출하고 추출된 데이터에 대한 정보를 인코딩하는 객체를 반환할 것으로 기대합니다.

처리기를 찾으려면, 등록소에서 다음 키를 찾는데, 처음 발견되는 것에서 멈춥니다:

- 전체 MIME 유형을 나타내는 문자열 (maintype/subtype)
- maintype을 나타내는 문자열
- 빈 문자열

이러한 키 중 아무것도 이러한 처리기를 생성하지 않으면, 전체 MIME 유형에 대해 *KeyError*를 발생시킵니다.

set_content(*msg*, *obj*, **args*, ***kw*)

maintype이 multipart이면, *TypeError*를 발생시킵니다; 그렇지 않으면 *obj*의 형을 기반으로 처리기 함수를 찾고 (다음 단락을 참조하십시오), *msg*에서 *clear_content()*를 호출한 다음, 모든 인자를 전달해서 처리기 함수를 호출합니다. 처리기가 *obj*를 *msg*로 변환하고 저장할 것으로 기대하는데, 저장된 데이터를 해석하는 데 필요한 정보를 인코딩하기 위해 다양한 MIME 헤더를 추가하는 등 *msg*에 다른 변경을 가할 수 있습니다.

처리기를 찾으려면, *obj*의 형을 얻고 (typ = type(obj)), 등록소에서 다음 키를 찾는데 처음 발견되는 것에서 멈춥니다:

- 형 자체 (typ)
- 형의 완전히 정규화된 이름 (typ.__module__ + '.' + typ.__qualname__).
- 형의 qualname (typ.__qualname__)
- 형의 이름 (typ.__name__).

이 중 아무것도 일치하지 않으면, *MRO* (typ.__mro__)의 각 형에 대해 위의 모든 검사를 반복합니다. 마지막으로, 다른 키가 처리기를 생성하지 않으면, None 키의 처리기를 확인합니다. None에 대한 처리기가 없으면, 형의 완전히 정규화된 이름으로 *KeyError*를 발생시킵니다.

MIME-Version 헤더가 없으면 추가합니다 (*MIMEPart*를 참조하십시오).

[1] 원래 3.4에서 잠정적 모듈로 추가되었습니다.

add_get_handler(*key*, *handler*)

함수 *handler*를 *key*의 처리기로 기록합니다. 가능한 *key* 값은 `get_content()`를 참조하십시오.

add_set_handler(*typekey*, *handler*)

*typekey*와 일치하는 형의 객체가 `set_content()`에 전달될 때 호출할 함수로 *handler*를 기록합니다. 가능한 *typekey* 값은 `set_content()`를 참조하십시오.

콘텐츠 관리자 인스턴스

현재 email 패키지는 하나의 구상 콘텐츠 관리자 `raw_data_manager`만 제공하지만, 향후에는 더 추가될 수 있습니다. `raw_data_manager`는 `EmailPolicy`와 그 파생물에 의해 제공되는 `content_manager`입니다.

email.contentmanager.**raw_data_manager**

이 콘텐츠 관리자는 `Message` 자체에서 제공하는 것 외에는 최소 인터페이스 만 제공합니다: 텍스트, 날 바이트열 및 `Message` 객체만 다룹니다. 그런데도 기본 API와 비교할 때 상당한 이점을 제공합니다: 텍스트 파트에 대한 `get_content`는 응용 프로그램이 수동으로 디코딩할 필요 없이 유니코드 문자열을 반환하고, `set_content`는 파트에 추가된 헤더를 제어하고 콘텐츠 전송 인코딩을 제어하기 위한 다양한 옵션을 제공하고, 다양한 `add_` 메서드를 사용할 수 있도록 해서, 멀티 파트 메시지 작성을 단순화합니다.

email.contentmanager.**get_content**(*msg*, *errors='replace'*)

파트의 페이 로드를 문자열(`text` 파트의 경우), `EmailMessage` 객체 (`message/rfc822` 파트의 경우) 또는 `bytes` 객체 (다른 모든 비 멀티 파트 유형의 경우)로 반환합니다. `multipart`에서 호출되면 `KeyError`를 발생시킵니다. 파트가 `text` 파트이고 *errors*가 지정되면, 페이 로드를 유니코드로 디코딩할 때 에러 처리기로 사용합니다. 기본 에러 처리기는 `replace`입니다.

email.contentmanager.**set_content**(*msg*, *<'str'>*, *subtype="plain"*, *charset='utf-8'*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

email.contentmanager.**set_content**(*msg*, *<'bytes'>*, *maintype*, *subtype*, *cte="base64"*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

email.contentmanager.**set_content**(*msg*, *<'EmailMessage'>*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)

*msg*에 헤더와 페이 로드를 추가합니다:

`maintype/subtype` 값으로 `Content-Type` 헤더를 추가합니다.

- `str`의 경우, MIME `maintype`을 `text`로 설정하고, 서브 유형은 지정되었으면 *subtype*으로 설정하고, 지정되지 않았으면 `plain`으로 설정합니다.
- `bytes`의 경우, 지정된 *maintype*과 *subtype*을 사용하거나, 지정되지 않았으면 `TypeError`를 발생시킵니다.
- `EmailMessage` 객체의 경우, 메인 유형을 `message`로 설정하고, 서브 유형은 지정되었으면 *subtype*으로 설정하고, 지정되지 않았으면 `rfc822`로 설정합니다. *subtype*이 `partial`이면 에러를 발생시킵니다 (`bytes` 객체를 사용하여 `message/partial` 파트를 구성해야 합니다).

*charset*이 제공되면 (`str`에만 유효합니다), 지정된 문자 집합을 사용하여 문자열을 바이트열로 인코딩합니다. 기본값은 `utf-8`입니다. 지정된 *charset*이 표준 MIME 문자 집합 이름의 알려진 별칭이면, 표준 문자 집합을 대신 사용합니다.

*cte*가 설정되면, 지정된 콘텐츠 전송 인코딩을 사용하여 페이 로드를 인코딩하고, `Content-Transfer-Encoding` 헤더를 해당 값으로 설정합니다. *cte*의 가능한 값은 `quoted-printable`, `base64`, `7bit`, `8bit` 및 `binary`입니다. 지정된 인코딩으로 입력을 인코딩할 수 없으면 (예를 들어, 비 ASCII 값을 포함하는 입력에 대해 *cte*를 `7bit`로 지정합니다), `ValueError`를 발생시킵니다.

- `str` 객체의 경우, *cte*가 설정되지 않으면 휴리스틱을 사용하여 가장 간결한 인코딩을 결정합니다. 인코딩하기 전에, 모든 줄 경계를 정규화하기 위해 `str.splitlines()`를 사용하여,

페이로드의 각 줄이 현재 정책의 *linesep* 프로퍼티로 끝나도록 합니다 (원래 문자열이 뭔가로 끝나지 않아도 마찬가지).

- For `bytes` objects, *cte* is taken to be base64 if not set, and the aforementioned newline translation is not performed.
- *EmailMessage*의 경우, **RFC 2046**에 따라, *subtype* `rfc822`에 대해 `quoted-printable`이나 `base64`의 *cte*가 요청되거나, *subtype* `external-body`에 대해 `7bit` 이외의 *cte*에 대해 에러를 발생시킵니다. `message/rfc822`의 경우, *cte*가 지정되지 않으면 `8bit`를 사용합니다. *subtype*의 다른 모든 값에는 `7bit`를 사용합니다.

> **참고**
>
> `binary`의 *cte*는 실제로 아직 제대로 작동하지 않습니다. `set_content`에 의해 수정된 `EmailMessage` 객체는 올바르지만, *BytesGenerator*는 이것을 올바르게 직렬화하지 않습니다.

*disposition*이 설정되면, 이를 *Content-Disposition* 헤더의 값으로 사용합니다. 지정되지 않고 *filename*이 지정되면, 값이 `attachment`인 헤더를 추가합니다. *disposition*이 지정되지 않고 *filename*도 지정되지 않으면, 헤더를 추가하지 않습니다. *disposition*에 유효한 값은 `attachment`와 `inline`뿐 입니다.

*filename*이 지정되면, 이를 *Content-Disposition* 헤더의 `filename` 파라미터의 값으로 사용합니다.

*cid*가 지정되면, *cid*를 값으로 사용하여 *Content-ID* 헤더를 추가합니다.

*params*가 지정되면, 그것의 `items` 메서드를 이터레이트하고 결과 `(key, value)` 쌍을 사용하여 *Content-Type* 헤더에 추가 파라미터를 설정합니다.

*headers*가 지정되고 `headername: headervalue` 형식의 문자열 리스트나 `header` 객체 (`name` 어트리뷰트를 가진 것으로 문자열과 구별됩니다) 리스트면, 헤더를 *msg*에 추가합니다.

20.1.8 email: 예제

다음은 *email* 패키지를 사용하여 간단한 전자 우편 메시지뿐만 아니라 더 복잡한 MIME 메시지를 읽고 쓰고 보내는 방법에 대한 몇 가지 예입니다.

먼저 간단한 텍스트 메시지를 만들고 보내는 방법을 살펴보겠습니다 (텍스트 내용과 주소에 유니코드 문자가 포함될 수 있습니다):

```
# 실제 전송 기능을 위해 smtplib를 임포트 합니다
import smtplib

# 필요한 email 모듈을 임포트 합니다
from email.message import EmailMessage

# textfile에 있는 이름의 텍스트 파일을 읽기 위해 엽니다.
with open(textfile) as fp:
    # Create a text/plain message
    msg = EmailMessage()
    msg.set_content(fp.read())

# me == 발신자의 전자 우편 주소
# you == 수신자의 전자 우편 주소
msg['Subject'] = f'The contents of {textfile}'
msg['From'] = me
msg['To'] = you

# 자체 SMTP 서버를 통해 메시지를 보냅니다.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

parser 모듈의 클래스를 사용하여 **RFC 822** 헤더를 쉽게 구문 분석할 수 있습니다:

```
# 필요한 email 모듈을 임포트 합니다
#from email.parser import BytesParser
from email.parser import Parser
from email.policy import default

# 전자 우편 헤더가 파일에 있으면, 이 두 줄의 주석 처리를 제거하십시오:
# with open(messagefile, 'rb') as fp:
#     headers = BytesParser(policy=default).parse(fp)

#  또는 문자열에서 헤더를 구문 분석하려면 (이는 드문 작업입니다), 다음을 사용하십시오:
headers = Parser(policy=default).parsestr(
        'From: Foo Bar <user@example.com>\n'
        'To: <someone_else@example.com>\n'
        'Subject: Test message\n'
        '\n'
        'Body would go here\n')

#  이제 헤더 항목을 딕셔너리로 액세스 할 수 있습니다:
print('To: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))

# 또한 주소의 일부에 액세스 할 수 있습니다:
print('Recipient username: {}'.format(headers['to'].addresses[0].username))
print('Sender name: {}'.format(headers['from'].addresses[0].display_name))
```

다음은 디렉터리에 있을 수 있는 가족사진을 포함하는 MIME 메시지를 보내는 방법의 예입니다:

```
# 실제 전송 기능을 위해 smtplib를 임포트 합니다.
import smtplib

# 다음은 필요한 email 패키지입니다.
from email.message import EmailMessage

# 컨테이너 전자 우편 메시지를 만듭니다.
msg = EmailMessage()
msg['Subject'] = 'Our family reunion'
# me == 발신자의 전자 우편 주소
# family = 모든 수신자의 전자 우편 주소 목록
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

# 바이너리 모드로 파일을 엽니다. MIMEImage 가 추측하길 원하면
# subtype 을 생략할 수도 있습니다.
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
    msg.add_attachment(img_data, maintype='image',
                                 subtype='png')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# 자체 SMTP 서버를 통해 전자 우편을 보냅니다.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

다음은 디렉터리의 전체 내용을 전자 우편 메시지로 보내는 방법의 예입니다:[1]

```
#!/usr/bin/env python3

"""디렉터리의 내용을 MIME 메시지로 보냅니다."""

import os
import smtplib
# 파일 이름 확장자를 기반으로 MIME 유형을 추측하기 위해
import mimetypes

from argparse import ArgumentParser

from email.message import EmailMessage
from email.policy import SMTP


def main():
    parser = ArgumentParser(description="""\
Send the contents of a directory as a MIME message.
Unless the -o option is given, the email is sent by forwarding to your local
SMTP server, which then does the normal delivery process.  Your local machine
must be running an SMTP server.
""")
    parser.add_argument('-d', '--directory',
                        help="""Mail the contents of the specified directory,
                        otherwise use the current directory.  Only the regular
                        files in the directory are sent, and we don't recurse to
                        subdirectories.""")
    parser.add_argument('-o', '--output',
                        metavar='FILE',
                        help="""Print the composed message to FILE instead of
                        sending the message to the SMTP server.""")
    parser.add_argument('-s', '--sender', required=True,
                        help='The value of the From: header (required)')
    parser.add_argument('-r', '--recipient', required=True,
                        action='append', metavar='RECIPIENT',
                        default=[], dest='recipients',
                        help='A To: header value (at least one required)')
    args = parser.parse_args()
    directory = args.directory
    if not directory:
        directory = '.'
    # 메시지를 만듭니다
    msg = EmailMessage()
    msg['Subject'] = f'Contents of directory {os.path.abspath(directory)}'
    msg['To'] = ', '.join(args.recipients)
    msg['From'] = args.sender
    msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

    for filename in os.listdir(directory):
```

(다음 페이지에 계속)

[1] 영감과 예를 주신 Matthew Dixon Cowles에게 감사드립니다.

(이전 페이지에서 계속)

```
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # 파일 확장자를 기준으로 콘텐츠 유형을 추측합니다.
        # gzip 되었거나 압축된 파일과 같은 간단한 사항을
        # 확인해야 하지만, 인코딩은 무시됩니다.
        ctype, encoding = mimetypes.guess_file_type(path)
        if ctype is None or encoding is not None:
            # 추측할 수 없거나 파일이 인코딩(압축)되었기 때문에,
            # 일반적인 비트 덩어리 유형을 사용합니다.
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        with open(path, 'rb') as fp:
            msg.add_attachment(fp.read(),
                               maintype=maintype,
                               subtype=subtype,
                               filename=filename)
    # 이제 메시지를 보내거나 저장합니다
    if args.output:
        with open(args.output, 'wb') as fp:
            fp.write(msg.as_bytes(policy=SMTP))
    else:
        with smtplib.SMTP('localhost') as s:
            s.send_message(msg)


if __name__ == '__main__':
    main()
```

다음은 위와 같은 MIME 메시지를 디렉터리로 푸는 방법의 예입니다:

```
#!/usr/bin/env python3

"""MIME 메시지를 디렉터리에 풉니다."""

import os
import email
import mimetypes

from email.policy import default

from argparse import ArgumentParser


def main():
    parser = ArgumentParser(description="""\
Unpack a MIME message into a directory of files.
""")
    parser.add_argument('-d', '--directory', required=True,
                        help="""Unpack the MIME message into the named
                        directory, which will be created if it doesn't already
                        exist.""")
    parser.add_argument('msgfile')
    args = parser.parse_args()

    with open(args.msgfile, 'rb') as fp:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        msg = email.message_from_binary_file(fp, policy=default)

    try:
        os.mkdir(args.directory)
    except FileExistsError:
        pass

    counter = 1
    for part in msg.walk():
        # multipart/* 는 단지 컨테이너입니다
        if part.get_content_maintype() == 'multipart':
            continue
        # 전자 우편 메시지를 사용하여 중요한 파일을 덮어쓸 수 없도록 응용 프로그램은
        # 주어진 파일명을 청소해야 합니다.
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # 일반적인 비트 덩어리 확장자를 사용합니다
                ext = '.bin'
            filename = f'part-{counter:03d}{ext}'
        counter += 1
        with open(os.path.join(args.directory, filename), 'wb') as fp:
            fp.write(part.get_payload(decode=True))


if __name__ == '__main__':
    main()
```

다음은 대체 일반 텍스트 버전으로 HTML 메시지를 만드는 방법의 예입니다. 좀 더 흥미롭게 하기 위해, html 부분에 관련 이미지를 포함하고, 보낼 뿐만 아니라, 보낼 것의 사본을 디스크에 저장합니다.

```
#!/usr/bin/env python3

import smtplib

from email.message import EmailMessage
from email.headerregistry import Address
from email.utils import make_msgid

# 기본 텍스트 메시지를 만듭니다.
msg = EmailMessage()
msg['Subject'] = "Pourquoi pas des asperges pour ce midi ?"
msg['From'] = Address("Pepé Le Pew", "pepe", "example.com")
msg['To'] = (Address("Penelope Pussycat", "penelope", "example.com"),
             Address("Fabrette Pussycat", "fabrette", "example.com"))
msg.set_content("""\
Salut!

Cette recette [1] sera sûrement un très bon repas.

[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718

--Pepé
""")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# html 버전을 추가합니다. 그러면 메시지가 원본 텍스트 메시지가
# 첫 번째 부분이고 새 html 메시지가 두 번째 부분인
# multipart/alternative 컨테이너로 변환됩니다.
asparagus_cid = make_msgid()
msg.add_alternative("""\
<html>
  <head></head>
  <body>
    <p>Salut!</p>
    <p>Cette
        <a href="http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718">
            recette
        </a> sera sûrement un très bon repas.
    </p>
    <img src="cid:{asparagus_cid}">
  </body>
</html>
""".format(asparagus_cid=asparagus_cid[1:-1]), subtype='html')
# html에서 사용하기 위해 msgid에서 <>를 벗겨야 했습니다.

# 이제 관련 이미지를 html 부분에 추가합니다.
with open("roasted-asparagus.jpg", 'rb') as img:
    msg.get_payload()[1].add_related(img.read(), 'image', 'jpeg',
                                     cid=asparagus_cid)

# 보낼 것의 지역 사본을 만듭니다.
with open('outgoing.msg', 'wb') as f:
    f.write(bytes(msg))

# 지역 SMTP 서버를 통해 메시지를 보냅니다.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

마지막 예에서 메시지를 보냈다면, 다음은 그것을 처리하는 한 가지 방법입니다:

```
import os
import sys
import tempfile
import mimetypes
import webbrowser

# 필요한 email 모듈을 임포트 합니다
from email import policy
from email.parser import BytesParser


def magic_html_parser(html_text, partfiles):
    """partfiles 에 링크된 안전 위생 처리된 html 을 반환합니다.

    partfiles의 파일명을 가리키도록 href="cid:...." 어트리뷰트를 다시 씁니다.
    사소하지는 않지만, html.parser 를 사용하면 가능합니다.
    """
    raise NotImplementedError("Add the magic needed")


# 실제 프로그램에서는 인자에서 파일명을 얻습니다.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# 이제 헤더 항목을 딕셔너리로 액세스할 수 있으며, 비 ASCII는
# 유니코드로 변환됩니다:
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# 메시지 내용의 미리 보기를 인쇄하려면, 가장 덜 포맷된 형식의
# 페이로드를 추출하고 처음 세 줄을 인쇄합니다. 물론, 메시지에
# 일반 텍스트 부분이 없을 때, html의 처음 세 줄을 인쇄하는 것은
# 쓸모가 없지만, 이것은 단지 개념적인 예일 뿐입니다.
simplest = msg.get_body(preferencelist=('plain', 'html'))
print()
print(''.join(simplest.get_content().splitlines(keepends=True)[:3]))

ans = input("View full message?")
if ans.lower()[0] == 'n':
    sys.exit()

# 표시하기 위해 가장 풍부한 대안을 추출할 수 있습니다:
richest = msg.get_body()
partfiles = {}
if richest['content-type'].maintype == 'text':
    if richest['content-type'].subtype == 'plain':
        for line in richest.get_content().splitlines():
            print(line)
        sys.exit()
    elif richest['content-type'].subtype == 'html':
        body = richest
    else:
        print("Don't know how to display {}".format(richest.get_content_type()))
        sys.exit()
elif richest['content-type'].content_type == 'multipart/related':
    body = richest.get_body(preferencelist=('html'))
    for part in richest.iter_attachments():
        fn = part.get_filename()
        if fn:
            extension = os.path.splitext(part.get_filename())[1]
        else:
            extension = mimetypes.guess_extension(part.get_content_type())
        with tempfile.NamedTemporaryFile(suffix=extension, delete=False) as f:
            f.write(part.get_content())
            # cid의 전자 우편 형식에서 html 형식으로 가기 위해 <>을 다시 제거합니다.
            partfiles[part['content-id'][1:-1]] = f.name
else:
    print("Don't know how to display {}".format(richest.get_content_type()))
    sys.exit()
with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
    f.write(magic_html_parser(body.get_content(), partfiles))
webbrowser.open(f.name)
os.remove(f.name)
for fn in partfiles.values():
    os.remove(fn)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# 물론, 이 간단한 프로그램을 망칠 수 있는 많은 전자 우편
# 메시지가 있지만, 가장 흔한 것을 처리합니다.
```

프롬프트까지, 위의 출력은 다음과 같습니다:

```
To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat <fabrette@example.
↪com>
From: Pepé Le Pew <pepe@example.com>
Subject: Pourquoi pas des asperges pour ce midi ?

Salut!

Cette recette [1] sera sûrement un très bon repas.
```

레거시 API:

20.1.9 email.message.Message: compat32 API를 사용하여 이메일 메시지 표현하기

Message 클래스는 *EmailMessage* 클래스와 매우 유사지만, 이 클래스에 의해 추가된 메서드가 없고, 다른 특정 메서드의 기본 동작이 약간 다릅니다. 또한 *EmailMessage* 클래스에서 지원하지만, 레거시 코드를 다루지 않는 한 권장되지 않는 일부 메서드도 여기에서 설명합니다.

두 클래스의 철학과 구조는 그 외에는 같습니다.

이 문서는 기본 (*Message*의 경우) 정책 *Compat32*의 동작을 설명합니다. 다른 정책을 사용하려면, *EmailMessage* 클래스를 대신 사용해야 합니다.

이메일 메시지는 헤더(*headers*)와 페이 로드(*payload*)로 구성됩니다. 헤더는 **RFC 5322** 스타일 이름과 값이어야 하며, 필드 이름과 값은 콜론으로 구분됩니다. 콜론은 필드 이름이나 필드 값의 일부가 아닙니다. 페이 로드는 간단한 텍스트 메시지, 바이너리 객체 또는 각각 자체 헤더 집합과 자체 페이 로드가 있는 서브 메시지의 구조화된 시퀀스일 수 있습니다. 후자의 페이 로드 유형은 *multipart/**나 *message/rfc822*와 같은 MIME 유형을 가진 메시지로 표시됩니다.

Message 객체가 제공하는 개념 모델은 헤더의 순서 있는 딕셔너리이면서, 헤더의 특수 정보에 액세스하고, 페이 로드에 액세스하고, 메시지의 직렬화된 버전을 생성하고, 객체 트리를 재귀적으로 탐색하기 위한 추가 메서드가 제공되는 것입니다. 중복 헤더가 지원되지만, 특수한 메서드를 사용하여 액세스해야 함에 유의하십시오.

Message 의사 딕셔너리는 헤더 이름으로 인덱싱되며, ASCII 값이어야 합니다. 딕셔너리의 값은 ASCII 문자만 포함해야 하는 문자열입니다; 비 ASCII 입력에 대한 특수 처리가 있지만, 항상 올바른 결과를 생성하지는 않습니다. 헤더는 대소 문자를 유지하면서 저장되고 반환되지만, 필드 이름은 대소 문자를 구분하지 않고 일치합니다. *Unix-From* 헤더나 `From_` 헤더라고도 하는 단일 봉투 헤더가 있을 수도 있습니다. 페이 로드(*payload*)는 단순 메시지 객체의 경우는 문자열이나 바이트열이고, MIME 컨테이너 문서(예를 들어 *multipart/**와 *message/rfc822*)의 경우는 *Message* 객체의 리스트입니다.

Message 클래스의 메서드는 다음과 같습니다:

class email.message.**Message**(*policy=compat32*)

*policy*가 지정되면 (*policy* 클래스의 인스턴스여야 합니다) 메시지 표현을 갱신하고 직렬화하기 위해 지정된 규칙을 사용합니다. *policy*가 설정되지 않으면, *compat32* 정책을 사용하는데, 파이썬 3.2 버전의 email 패키지와의 과거 호환성을 유지합니다. 자세한 내용은 *policy* 설명서를 참조하십시오.

버전 3.3에서 변경: *policy* 키워드 인자가 추가되었습니다.

as_string(*unixfrom=False, maxheaderlen=0, policy=None*)

평활화된 전체 메시지를 문자열로 반환합니다. 선택적 *unixfrom*이 참이면, 봉투 헤더가 반환된 문자열에 포함됩니다. *unixfrom*의 기본값은 `False`입니다. 이전 버전과의 호환성을 위해, *maxheaderlen*의 기본값이 `0`이라서 다른 값을 원하면 명시적으로 재정의해야 합니다(정책에서 *max_line_length*에 지정된 값은 이 메서드가 무시합니다). *policy* 인자는 메시지 인스턴스에서 얻은 기본 정책을 대체하는 데 사용될 수 있습니다. 지정된 *policy*가 `Generator`로 전달되므로, 메서드에서 생성된 일부 포매팅을 제어하는 데 사용할 수 있습니다.

문자열로의 변환을 완료하기 위해 기본값을 채워야 하면 메시지 평활화는 *Message*에 대한 변경을 유발할 수 있습니다 (예를 들어, MIME 경계가 생성되거나 수정될 수 있습니다).

이 메서드는 편의상 제공되며 원하는 방식으로 메시지를 항상 포맷하지는 않음에 유의하십시오. 예를 들어, 기본적으로 유닉스 mbox 형식에 필요한 `From`으로 시작하는 줄을 맹글링(mangling)하지 않습니다. 유연성을 높이려면, *Generator* 인스턴스를 인스턴스 화하고 그것의 *flatten()* 메서드를 직접 사용하십시오. 예를 들면:

```
from io import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

메시지 객체가 RFC 표준에 따라 인코딩되지 않은 바이너리 데이터를 포함하면, 비 호환 데이터는 유니코드 "unknown character" 코드 포인트로 대체됩니다. (*as_bytes()*와 *BytesGenerator*도 참조하십시오.)

버전 3.4에서 변경: *policy* 키워드 인자가 추가되었습니다.

__str__()

*as_string()*과 동등합니다. `str(msg)` 가 포맷된 메시지를 포함하는 문자열을 생성할 수 있도록 합니다.

as_bytes(*unixfrom=False*, *policy=None*)

평활화된 전체 메시지를 바이트열 객체로 반환합니다. 선택적 *unixfrom*이 참이면, 봉투 헤더가 반환된 문자열에 포함됩니다. *unixfrom*의 기본값은 `False`입니다. *policy* 인자는 메시지 인스턴스에서 얻은 기본 정책을 대체하는 데 사용될 수 있습니다. 지정된 *policy*가 `BytesGenerator`로 전달되므로, 메서드에서 생성된 일부 포매팅을 제어하는 데 사용할 수 있습니다.

문자열로의 변환을 완료하기 위해 기본값을 채워야 하면 메시지 평활화는 *Message*에 대한 변경을 유발할 수 있습니다 (예를 들어, MIME 경계가 생성되거나 수정될 수 있습니다).

이 메서드는 편의상 제공되며 원하는 방식으로 메시지를 항상 포맷하지는 않음에 유의하십시오. 예를 들어, 기본적으로 유닉스 mbox 형식에 필요한 `From`으로 시작하는 줄을 맹글링(mangling)하지 않습니다. 유연성을 높이려면, *BytesGenerator* 인스턴스를 인스턴스 화하고 그것의 *flatten()* 메서드를 직접 사용하십시오. 예를 들면:

```
from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
g = BytesGenerator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

Added in version 3.4.

__bytes__()

*as_bytes()*와 동등합니다. `bytes(msg)` 가 포맷된 메시지를 포함하는 바이트열 객체를 생성할 수 있도록 합니다.

Added in version 3.4.

is_multipart()

메시지의 페이 로드가 서브-*Message* 객체의 리스트이면 `True`를, 그렇지 않으면 `False`를 반환합니다. *is_multipart()*가 `False`를 반환할 때, 페이 로드는 문자열 객체(CTE로 인코딩된 바이너리 페이 로드일 수 있습니다)여야 합니다. (`True`를 반환하는 *is_multipart()*가 반드시 "msg.get_content_maintype() == 'multipart'" 가 `True`를 반환한다는 것을 의미하지는 않습니다. 예를 들어, *Message*가 `message/rfc822` 유형일 때, `is_multipart`는 `True`를 반환합니다.

set_unixfrom(*unixfrom*)

메시지의 봉투 헤더를 (문자열이어야 하는) *unixfrom*으로 설정합니다.

get_unixfrom()

메시지의 봉투 헤더를 반환합니다. 봉투 헤더가 설정되지 않았으면 기본값은 `None`입니다.

attach(*payload*)

지정된 *payload*를 현재 페이 로드에 추가합니다. 호출 전에는 페이 로드가 `None`이나 `Message` 객체의 리스트여야 합니다. 호출 후에는 페이 로드가 항상 `Message` 객체의 리스트입니다. 페이 로드를 스칼라 객체(예를 들어 문자열)로 설정하려면, `set_payload()`를 대신 사용하십시오.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by `set_content()` and the related `make` and `add` methods.

get_payload(*i=None*, *decode=False*)

현재 페이 로드를 반환합니다. `is_multipart()`가 `True`이면 `Message` 객체의 리스트이고, `is_multipart()`가 `False`이면 문자열입니다. 페이 로드가 리스트이고 리스트 객체를 변경하면, 메시지의 페이 로드를 제자리에서 수정하는 것입니다.

선택적 인자 *i*를 사용하면, `get_payload()`는 `is_multipart()`가 `True`일 때, 페이 로드의 *i* 번째 요소를 0부터 세어 반환합니다. *i*가 0보다 작거나 페이 로드의 항목 수보다 크거나 같으면 `IndexError`가 발생합니다. 페이 로드가 문자열이고 (즉 `is_multipart()`가 `False`) *i*가 제공되면, `TypeError`가 발생합니다.

선택적 *decode*는 `Content-Transfer-Encoding` 헤더에 따라 페이 로드를 디코딩해야 하는지를 나타내는 플래그입니다. `True`이고 메시지가 멀티 파트가 아닐 때, 이 헤더 값이 `quoted-printable`이나 `base64`이면 페이 로드가 디코딩됩니다. 다른 인코딩이 사용되거나 `Content-Transfer-Encoding` 헤더가 누락되면, 페이 로드는 그대로 (디코딩되지 않고) 반환됩니다. 모든 경우에 반환된 값은 바이너리 데이터입니다. 메시지가 멀티 파트이고 *decode* 플래그가 `True`이면, `None`이 반환됩니다. 페이 로드가 base64이고 완벽하게 구성되지 않았으면 (패딩 누락, base64 알파벳 이외의 문자), 메시지의 결함 프로퍼티에 적절한 결함이 추가됩니다 (각각 `InvalidBase64PaddingDefect`나 `InvalidBase64CharactersDefect`).

*decode*가 `False`(기본값)일 때 본문은 `Content-Transfer-Encoding`을 디코딩하지 않고 문자열로 반환됩니다. 그러나, `Content-Transfer-Encoding`이 8비트인 경우, `replace` 에러 처리기로, `Content-Type` 헤더에 지정된 `charset`을 사용하여 원래 바이트를 디코딩하려고 시도합니다. `charset`이 지정되지 않았거나 email 패키지가 `charset`을 인식하지 못하면, 본문은 기본 ASCII 문자 집합을 사용하여 디코딩됩니다.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by `get_content()` and `iter_parts()`.

set_payload(*payload*, *charset=None*)

전체 메시지 객체의 페이 로드를 *payload*로 설정합니다. 페이 로드 불변량(invariants)을 보장하는 것은 고객의 책임입니다. 선택적 *charset*은 메시지의 기본 문자 집합을 설정합니다; 자세한 내용은 `set_charset()`을 참조하십시오.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by `set_content()`.

set_charset(*charset*)

페이 로드의 문자 집합을 *charset*으로 설정합니다. 이는 `Charset` 인스턴스 (`email.charset`을 참조하십시오), 문자 집합의 이름을 지정하는 문자열 또는 `None`일 수 있습니다. 문자열이면, `Charset` 인스턴스로 변환됩니다. *charset*이 `None`이면, `charset` 매개 변수가 `Content-Type` 헤더에서 제거됩니다 (그렇지 않으면 메시지가 수정되지 않습니다). 그 외의 경우는 `TypeError`를 생성합니다.

기존 `MIME-Version` 헤더가 없으면 추가됩니다. 기존 `Content-Type` 헤더가 없으면, `text/plain` 값으로 추가됩니다. `Content-Type` 헤더가 존재하는지와 관계없이, `charset` 매개 변수는 *charset.output_charset*으로 설정됩니다. *charset.input_charset*과 *charset.output_charset*이 다르면, 페이 로드가 *output_charset*으로 다시 인코딩됩니다. 기존 `Content-Transfer-Encoding` 헤더가 없으면, 필요하면 지정된 `Charset`을 사용하여 페이 로드가 전송 인코딩되고(transfer-encoded), 적절한 값을 가진 헤더가 추가됩니다. `Content-Transfer-Encoding` 헤더가 이미 존재하면,

페이로드는 해당 *Content-Transfer-Encoding*을 사용하여 이미 올바르게 인코딩된 것으로 가정하며 수정하지 않습니다.

This is a legacy method. On the *EmailMessage* class its functionality is replaced by the *charset* parameter of the *email.message.EmailMessage.set_content()* method.

get_charset()

메시지 페이로드와 연관된 *Charset* 인스턴스를 반환합니다.

This is a legacy method. On the *EmailMessage* class it always returns `None`.

다음 메서드는 메시지의 **RFC 2822** 헤더에 액세스하기 위한 매핑과 유사한 인터페이스를 구현합니다. 이 메서드들과 일반적인 매핑 (즉, 딕셔너리) 인터페이스 사이에는 의미상 차이가 있습니다. 예를 들어, 딕셔너리에는 중복 키가 없지만, 여기에는 중복 메시지 헤더가 있을 수 있습니다. 또한, 딕셔너리에서는 *keys()*가 반환하는 키의 순서가 보장되지 않지만, *Message* 객체에서는 헤더가 항상 원래 메시지에 나타나거나 나중에 메시지에 추가된 순서대로 반환됩니다. 삭제한 후 다시 추가된 헤더는 항상 헤더 리스트의 끝에 추가됩니다.

이러한 의미상 차이는 의도적이며 최대한의 편의를 위해 편향되어 있습니다.

모든 경우에, 메시지에 존재하는 봉투 헤더는 매핑 인터페이스에 포함되지 않습니다.

바이트열에서 생성된 모델에서, (RFC에 반하여) 비 ASCII 바이트를 포함하는 헤더 값은, 이 인터페이스를 통해 꺼낼 때, 문자 집합이 `unknown-8bit`인 *Header* 객체로 표시됩니다.

__len__()

중복을 포함하여, 총 헤더 수를 반환합니다.

__contains__(*name*)

메시지 객체에 *name*이라는 이름의 필드가 있으면 `True`를 반환합니다. 대소 문자를 구분하지 않고 일치하며 *name*은 후행 콜론을 포함하지 않아야 합니다. `in` 연산자에 사용됩니다, 예를 들어:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)

명명된 헤더 필드의 값을 반환합니다. *name*은 콜론 필드 구분자를 포함하지 않아야 합니다. 헤더가 없으면, `None`이 반환됩니다; *KeyError*는 절대 발생하지 않습니다.

이름이 지정된 필드가 메시지 헤더에 두 번 이상 나타나면, 해당 필드 값 중 정확히 어떤 필드 값이 반환되는지 정의되지 않습니다. 기존의 모든 명명된 헤더의 값을 가져오려면 *get_all()* 메서드를 사용하십시오.

__setitem__(*name*, *val*)

메시지에 필드 이름이 *name*이고 값이 *val*인 헤더를 추가합니다. 필드는 메시지의 기존 필드 끝에 추가됩니다.

이것이 같은 이름을 가진 기존 헤더를 덮어쓰거나 삭제하지 않음에 유의하십시오. 새 헤더가 메시지에 필드 이름이 *name*인 유일한 헤더가 되도록 하려면, 먼저 필드를 삭제하십시오. 예를 들어:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(*name*)

메시지 헤더에서 이름이 *name*인 모든 필드를 삭제합니다. 명명된 필드가 헤더에 없어도 예외가 발생하지 않습니다.

keys()

메시지 헤더의 모든 필드 이름의 리스트를 반환합니다.

values()

메시지의 모든 필드 값의 리스트를 반환합니다.

items()

메시지의 모든 필드 헤더와 값을 포함하는 2-튜플의 리스트를 반환합니다.

get(*name*, *failobj=None*)

명명된 헤더 필드의 값을 반환합니다. 명명된 헤더가 없으면 선택적 *failobj*가 반환된다는 점을 제외하고는 `__getitem__()`과 동일합니다 (기본값은 `None`입니다).

몇 가지 유용한 추가 메서드가 있습니다:

get_all(*name*, *failobj=None*)

*name*이라는 필드의 모든 값 리스트를 반환합니다. 메시지에 이런 이름의 헤더가 없으면, *failobj*가 반환됩니다 (기본값은 `None`입니다).

add_header(*_name*, *_value*, ***_params*)

확장된 헤더 설정. 이 메서드는 추가 헤더 매개 변수가 키워드 인자로 제공될 수 있다는 점을 제외하고는 `__setitem__()`과 유사합니다. *_name*은 추가할 헤더 필드이고 *_value*는 헤더의 기본(*primary*) 값입니다.

키워드 인자 딕셔너리 *_params*의 각 항목에 대해, 키는 매개 변수 이름으로 사용되며, 밑줄은 대시로 변환됩니다 (대시는 파이썬 식별자로 유효하지 않기 때문입니다). 일반적으로, 값이 `None`이 아니면 매개 변수가 `key="value"`로 추가되며, `None`이면 키만 추가됩니다. 값에 ASCII가 아닌 문자가 포함되면, `(CHARSET, LANGUAGE, VALUE)` 형식으로 3-튜플로 지정할 수 있습니다. 여기서 `CHARSET`은 값을 인코딩하는 데 사용할 문자 집합의 이름을 지정하는 문자열이고, `LANGUAGE`는 일반적으로 `None`이나 빈 문자열(다른 가능성에 대해서는 **RFC 2231**을 참조하십시오)로 설정될 수 있고, `VALUE`는 비 ASCII 코드 포인트를 포함하는 문자열 값입니다. 3-튜플이 전달되지 않고 값에 ASCII가 아닌 문자가 포함되면, `CHARSET`으로 `utf-8`을 `LANGUAGE`로 `None`을 사용하여 **RFC 2231** 형식으로 자동 인코딩됩니다.

예를 들면 다음과 같습니다:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

이것은 다음과 같은 헤더를 추가합니다

```
Content-Disposition: attachment; filename="bud.gif"
```

비 ASCII 문자의 예:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

이것은 다음을 생성합니다

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

replace_header(*_name*, *_value*)

헤더를 교체합니다. 헤더 순서와 필드 이름 대소 문자를 유지하면서, 메시지에서 *_name*과 일치하는 첫 번째로 발견된 헤더를 교체합니다. 일치하는 헤더가 없으면, `KeyError`가 발생합니다.

get_content_type()

메시지의 콘텐츠 유형을 반환합니다. 반환된 문자열은 *maintype/subtype* 형식의 소문자로 강제 변환됩니다. 메시지에 *Content-Type* 헤더가 없으면 `get_default_type()`에서 제공하는 기본 유형이 반환됩니다. **RFC 2045**에 따르면, 메시지는 항상 기본 유형을 가지므로, `get_content_type()`은 항상 값을 반환합니다.

RFC 2045는 메시지가 *multipart/digest* 컨테이너 안에 등장(이 경우 기본 유형은 *message/rfc822*가 됩니다)하지 않는 한 기본 유형을 *text/plain*으로 정의합니다. *Content-Type* 헤더에 유효하지 않은 유형 명세가 있으면, **RFC 2045**는 기본 유형이 *text/plain*으로 지정합니다.

get_content_maintype()

메시지의 주 콘텐츠 유형을 반환합니다. 이것은 *get_content_type()*이 반환하는 문자열의 *maintype* 부분입니다.

get_content_subtype()

메시지의 부 콘텐츠 유형을 반환합니다. 이것은 *get_content_type()*이 반환하는 문자열의 *subtype* 부분입니다.

get_default_type()

기본 콘텐츠 유형을 반환합니다. *multipart/digest* 컨테이너의 서브 파트인 메시지를 제외하고, 대부분의 메시지는 기본 콘텐츠 유형이 *text/plain*입니다. 이러한 서브 파트는 기본 콘텐츠 유형이 *message/rfc822*입니다.

set_default_type(*ctype*)

기본 콘텐츠 유형을 설정합니다. *ctype*은 *text/plain*이나 *message/rfc822*여야 하지만 강제하지는 않습니다. 기본 콘텐츠 유형은 *Content-Type* 헤더에 저장되지 않습니다.

get_params(*failobj=None*, *header='content-type'*, *unquote=True*)

메시지의 *Content-Type* 매개 변수를 리스트로 반환합니다. 반환된 리스트의 요소는 `'='` 부호로 분할된 키/값 쌍의 2-튜플입니다. `'='`의 왼쪽은 키이고 오른쪽은 값입니다. 매개 변수에 `'='` 부호가 없으면 값은 빈 문자열이고, 그렇지 않으면 값은 *get_param()*에 설명된대로이며 선택적 *unquote*가 `True`(기본값)이면 인용되지 않습니다.

선택적 *failobj*는 *Content-Type* 헤더가 없을 때 반환할 객체입니다. 선택적 *header*는 *Content-Type* 대신 검색할 헤더입니다.

This is a legacy method. On the *EmailMessage* class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

get_param(*param*, *failobj=None*, *header='content-type'*, *unquote=True*)

Content-Type 헤더의 매개 변수 *param*의 값을 문자열로 반환합니다. 메시지에 *Content-Type* 헤더가 없거나 그러한 매개 변수가 없으면, *failobj*가 반환됩니다 (기본값은 `None`입니다).

선택적 *header*가 제공되면, *Content-Type* 대신 사용할 메시지 헤더를 지정합니다.

매개 변수 키는 항상 대소 문자를 구분하지 않고 비교됩니다. 반환 값은 문자열이거나, 매개 변수가 **RFC 2231**로 인코딩되었으면 3-튜플일 수 있습니다. 3-튜플일 때, 값의 요소는 `(CHARSET, LANGUAGE, VALUE)` 형식입니다. `CHARSET`과 `LANGUAGE`는 모두 `None`일 수 있으며, 이 경우 `VALUE`가 `us-ascii` 문자 집합으로 인코딩된 것으로 간주해야 함에 유의하십시오. 일반적으로 `LANGUAGE`를 무시할 수 있습니다.

응용 프로그램이 **RFC 2231**로 매개 변수가 인코딩되었는지를 신경 쓰지 않으면, *email.utils.collapse_rfc2231_value()*를 호출하면서 *get_param()*의 반환 값을 전달하여 매개 변숫값을 축소할 수 있습니다. 이것은 값이 튜플이면 적절하게 디코딩된 유니코드 문자열을 반환하고, 그렇지 않으면 원래 문자열을 인용 없이 반환합니다. 예를 들면:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

모든 경우에, *unquote*가 `False`로 설정되어 있지 않으면, 매개 변숫값(반환된 문자열이나 3-튜플의 `VALUE` 항목)은 항상 인용되지 않습니다.

This is a legacy method. On the *EmailMessage* class its functionality is replaced by the *params* property of the individual header objects returned by the header access methods.

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)

Content-Type 헤더에서 매개 변수를 설정합니다. 매개 변수가 이미 헤더에 존재하면, 해당 값은 *value*로 대체됩니다. 이 메시지에 대해 *Content-Type* 헤더가 아직 정의되지 않았으면, *text/plain*으로 설정되고 **RFC 2045**에 따라 새 매개 변숫값이 추가됩니다.

선택적 *header*는 *Content-Type*의 대체 헤더를 지정하며, 선택적 *requote*가 `False`(기본값은 `True`)가 아닌 한 모든 매개 변수가 필요에 따라 인용됩니다.

선택적 *charset*이 지정되면, 매개 변수는 **RFC 2231**에 따라 인코딩됩니다. 선택적 *language*는 RFC 2231 언어를 지정하며, 기본값은 빈 문자열입니다. *charset*과 *language*는 모두 문자열이어야 합니다.

*replace*가 `False`(기본값)이면 헤더가 헤더 리스트의 끝으로 이동합니다. *replace*가 `True`이면, 헤더는 제자리에서 갱신됩니다.

버전 3.4에서 변경: `replace` 키워드가 추가되었습니다.

del_param(*param*, *header='content-type'*, *requote=True*)

`Content-Type` 헤더에서 지정된 매개 변수를 완전히 제거합니다. 매개 변수나 그 값 없이 헤더가 다시 작성됩니다. *requote*가 `False`(기본값은 `True`)가 아닌 한 모든 값이 필요에 따라 인용됩니다. 선택적 *header*는 `Content-Type`의 대체 헤더를 지정합니다.

set_type(*type*, *header='Content-Type'*, *requote=True*)

`Content-Type` 헤더의 주 유형과 부 유형을 설정합니다. *type*은 `maintype/subtype` 형식의 문자열이어야 합니다, 그렇지 않으면 `ValueError`가 발생합니다.

이 메서드는 모든 매개 변수를 그대로 유지하면서 `Content-Type` 헤더를 대체합니다. *requote*가 `False`이면, 기존 헤더의 인용을 그대로 두고, 그렇지 않으면 매개 변수가 인용됩니다 (기본값).

header 인자에 대체 헤더를 지정할 수 있습니다. `Content-Type` 헤더가 설정될 때 `MIME-Version` 헤더도 추가됩니다.

This is a legacy method. On the `EmailMessage` class its functionality is replaced by the `make_` and `add_` methods.

get_filename(*failobj=None*)

메시지의 `Content-Disposition` 헤더의 `filename` 매개 변숫값을 반환합니다. 헤더에 `filename` 매개 변수가 없으면, 이 메서드는 `Content-Type` 헤더에서 `name` 매개 변수를 찾는 것으로 폴 백합니다. 둘 다 없거나, 헤더가 없으면, *failobj*가 반환됩니다. 반환된 문자열은 항상 `email.utils.unquote()`로 인용 해제됩니다.

get_boundary(*failobj=None*)

메시지의 `Content-Type` 헤더의 `boundary` 매개 변숫값이나, 헤더가 없거나 `boundary` 매개 변수가 없으면 *failobj*를 반환합니다. 반환된 문자열은 항상 `email.utils.unquote()`로 인용 해제됩니다.

set_boundary(*boundary*)

`Content-Type` 헤더의 `boundary` 매개 변수를 *boundary*로 설정합니다. 필요하면 `set_boundary()`는 항상 *boundary*를 인용합니다. 메시지 객체에 `Content-Type` 헤더가 없으면 `HeaderParseError` 가 발생합니다.

`set_boundary()`는 헤더 리스트에서 `Content-Type` 헤더의 순서를 유지하므로, 이 메서드를 사용하는 것은 이전 `Content-Type` 헤더를 삭제하고 `add_header()`를 통해 새 경계를 가진 새 헤더를 추가하는 것과 미묘하게 다름에 유의하십시오. 그러나 원래 `Content-Type` 헤더에 있을 수 있는 이어지는 줄(continuation lines)을 유지하지 않습니다.

get_content_charset(*failobj=None*)

`Content-Type` 헤더의 `charset` 매개 변수를 소문자로 강제 변환하여 반환합니다. `Content-Type` 헤더가 없거나, 해당 헤더에 `charset` 매개 변수가 없으면 *failobj*가 반환됩니다.

이 메서드는 메시지 본문의 기본 인코딩에 대한 `Charset` 인스턴스를 반환하는 `get_charset()`과 다름에 유의하십시오.

get_charsets(*failobj=None*)

메시지 내의 문자 집합 이름을 포함하는 리스트를 반환합니다. 메시지가 `multipart`이면, 리스트에 페이 로드의 각 서브 파트마다 하나의 요소가 포함되며, 그렇지 않으면 길이 1인 리스트가 됩니다.

리스트의 각 항목은 표현된 서브 파트에 대한 `Content-Type` 헤더의 `charset` 매개 변숫값인 문자열입니다. 그러나, 서브 파트에 `Content-Type` 헤더가 없거나, `charset` 매개 변수가 없거나, `text` 주 MIME 유형이 아니면, 반환된 목록의 해당 항목은 *failobj*가 됩니다.

get_content_disposition()

있다면 메시지의 *Content-Disposition* 헤더의 (매개 변수가 없는) 소문자 값을, 그렇지 않으면 None을 반환합니다. 메시지가 **RFC 2183**을 따른다면 이 메서드의 가능한 값은 *inline*, *attachment* 또는 None 입니다.

Added in version 3.5.

walk()

walk() 메서드는 메시지 객체 트리의 모든 파트와 서브 파트를 깊이 우선 탐색 순서로 이터레이트 하는 데 사용할 수 있는 범용 제너레이터입니다. 일반적으로 *walk()*를 for 루프에서 이터레이터로 사용합니다; 각 이터레이션은 다음 서브 파트를 반환합니다.

다음은 멀티 파트 메시지 구조의 모든 파트에 대한 MIME 유형을 인쇄하는 예입니다:

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

msg.get_content_maintype() == 'multipart'가 False를 반환하더라도, walk는 *is_multipart()*가 True를 반환하는 모든 파트의 서브 파트를 이터레이트 합니다. _structure 디버그 도우미 함수를 사용하여 예제에서 이를 확인할 수 있습니다:

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

여기서 message 파트는 multiparts가 아니지만, 서브 파트를 포함합니다. is_multipart()는 True를 반환하고 walk는 서브 파트로 내려갑니다.

Message 객체는 MIME 메시지의 단순 텍스트를 생성할 때 사용할 수 있는 두 개의 인스턴스 어트리뷰트를 선택적으로 포함할 수 있습니다.

preamble

MIME 문서의 형식은 헤더 다음의 빈 줄과 첫 번째 멀티 파트 경계 문자열 사이에 일부 텍스트를 허용합니다. 일반적으로, 이 텍스트는 표준 MIME 장비를 벗어나기 때문에 MIME 인식 메일 리더에서 볼 수 없습니다. 그러나, 메시지의 원시 텍스트를 보거나, MIME을 인식하지 않는 리더에서 메시지를 볼 때, 이 텍스트가 보일 수 있습니다.

preamble 어트리뷰트에는 MIME 문서에 대한 이 선행 추가 텍스트가 포함됩니다. *Parser*가 헤더 다음이지만 첫 번째 경계 문자열 이전에 어떤 텍스트를 감지하면, 이 텍스트를 메시지의

preamble 어트리뷰트에 대입합니다. `Generator`가 MIME 메시지의 단순 텍스트 표현을 기록할 때, 메시지에 *preamble* 어트리뷰트가 있는 것을 발견하면, 헤더와 첫 번째 경계 사이의 영역에 이 텍스트를 기록합니다. 자세한 내용은 `email.parser`와 `email.generator`를 참조하십시오.

메시지 객체에 프리앰블이 없으면, *preamble* 어트리뷰트는 `None`이 됨에 유의하십시오.

epilogue

epilogue 어트리뷰트는 메시지의 마지막 경계와 끝 사이에 나타나는 텍스트를 포함한다는 점을 제외하고, *preamble* 어트리뷰트와 같은 방식으로 작동합니다.

`Generator`가 파일 끝에서 줄 넘김을 인쇄하도록 하기 위해 에필로그를 빈 문자열로 설정할 필요는 없습니다.

defects

defects 어트리뷰트는 이 메시지를 구문 분석할 때 발견된 모든 결함의 리스트를 포함합니다. 가능한 구문 분석 결함에 대한 자세한 설명은 `email.errors`를 참조하십시오.

20.1.10 email.mime: 처음부터 이메일과 MIME 객체 만들기

소스 코드: Lib/email/mime/

이 모듈은 레거시 (`Compat32`) 이메일 API의 일부입니다. 새로운 API에서는 기능이 `contentmanager`로 부분적으로 대체되지만, 특정 응용 프로그램에서는 이 클래스들이 레거시 코드가 아닌 곳에서도 여전히 유용할 수 있습니다.

일반적으로, 파일이나 일부 텍스트를, 텍스트를 구문 분석하고 루트 메시지 객체를 반환하는 구문 분석기에 전달하여 메시지 객체 구조를 얻습니다. 그러나 처음부터 완전한 메시지 구조를 만들거나, 개별 `Message` 객체를 직접 만들 수도 있습니다. 실제로, 기존 구조를 가져와서 새로운 `Message` 객체를 추가하거나 이동시킬 수도 있습니다. 이렇게 하면 MIME 메시지를 잘게 자르고 재배치하는 매우 편리한 인터페이스가 만들어집니다.

`Message` 인스턴스를 만들고 첨부 파일과 모든 적절한 헤더를 수동으로 추가하여 새 객체 구조를 만들 수 있습니다. 그러나 MIME 메시지의 경우, `email` 패키지는 작업을 쉽게 하기 위해 편리한 서브 클래스를 제공합니다.

클래스는 다음과 같습니다:

class email.mime.base.**MIMEBase**(*_maintype, _subtype, *, policy=compat32, **_params*)

모듈: `email.mime.base`

이것은 `Message`의 모든 MIME 특정 서브 클래스의 베이스 클래스입니다. 일반적으로, 할 수는 있지만, `MIMEBase`의 인스턴스를 만들지는 않습니다. `MIMEBase`는 주로 더 구체적인 MIME 인식 서브 클래스를 위한 편리한 베이스 클래스로 제공됩니다.

*_maintype*은 `Content-Type` 주 유형(예를 들어 `text`나 `image`)이고, *_subtype*은 `Content-Type` 부 유형(예를 들어 `plain`이나 `gif`)입니다. *_params*는 매개 변수 키/값 딕셔너리이며 `Message.add_header`로 직접 전달됩니다.

*policy*가 지정되면 (기본값은 `compat32` 정책입니다), `Message`로 전달됩니다.

`MIMEBase` 클래스는 항상 (*_maintype*, *_subtype* 및 *_params*에 기반하는) `Content-Type` 헤더와 (항상 `1.0`으로 설정되는) `MIME-Version` 헤더를 추가합니다.

버전 3.6에서 변경: *policy* 키워드 전용 매개 변수를 추가했습니다.

class email.mime.nonmultipart.**MIMENonMultipart**

모듈: `email.mime.nonmultipart`

`MIMEBase`의 서브 클래스, `multipart` 가 아닌 MIME 메시지의 중간 베이스 클래스입니다. 이 클래스의 기본 목적은 `multipart` 메시지에만 적합한 `attach()` 메서드 사용을 방지하는 것입니다. `attach()`가 호출되면 `MultipartConversionError` 예외가 발생합니다.

class email.mime.multipart.**MIMEMultipart**(*_subtype='mixed', boundary=None, _subparts=None, *, policy=compat32, **_params*)

모듈: email.mime.multipart

MIMEBase의 서브 클래스, multipart 인 MIME 메시지의 중간 베이스 클래스입니다. 선택적 *_subtype*의 기본값은 mixed이지만, 메시지의 부 유형을 지정하는 데 사용할 수 있습니다. multipart/_subtype의 Content-Type 헤더가 메시지 객체에 추가됩니다. MIME-Version 헤더도 추가됩니다.

선택적 *boundary*는 멀티 파트 경계 문자열입니다. None(기본값)이면, 경계는 필요할 때 (예를 들어 메시지가 직렬화될 때) 계산됩니다.

*_subparts*는 페이 로드의 초기 서브 파트 시퀀스입니다. 이 시퀀스를 리스트로 변환할 수 있어야 합니다. Message.attach 메서드를 사용하여 항상 메시지에 새 서브 파트를 첨부할 수 있습니다.

선택적 *policy* 인자의 기본값은 compat32입니다.

Content-Type 헤더에 대한 추가 매개 변수는 키워드 인자에서 취하거나, 키워드 딕셔너리인 *_params* 인자로 전달됩니다.

버전 3.6에서 변경: *policy* 키워드 전용 매개 변수를 추가했습니다.

class email.mime.application.**MIMEApplication**(*_data, _subtype='octet-stream', _encoder=email.encoders.encode_base64, *, policy=compat32, **_params*)

모듈: email.mime.application

MIMENonMultipart 의 서브 클래스, MIMEApplication 클래스는 주 유형 application의 MIME 메시지 객체를 나타내는 데 사용됩니다. *_data*는 원시 애플리케이션 데이터의 바이트 열을 포함합니다. 선택적 *_subtype*은 MIME 부 유형을 지정하고 기본값은 octet-stream입니다.

선택적 *_encoder*는 전송을 위해 데이터의 실제 인코딩을 수행할 콜러블(즉, 함수)입니다. 이 콜러블은 MIMEApplication 인스턴스인 하나의 인자를 취합니다. 페이 로드를 인코딩된 형식으로 변경하려면 get_payload()와 set_payload()를 사용해야 합니다. 또한, 필요에 따라 Content-Transfer-Encoding이나 기타 헤더를 메시지 객체에 추가해야 합니다. 기본 인코딩은 base64입니다. 내장 인코더의 목록은 email.encoders 모듈을 참조하십시오.

선택적 *policy* 인자의 기본값은 compat32입니다.

*_params*는 베이스 클래스 생성자로 바로 전달됩니다.

버전 3.6에서 변경: *policy* 키워드 전용 매개 변수를 추가했습니다.

class email.mime.audio.**MIMEAudio**(*_audiodata, _subtype=None, _encoder=email.encoders.encode_base64, *, policy=compat32, **_params*)

모듈: email.mime.audio

MIMENonMultipart 의 서브 클래스, MIMEAudio 클래스는 주 유형 audio의 MIME 메시지 객체를 만드는 데 사용됩니다. *_audiodata*는 원시 오디오 데이터의 바이트 열을 포함합니다. 이 데이터를 au, wav, aiff 또는 aifc로 디코딩 할 수 있으면, 부 유형이 Content-Type 헤더에 자동으로 포함됩니다. 그렇지 않으면 *_subtype* 인자를 통해 audio 부 유형을 명시적으로 지정할 수 있습니다. 부 유형을 추측할 수 없고 *_subtype*이 제공되지 않으면, TypeError가 발생합니다.

선택적 *_encoder*는 전송을 위해 오디오 데이터의 실제 인코딩을 수행할 콜러블(즉, 함수)입니다. 이 콜러블은 MIMEAudio 인스턴스인 하나의 인자를 취합니다. 페이 로드를 인코딩된 형식으로 변경하려면 get_payload()와 set_payload()를 사용해야 합니다. 또한 필요에 따라 Content-Transfer-Encoding이나 기타 헤더를 메시지 객체에 추가해야 합니다. 기본 인코딩은 base64입니다. 내장 인코더의 목록은 email.encoders 모듈을 참조하십시오.

선택적 *policy* 인자의 기본값은 compat32입니다.

*_params*는 베이스 클래스 생성자로 바로 전달됩니다.

버전 3.6에서 변경: *policy* 키워드 전용 매개 변수를 추가했습니다.

class email.mime.image.**MIMEImage**(*_imagedata, _subtype=None, _encoder=email.encoders.encode_base64, *, policy=compat32, **_params*)

모듈: email.mime.image

MIMENonMultipart 의 서브 클래스, MIMEImage 클래스는 주 유형 image의 MIME 메시지 객체를 만드는 데 사용됩니다. *_imagedata*는 원시 이미지 데이터의 바이트 열을 포함합니다. 이 데이터 형이 감지된다면 (jpeg, png, gif, tiff, rgb, pbm, pgm, ppm, rast, xbm, bmp, webp 과 exr 을 시도합니다), 부 유형이 Content-Type 헤더에 자동으로 포함됩니다. 그렇지 않으면 *_subtype* 인자를 통해 image 부 유형을 명시적으로 지정할 수 있습니다. 부 유형을 추측할 수 없고 *_subtype*이 제공되지 않으면, TypeError가 발생합니다.

선택적 *_encoder*는 전송을 위해 이미지 데이터의 실제 인코딩을 수행할 콜러블(즉, 함수)입니다. 이 콜러블은 MIMEImage 인스턴스인 하나의 인자를 취합니다. 페이 로드를 인코딩된 형식으로 변경하려면 get_payload()와 set_payload()를 사용해야 합니다. 또한 필요에 따라 Content-Transfer-Encoding이나 기타 헤더를 메시지 객체에 추가해야 합니다. 기본 인코딩은 base64입니다. 내장 인코더의 목록은 email.encoders 모듈을 참조하십시오.

선택적 *policy* 인자의 기본값은 compat32입니다.

*_params*는 MIMEBase 생성자로 바로 전달됩니다.

버전 3.6에서 변경: *policy* 키워드 전용 매개 변수를 추가했습니다.

class email.mime.message.**MIMEMessage**(*_msg, _subtype='rfc822', *, policy=compat32*)

모듈: email.mime.message

MIMENonMultipart 의 서브 클래스, MIMEMessage 클래스는 주 유형 message의 MIME 객체를 만드는 데 사용됩니다. *_msg*는 페이 로드로 사용되며, Message 클래스(또는 그 서브 클래스)의 인스턴스여야 합니다, 그렇지 않으면 TypeError가 발생합니다.

선택적 *_subtype*은 메시지의 부 유형을 설정합니다; 기본값은 rfc822입니다.

선택적 *policy* 인자의 기본값은 compat32입니다.

버전 3.6에서 변경: *policy* 키워드 전용 매개 변수를 추가했습니다.

class email.mime.text.**MIMEText**(*_text, _subtype='plain', _charset=None, *, policy=compat32*)

모듈: email.mime.text

MIMENonMultipart 의 서브 클래스, MIMEText 클래스는 주 유형 text의 MIME 객체를 만드는 데 사용됩니다. *_text*는 페이 로드의 문자열입니다. *_subtype*은 부 유형이며 기본값은 plain입니다. *_charset*은 텍스트의 문자 집합이며 MIMENonMultipart 생성자에 인자로 전달됩니다; 기본값은 문자열에 ascii 코드 포인트만 포함되어 있으면 us-ascii이고, 그렇지 않으면 utf-8입니다. *_charset* 매개 변수는 문자열이나 Charset 인스턴스를 받아들입니다.

_charset 인자가 명시적으로 None으로 설정되어 있지 않은 한, 만들어진 MIMEText 객체에는 charset 매개 변수가 있는 Content-Type 헤더와 Content-Transfer-Encoding 헤더가 모두 있습니다. 이는 charset이 set_payload 명령으로 전달되더라도, 후속 set_payload 호출이 인코딩된 페이 로드를 만들지 않음을 의미합니다. Content-Transfer-Encoding 헤더를 삭제하여 이 동작을 "재설정" 할 수 있으며, 그 후에 set_payload 호출은 자동으로 새 페이 로드를 인코딩합니다 (그리고 새 Content-Transfer-Encoding 헤더를 추가합니다).

선택적 *policy* 인자의 기본값은 compat32입니다.

버전 3.5에서 변경: *_charset*은 Charset 인스턴스도 받아들입니다.

버전 3.6에서 변경: *policy* 키워드 전용 매개 변수를 추가했습니다.

20.1.11 email.header: 국제화된 헤더

소스 코드: Lib/email/header.py

이 모듈은 레거시 (Compat32) 이메일 API의 일부입니다. 현재 API에서 헤더의 인코딩과 디코딩은 EmailMessage 클래스의 딕셔너리와 유사한 API에 의해 투명하게 처리됩니다. 레거시 코드에서 사용

하는 것 외에도, 이 모듈은 헤더를 인코딩할 때 사용되는 문자 집합을 완전히 제어해야 하는 응용 프로그램에서 유용할 수 있습니다.

이 섹션의 나머지 텍스트는 모듈의 원본 설명서입니다.

RFC 2822는 이메일 메시지 형식을 기술하는 기본 표준입니다. 대부분의 이메일이 ASCII 문자로만 구성된 당시에 널리 사용된 이전 **RFC 822** 표준에서 파생됩니다. **RFC 2822**는 이메일에 7비트 ASCII 문자만 포함되어 있다고 가정한 명세입니다.

물론, 이메일이 전 세계에 배포되면서, 국제화되어 언어별 문자 집합을 이메일 메시지에 사용할 수 있게 되었습니다. 기본 표준에서는 여전히 7비트 ASCII 문자만 사용하여 이메일 메시지를 전송해야 하므로, ASCII가 아닌 문자가 포함된 이메일을 **RFC 2822** 호환 형식으로 인코딩하는 방법을 설명하는 많은 RFC가 작성되었습니다. 이러한 RFC에는 **RFC 2045**, **RFC 2046**, **RFC 2047** 및 **RFC 2231**이 포함됩니다. `email` 패키지는 `email.header`와 `email.charset` 모듈에서 이러한 표준을 지원합니다.

이메일 헤더에 ASCII가 아닌 문자를 포함 시키려면 (가령 `Subject`나 `To` 필드에), `Header` 클래스를 사용하고 헤더 값에 문자열을 사용하는 대신 `Message` 객체의 필드를 `Header` 인스턴스로 대입해야 합니다. `email.header` 모듈에서 `Header` 클래스를 임포트 합니다. 예를 들면:

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'
```

`Subject` 필드에 비 ASCII 문자를 포함하기 위해 어떻게 했는지 아시겠습니까? 우리는 `Header` 인스턴스를 만들고 바이트 문자열이 인코딩된 문자 집합을 전달하여 이를 수행했습니다. 뒤에 `Message` 인스턴스가 평탄화될 때, `Subject` 필드는 올바르게 **RFC 2047** 인코딩되었습니다. MIME 인식 메일 리더는 내장된 ISO-8859-1 문자를 사용하여 이 헤더를 표시하게 됩니다.

`Header` 클래스 설명은 다음과 같습니다:

class email.header.**Header**(*s=None, charset=None, maxlinelen=None, header_name=None, continuation_ws=' ', errors='strict'*)

다른 문자 집합의 문자열을 포함할 수 있는 MIME 호환 헤더를 만듭니다.

선택적 *s*는 초기 헤더 값입니다. `None`(기본값)이면, 초기 헤더 값이 설정되지 않습니다. 나중에 `append()` 메서드 호출로 헤더에 추가할 수 있습니다. *s*는 `bytes`나 `str`의 인스턴스일 수 있지만, 의미에 대해서는 `append()` 설명서를 참조하십시오.

선택적 *charset*은 두 가지 용도로 사용됩니다: `append()` 메서드에 대한 *charset* 인자와 같은 의미입니다. 또한, *charset* 인자를 생략하는 모든 후속 `append()` 호출에 대한 기본 문자 집합을 설정합니다. *charset*이 생성자에 제공되지 않으면 (기본값), `us-ascii` 문자 집합이 *s*의 초기 문자 집합과 후속 `append()` 호출의 기본값으로 사용됩니다.

최대 줄 길이는 *maxlinelen*을 통해 명시적으로 지정할 수 있습니다. (*s*에 포함되지 않은 필드 헤더를 고려하기 위해, 예를 들어 `Subject`) 첫 번째 줄을 더 짧은 값으로 분할하려면 *header_name*에 필드 이름을 전달하십시오. 기본 *maxlinelen*은 78이고, *header_name*의 기본값은 `None`입니다, 이는 긴 분할 헤더의 첫 번째 줄을 고려하지 않음을 의미합니다.

선택적인 *continuation_ws*는 **RFC 2822** 호환 접는 공백(folding whitespace)이어야하며, 일반적으로 스페이스나 하드 탭 문자입니다. 이 문자는 연속 줄 앞에 추가됩니다. *continuation_ws*는 기본적으로 단일 스페이스 문자입니다.

선택적 *errors*는 `append()` 메서드로 바로 전달됩니다.

append(*s, charset=None, errors='strict'*)

문자열 *s*를 MIME 헤더에 추가합니다.

선택적인 *charset*(제공되면)은 `Charset` 인스턴스(`email.charset`을 참조하십시오)나 문자 집합의 이름이어야 하며, 이는 `Charset` 인스턴스로 변환됩니다. `None`(기본값) 값은 생성자에 지정된 *charset*이 사용됨을 의미합니다.

*s*는 `bytes`나 `str`의 인스턴스일 수 있습니다. `bytes`의 인스턴스이면, *charset*은 해당 바이트 문자열의 인코딩이며, 문자열을 해당 문자 집합으로 디코딩할 수 없으면 `UnicodeError`가 발생합니다.

*s*가 `str`의 인스턴스이면, *charset*은 문자열에 있는 문자의 문자 집합을 지정하는 힌트입니다.

두 경우 모두, **RFC 2047** 규칙을 사용하여 **RFC 2822** 호환 헤더를 생성할 때, 문자열은 charset의 출력 코덱을 사용하여 인코딩됩니다. 출력 코덱을 사용하여 문자열을 인코딩할 수 없으면 UnicodeError가 발생합니다.

선택적 *errors*는 *s*가 바이트 문자열일 때 decode 호출에 errors 인자로 전달됩니다.

encode(*splitchars*=';, \t', *maxlinelen*=*None*, *linesep*='\n')

메시지 헤더를 RFC 호환 형식으로 인코딩합니다. 긴 줄을 래핑하고 비 ASCII 부분을 base64나 quoted-printable 인코딩으로 캡슐화할 수 있습니다.

선택적 *splitchars*는 일반 헤더 래핑 중 분할 알고리즘에 의해 추가 가중치를 받아야 하는 문자를 포함하는 문자열입니다. 이것은 **RFC 2822**의 '높은 수준의 구문 분할' 을 아주 거칠게 지원합니다: 분할 문자 뒤에 오는 분리 점이 줄 분할 중에 선호되며, 문자열에 나타나는 순서대로 문자가 선호됩니다. 스페이스와 탭이 문자열에 포함되어 다른 분할 문자가 분할되는 줄에 나타나지 않을 때 분할 지점으로 어느 것을 선호해야 하는지를 나타낼 수 있습니다. Splitchars는 **RFC 2047** 인코딩 된 줄에 영향을 미치지 않습니다.

주어지면, *maxlinelen*은 최대 줄 길이에 대한 인스턴스의 값을 대체합니다.

*linesep*은 접힌 헤더의 줄을 구분하는 데 사용되는 문자를 지정합니다. 기본적으로 파이썬 응용 프로그램 코드에 가장 유용한 값이지만 (`\n`), RFC 호환 줄 구분자로 헤더를 생성하기 위해 `\r\n`을 지정할 수 있습니다.

버전 3.2에서 변경: *linesep* 인자를 추가했습니다.

`Header` 클래스는 표준 연산자와 내장 함수를 지원하기 위한 많은 메서드도 제공합니다.

__str__()

무제한 줄 길이를 사용하여, `Header`의 근삿값을 문자열로 반환합니다. 모든 조각은 지정된 인코딩을 사용하여 유니코드로 변환되고 적절하게 결합합니다. 문자 집합이 `'unknown-8bit'`인 조각은 `'replace'` 에러 처리기를 사용하여 ASCII로 디코딩됩니다.

버전 3.2에서 변경: `'unknown-8bit'` 문자 집합에 대한 처리가 추가되었습니다.

__eq__(*other*)

이 메서드를 사용하면 두 개의 `Header` 인스턴스가 같은지 비교할 수 있습니다.

__ne__(*other*)

이 메서드를 사용하면 두 `Header` 인스턴스가 다른지 비교할 수 있습니다.

`email.header` 모듈은 다음과 같은 편의 함수도 제공합니다.

email.header.**decode_header**(*header*)

문자 집합을 변환하지 않고 메시지 헤더 값을 디코딩합니다. 헤더 값은 *header*에 있습니다.

For historical reasons, this function may return either:

1. A list of pairs containing each of the decoded parts of the header, `(decoded_bytes, charset)`, where *decoded_bytes* is always an instance of `bytes`, and *charset* is either:
   - A lower case string containing the name of the character set specified.
   - `None` for non-encoded parts of the header.
2. A list of length 1 containing a pair `(string, None)`, where *string* is always an instance of `str`.

An `email.errors.HeaderParseError` may be raised when certain decoding errors occur (e.g. a base64 decoding exception).

Here are examples:

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[(b'p\xf6stal', 'iso-8859-1')]
>>> decode_header('unencoded_string')
[('unencoded_string', None)]
>>> decode_header('bar =?utf-8?B?ZsOzbw==?=')
[(b'bar ', None), (b'f\xc3\xb3o', 'utf-8')]
```

> **참고**
>
> This function exists for backwards compatibility only. For new code, we recommend using `email.headerregistry.HeaderRegistry`.

email.header.**make_header**(*decoded_seq, maxlinelen=None, header_name=None, continuation_ws=' '*)

`decode_header()`에 의해 반환된 것과 같은 쌍의 시퀀스로부터 `Header` 인스턴스를 만듭니다.

`decode_header()`는 헤더 값 문자열을 취하고 `(decoded_string, charset)` 형식의 쌍의 시퀀스를 반환합니다. 여기서 *charset*은 문자 집합의 이름입니다.

이 함수는 해당 쌍의 시퀀스 중 하나를 취해서 `Header` 인스턴스를 반환합니다. 선택적 *maxlinelen*, *header_name* 및 *continuation_ws*는 `Header` 생성자에서와 같습니다.

> **참고**
>
> This function exists for backwards compatibility only, and is not recommended for use in new code.

20.1.12 email.charset: 문자 집합 표현

소스 코드: Lib/email/charset.py

이 모듈은 레거시 (`Compat32`) 이메일 API의 일부입니다. 새 API에서는 별칭 표만 사용됩니다.

이 섹션의 나머지 텍스트는 모듈의 원본 설명서입니다.

이 모듈은 문자 집합 레지스트리와 이 레지스트리를 조작하기 위한 몇 가지 편의 메서드뿐만 아니라, 이메일 메시지의 문자 집합과 문자 집합 변환을 나타내는 `Charset` 클래스를 제공합니다. `Charset` 인스턴스는 `email` 패키지 내의 다른 여러 모듈에서 사용됩니다.

`email.charset` 모듈에서 이 클래스를 임포트 하십시오.

class email.charset.**Charset**(*input_charset=DEFAULT_CHARSET*)

문자 집합을 이메일 속성으로 매핑합니다.

이 클래스는 특정 문자 집합에 대해 이메일에 요구되는 요구 사항에 대한 정보를 제공합니다. 또한 해당 코덱을 사용할 수 있으면, 문자 집합 간 변환을 위한 편의 루틴을 제공합니다. 문자 집합이 주어지면, RFC 호환 방식으로 이메일 메시지에서 해당 문자 집합을 사용하는 방법에 대한 정보를 제공하는 데 최선을 다합니다.

이메일 헤더나 본문에 사용될 때 특정 문자 집합은 quoted-printable이나 base64로 인코딩해야 합니다. 특정 문자 집합은 완전히 변환해야 하며, 이메일에서는 허용되지 않습니다.

선택적 *input_charset*은 아래에 설명된 것과 같습니다; 항상 소문자로 강제 변환됩니다. 별칭이 정규화된 후에는 문자 집합 레지스트리에서 조회로 사용되어 문자 집합을 위해 사용할 헤더 인코딩, 본문 인코딩 및 출력 변환 코덱을 찾습니다. 예를 들어, *input_charset*이 `iso-8859-1`이면, 헤더와 본문은 quoted-printable을 사용하여 인코딩되며 출력 변환 코덱은 필요하지 않습니다. *input_charset*이 `euc-jp`면, 헤더는 base64로 인코딩되고, 본문은 인코딩되지 않지만, 출력 텍스트는 `euc-jp` 문자 집합에서 `iso-2022-jp` 문자 집합으로 변환됩니다.

`Charset` 인스턴스에는 다음과 같은 데이터 어트리뷰트가 있습니다:

input_charset

지정된 초기 문자 집합. 일반적인 별칭은 공식 이메일 이름으로 변환됩니다 (예를 들어 `latin_1` 은 `iso-8859-1` 로 변환됩니다). 기본값은 7비트 `us-ascii` 입니다.

header_encoding

문자 집합을 이메일 헤더에서 사용하기 전에 인코딩해야 하면, 이 어트리뷰트는 `charset.QP` (quoted-printable), `charset.BASE64` (base64 인코딩) 또는 QP나 BASE64 인코딩 중 가장 짧은 것을 뜻하는 `charset.SHORTEST` 로 설정됩니다. 그렇지 않으면, `None` 이 됩니다.

body_encoding

header_encoding 과 같지만, 메일 메시지 본문의 인코딩을 기술합니다. 헤더 인코딩과 다를 수 있습니다. *body_encoding* 에는 `charset.SHORTEST` 가 허용되지 않습니다.

output_charset

일부 문자 집합은 이메일 헤더나 본문에 사용하기 전에 변환해야 합니다. *input_charset* 이 그중 하나이면, 이 어트리뷰트에는 출력이 변환될 문자 집합의 이름이 포함됩니다. 그렇지 않으면 `None` 이 됩니다.

input_codec

input_charset 을 유니코드로 변환하는 데 사용되는 파이썬 코덱의 이름. 변환 코덱이 필요하지 않으면, 이 어트리뷰트는 `None` 입니다.

output_codec

유니코드를 *output_charset* 으로 변환하는 데 사용되는 파이썬 코덱의 이름. 변환 코덱이 필요하지 않으면, 이 어트리뷰트의 값은 *input_codec* 과 같습니다.

`Charset` 인스턴스에는 다음과 같은 메서드도 있습니다:

get_body_encoding()

본문 인코딩에 사용된 콘텐츠 전송 인코딩 (content transfer encoding) 을 반환합니다.

사용된 인코딩에 따라 문자열 `quoted-printable` 이나 `base64` 입니다. 또는 함수일 수 있는데, 이때는 인코딩되는 Message 객체를 단일 인자로 함수를 호출해야 합니다. 그러면 함수는 `Content-Transfer-Encoding` 헤더 자체를 적절한 것으로 설정해야 합니다.

body_encoding 이 `QP` 이면 문자열 `quoted-printable` 을 반환하고, *body_encoding* 이 `BASE64` 이면 문자열 `base64` 를 반환하고, 그렇지 않으면 문자열 `7bit` 를 반환합니다.

get_output_charset()

출력 문자 집합을 반환합니다.

`None` 이 아니라면 *output_charset* 어트리뷰트이고, 그렇지 않으면 *input_charset* 입니다.

header_encode(*string*)

문자열 *string* 을 헤더 인코딩합니다.

인코딩 유형 (base64 나 quoted-printable) 은 *header_encoding* 어트리뷰트를 기반으로 합니다.

header_encode_lines(*string*, *maxlengths*)

string 을 먼저 바이트열로 변환하여 헤더 인코딩합니다.

이는 문자열이 인자 *maxlengths* 에 의해 주어진 최대 줄 길이에 맞춰진다는 점을 제외하고는 `header_encode()` 와 유사합니다. *maxlengths* 는 이터레이터여야 합니다: 이 이터레이터에서 반환된 각 요소는 다음 최대 줄 길이를 제공합니다.

body_encode(*string*)

문자열 *string* 을 본문 인코딩합니다.

인코딩 유형 (base64 나 quoted-printable) 은 *body_encoding* 어트리뷰트를 기반으로 합니다.

`Charset` 클래스는 표준 연산과 내장 함수를 지원하는 여러 가지 메서드도 제공합니다.

__str__()

input_charset 을 소문자로 강제 변환된 문자열로 반환합니다. `__repr__()` 은 `__str__()` 의 별칭입니다.

__eq__(*other*)

이 메서드를 사용하면 두 개의 `Charset` 인스턴스가 같은지 비교할 수 있습니다.

__ne__(*other*)

이 메서드를 사용하면 두 `Charset` 인스턴스가 다른지 비교할 수 있습니다.

`email.charset` 모듈은 또한 전역 문자 집합, 별명 및 코덱 레지스트리에 새 항목을 추가하기 위해 다음 함수를 제공합니다:

email.charset.**add_charset**(*charset*, *header_enc=None*, *body_enc=None*, *output_charset=None*)

전역 레지스트리에 문자 속성을 추가합니다.

*charset*은 입력 문자 집합이며, 문자 집합의 규범적 이름이어야 합니다.

선택적 *header_enc*와 *body_enc*는 quoted-printable의 경우 `charset.QP`, base64 인코딩의 경우 `charset.BASE64`, quoted-printable이나 base64 인코딩 중 가장 짧은 것의 경우 `charset.SHORTEST`, 또는 인코딩이 없으면 `None`입니다. `SHORTEST`는 *header_enc*에만 유효합니다. 인코딩이 없으면 기본값은 `None`입니다.

선택적 *output_charset*은 출력에 적용되어야 하는 문자 집합입니다. `Charset.convert()` 메서드가 호출될 때 입력 문자 집합에서 유니코드로, 다시 출력 문자 집합으로 변환이 진행됩니다. 기본값은 입력과 같은 문자 집합으로 출력하는 것입니다.

*input_charset*과 *output_charset*은 모두 모듈의 문자 집합에서 코덱으로의 매핑에 유니코드 코덱 항목이 있어야 합니다; `add_codec()`을 사용하여 모듈이 모르는 코덱을 추가하십시오. 자세한 내용은 `codecs` 모듈 설명서를 참조하십시오.

전역 문자 집합 레지스트리는 모듈 전역 딕셔너리 `CHARSETS`에 유지됩니다.

email.charset.**add_alias**(*alias*, *canonical*)

문자 집합 별칭을 추가합니다. *alias*는 별칭 이름입니다, 예를 들어 `latin-1`. *canonical*은 문자 집합의 규범적 이름입니다, 예를 들어 `iso-8859-1`.

전역 문자 집합 별칭 레지스트리는 모듈 전역 딕셔너리 `ALIASES`에 유지됩니다.

email.charset.**add_codec**(*charset*, *codecname*)

주어진 문자 집합의 문자를 유니코드와 매핑하는 코덱을 추가합니다.

*charset*은 문자 집합의 규범적 이름입니다. *codecname*은 `str`의 `encode()` 메서드의 두 번째 인자에 적합한 파이썬 코덱의 이름입니다.

20.1.13 email.encoders: 인코더

소스 코드: Lib/email/encoders.py

이 모듈은 레거시(`Compat32`) 이메일 API의 일부입니다. 새로운 API에서 기능은 `set_content()` 메서드의 *cte* 매개 변수에 의해 제공됩니다.

이 모듈은 파이썬 3에서 폐지되었습니다. `MIMEText` 클래스는 인스턴스 화 중에 전달된 *_subtype*과 *_charset* 값을 사용하여 콘텐츠 유형과 CTE 헤더를 설정하므로 여기에 제공된 함수를 명시적으로 호출하면 안 됩니다.

이 섹션의 나머지 텍스트는 모듈의 원본 설명서입니다.

`Message` 객체를 처음부터 만들 때, 종종 호환 메일 서버를 통한 전송을 위해 페이 로드를 인코딩해야 합니다. 바이너리 데이터가 포함된 `image/*`와 `text/*` 유형 메시지의 경우 특히 그렇습니다.

`email` 패키지는 `encoders` 모듈에서 편리한 인코더를 제공합니다. 이 인코더는 실제로 `MIMEAudio`와 `MIMEImage` 클래스 생성자가 기본 인코딩을 제공하는 데 사용됩니다. 모든 인코더 함수는 정확히 하나의 인자, 인코딩할 메시지 객체를 취합니다. 일반적으로 페이 로드를 추출하여, 인코딩한 다음, 페이 로드를 새로 인코딩된 값으로 재설정합니다. 또한 `Content-Transfer-Encoding` 헤더를 적절하게 설정합니다.

이러한 함수는 멀티 파트 메시지에는 의미가 없음에 유의하십시오. 대신 개별 서브 파트에 적용해야 하며, 유형이 멀티 파트인 메시지를 전달하면 `TypeError`가 발생합니다.

제공되는 인코딩 함수는 다음과 같습니다:

email.encoders.**encode_quopri**(*msg*)

페이 로드를 인용 quoted-printable 형식으로 인코딩하고 *Content-Transfer-Encoding* 헤더를 quoted-printable로 설정합니다[1]. 이것은 대부분의 페이 로드가 인쇄 가능한 일반 데이터이지만, 인쇄할 수 없는 문자가 몇 개 있을 때 사용하기에 적합한 인코딩입니다.

email.encoders.**encode_base64**(*msg*)

페이 로드를 base64 형식으로 인코딩하고 *Content-Transfer-Encoding* 헤더를 base64로 설정합니다. 이것은 페이 로드가 대부분 인쇄할 수 없는 데이터일 때 사용하기에 좋은 인코딩입니다. quoted-printable보다 더 압축된 형식이기 때문입니다. base64 인코딩의 단점은 텍스트를 사람이 읽을 수 없도록 만든다는 것입니다.

email.encoders.**encode_7or8bit**(*msg*)

이것은 실제로 메시지의 페이 로드를 수정하지는 않지만, 페이 로드 데이터를 기반으로, *Content-Transfer-Encoding* 헤더를 적절하게 7bit나 8bit로 설정합니다.

email.encoders.**encode_noop**(*msg*)

이것은 아무것도 하지 않습니다. 심지어 *Content-Transfer-Encoding* 헤더도 설정하지 않습니다.

20.1.14 email.utils: 기타 유틸리티

소스 코드: Lib/email/utils.py

---

email.utils 모듈에서 제공되는 몇 가지 유용한 유틸리티가 있습니다:

email.utils.**localtime**(*dt=None*)

Return local time as an aware datetime object. If called without arguments, return current time. Otherwise *dt* argument should be a *datetime* instance, and it is converted to the local time zone according to the system time zone database. If *dt* is naive (that is, dt.tzinfo is None), it is assumed to be in local time.

Added in version 3.3.

Deprecated since version 3.12, removed in version 3.14: *isdst* 매개 변수.

email.utils.**make_msgid**(*idstring=None*, *domain=None*)

RFC 2822-준수 *Message-ID* 헤더에 적합한 문자열을 반환합니다. 선택적인 *idstring*이 주어지면, 메시지 id의 고유성을 강화하는 데 사용되는 문자열입니다. 선택적인 *domain*이 주어지면, msgid의 '@' 다음 부분을 제공합니다. 기본값은 로컬 호스트 명입니다. 일반적으로 이 기본값을 재정의할 필요는 없지만, 여러 호스트에 걸쳐 일관된 도메인 이름을 사용하는 분산 시스템을 구성하는 경우와 같이 특정 경우에 유용할 수 있습니다.

버전 3.2에서 변경: *domain* 키워드가 추가되었습니다.

나머지 함수는 레거시 (Compat32) email API의 일부입니다. 이것들이 제공하는 구문 분석과 포매팅은 새 API의 헤더 구문 분석 장치가 자동으로 수행하므로, 새 API에서 이것들을 직접 사용할 필요는 없습니다.

email.utils.**quote**(*str*)

*str*에 있는 역 슬래시를 두 개의 역 슬래시로 대체하고, 큰따옴표는 역 슬래시-큰따옴표로 대체한 새 문자열을 반환합니다.

email.utils.**unquote**(*str*)

*str*의 *unquote* 된 버전인 새 문자열을 반환합니다. *str*가 큰따옴표로 끝나고 시작하면, 큰따옴표가 제거됩니다. 마찬가지로 *str*이 화살괄호(angle brackets)로 끝나고 시작하면, 제거됩니다.

email.utils.**parseaddr**(*address*, *, *strict=True*)

address(*To*나 *Cc*와 같은 주소를 포함하는 필드의 값이어야 합니다)를 *realname*과 *email* 주소 구성 요소로 구문 분석합니다. 구문 분석에 실패하지 않는 한 해당 정보의 튜플을 반환합니다. 실패하면 ('', '')의 2-튜플이 반환됩니다.

[1] *encode_quopri()*로 인코딩하면 데이터의 모든 탭과 공백 문자도 인코딩됨에 유의하십시오.

If *strict* is true, use a strict parser which rejects malformed inputs.

버전 3.13에서 변경: Add *strict* optional parameter and reject malformed inputs by default.

email.utils.**formataddr**(*pair*, *charset='utf-8'*)

*parseaddr()*의 역, (realname, email_address) 형식의 2-튜플을 취해 *To*나 *Cc* 헤더에 적합한 문자열 값을 반환합니다. *pair*의 첫 번째 요소가 거짓이면, 두 번째 요소는 수정되지 않은 채 반환됩니다.

선택적 *charset*은 realname에 비 ASCII 문자가 포함되어있을 때 realname의 **RFC 2047** 인코딩에 사용될 문자 집합입니다. *str*이나 *Charset*의 인스턴스가 될 수 있습니다. 기본값은 utf-8입니다.

버전 3.3에서 변경: *charset* 옵션이 추가되었습니다.

email.utils.**getaddresses**(*fieldvalues*, *, *strict=True*)

이 메서드는 parseaddr()에 의해 반환된 형식의 2-튜플 리스트를 반환합니다. *fieldvalues*는 *Message.get_all*에 의해 반환될 수 있는 헤더 필드 값의 시퀀스입니다.

If *strict* is true, use a strict parser which rejects malformed inputs.

Here' s a simple example that gets all the recipients of a message:

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

버전 3.13에서 변경: Add *strict* optional parameter and reject malformed inputs by default.

email.utils.**parsedate**(*date*)

RFC 2822의 규칙에 따라 날짜를 구문 분석하려고 시도합니다. 그러나, 일부 메일러는 지정된 대로 이 형식을 따르지 않으므로 *parsedate()*는 이러한 경우에 올바르게 추측하려고 합니다. *date*는 **RFC 2822** 날짜를 포함하는 문자열입니다 (가령 "Mon, 20 Nov 1995 19:12:08 -0500"). 날짜 구문 분석에 성공하면, *parsedate()*는 *time.mktime()*에 직접 전달할 수 있는 9-튜플을 반환합니다; 그렇지 않으면, None을 반환합니다. 결과 튜플의 인덱스 6, 7 및 8은 사용할 수 없음에 유의하십시오.

email.utils.**parsedate_tz**(*date*)

*parsedate()*와 같은 기능을 수행하지만, None이나 10-튜플을 반환합니다; 앞의 9개 요소는 *time.mktime()*에 직접 전달할 수 있는 튜플을 구성하고, 열 번째 요소는 UTC(그리니치 표준 시의 공식 용어)로부터의 날짜의 시간대 오프셋입니다[1]. 입력 문자열에 시간대가 없으면, 반환되는 튜플의 마지막 요소는 0입니다, UTC를 나타냅니다. 결과 튜플의 인덱스 6, 7 및 8은 사용할 수 없음에 유의하십시오.

email.utils.**parsedate_to_datetime**(*date*)

*format_datetime()*의 역. *parsedate()*와 같은 기능을 수행하지만, 성공 시에 *datetime*을 반환합니다; 그렇지 않으면, 23보다 큰 시간이나 -24 와 24 시간 사이에 있지 않은 시간대와 같이 *date*가 잘못된 값을 가질 때 ValueError가 발생합니다. 입력 date의 시간대가 -0000이면, datetime은 나이브 datetime이 되고, date가 RFC를 준수하면 UTC로 시간이 표시되지만, date가 온 메시지의 실제 소스 시간대는 표시되지 않습니다. 입력 date에 다른 유효한 시간대 오프셋이 있으면, datetime은 해당 *timezone tzinfo*가 있는 어웨어 datetime이 됩니다.

Added in version 3.3.

email.utils.**mktime_tz**(*tuple*)

*parsedate_tz()*에 의해 반환된 10-튜플을 UTC 타임스탬프(Epoch 이후 초)로 바꿉니다. 튜플의 시간대 항목이 None이면, 지역 시간으로 간주합니다.

[1] 시간대 오프셋의 부호는 같은 시간대에 대한 time.timezone 변수의 부호와 반대임에 유의하십시오; 이 모듈이 **RFC 2822**를 따르지만, 후자의 변수는 POSIX 표준을 따릅니다.

email.utils.**formatdate**(*timeval=None, localtime=False, usegmt=False*)

RFC 2822에 따르는 날짜 문자열을 반환합니다, 예를 들어:

```
Fri, 09 Nov 2001 01:08:47 -0000
```

선택적 *timeval*이 주어지면 `time.gmtime()`과 `time.localtime()`이 받아들이는 부동 소수점 시간 값입니다, 그렇지 않으면 현재 시각이 사용됩니다.

선택적 *localtime*은, `True`일 때, *timeval*을 해석하고, UTC 대신 일광 절약 시간을 적절히 고려하는 지역 시간대에 상대적인 날짜를 반환토록 하는 플래그입니다. 기본값은 UTC가 사용된다는 뜻인 `False`입니다.

선택적 *usegmt*는, `True`일 때, 날짜 문자열의 시간대를 숫자 `-0000` 대신 ASCII 문자열 `GMT`로 출력하도록 하는 플래그입니다. 일부 프로토콜(가령 HTTP)에 필요합니다. 이것은 *localtime*이 `False`일 때만 적용됩니다. 기본값은 `False`입니다.

email.utils.**format_datetime**(*dt, usegmt=False*)

`formatdate`와 같지만, 입력이 `datetime` 인스턴스입니다. 나이브 datetime이면, "소스 시간대에 대한 정보가 없는 UTC"로 간주하며, 관습적으로 `-0000`이 시간대로 사용됩니다. 어웨어 `datetime`이면, 숫자 시간대 오프셋이 사용됩니다. 오프셋이 0인 어웨어 시간대면, *usegmt*를 `True`로 설정해서 `GMT` 문자열을 숫자 시간대 오프셋 대신 사용할 수 있습니다. 이것은 표준을 준수하는 HTTP date 헤더를 생성하는 방법을 제공합니다.

Added in version 3.3.

email.utils.**decode_rfc2231**(*s*)

RFC 2231에 따라 *s* 문자열을 디코드합니다.

email.utils.**encode_rfc2231**(*s, charset=None, language=None*)

RFC 2231에 따라 *s* 문자열을 인코드합니다. 선택적인 *charset*과 *language*가 주어지면, 사용할 문자 집합 이름과 언어 이름입니다. 둘 다 지정되지 않으면, *s*가 그대로 반환됩니다. *charset*이 주어졌지만, *language*가 지정되지 않으면, 문자열은 *language*에 대해 빈 문자열을 사용하여 인코딩됩니다.

email.utils.**collapse_rfc2231_value**(*value, errors='replace', fallback_charset='us-ascii'*)

header 매개 변수가 **RFC 2231**로 인코딩되었을 때, `Message.get_param`은 문자 집합, 언어 및 값이 포함된 3-튜플을 반환할 수 있습니다. `collapse_rfc2231_value()`는 이것을 유니코드 문자열로 변환합니다. 선택적 *errors*는 `str`의 `encode()` 메서드의 *errors* 인자로 전달됩니다; 기본값은 `'replace'`입니다. 선택적 *fallback_charset*은 **RFC 2231** 헤더에 있는 것이 파이썬에 알려지지 않았을 때 사용할 문자 집합을 지정합니다; 기본값은 `'us-ascii'`입니다.

편의상, `collapse_rfc2231_value()`에 전달된 *value*가 튜플이 아니면, 문자열이어야 하고 unquote 되어 반환됩니다.

email.utils.**decode_params**(*params*)

RFC 2231에 따라 매개 변수 리스트를 디코드합니다. *params*는 `(content-type, string-value)` 형식의 요소를 포함하는 2-튜플의 시퀀스입니다.

20.1.15 email.iterators: 이터레이터

소스 코드: Lib/email/iterators.py

메시지 객체 트리를 이터레이트 하는 것은 `Message.walk` 메서드를 사용하면 매우 쉽습니다. `email.iterators` 모듈은 메시지 객체 트리에 대한 유용한 고수준 이터레이션을 제공합니다.

email.iterators.**body_line_iterator**(*msg, decode=False*)

*msg*의 모든 서브 파트에 있는 모든 페이 로드를 이터레이트 하여, 문자열 페이 로드를 한 줄씩 반환합니다. 모든 서브 파트 헤더를 건너뛰고, 파이썬 문자열이 아닌 페이 로드가 있는 서브 파트를 건너뜁니다. 이는 `readline()`을 사용하여 파일에서 메시지의 평평한(flat) 텍스트 표현을 읽는 것과 다소 유사하며, 모든 중간 헤더를 건너뜁니다.

선택적 *decode*는 `Message.get_payload`로 전달됩니다.

email.iterators.**typed_subpart_iterator**(*msg*, *maintype='text'*, *subtype=None*)

*msg*의 모든 서브 파트를 이터레이트 하여, *maintype*과 *subtype*으로 지정된 MIME 유형과 일치하는 서브 파트만 반환합니다.

*subtype*은 선택적임에 유의하십시오; 생략하면, 서브 파트 MIME 형식 일치는 메인 형식으로만 수행됩니다. *maintype*도 선택적입니다; 기본값은 `text`입니다.

따라서, 기본적으로 `typed_subpart_iterator()`는 MIME 유형이 `text/*`인 각 서브 파트를 반환합니다.

유용한 디버깅 도구로 다음 함수가 추가되었습니다. 이것은 패키지에서 지원되는 공용 인터페이스의 일부로 간주하지 않아야 합니다.

email.iterators.**_structure**(*msg*, *fp=None*, *level=0*, *include_default=False*)

메시지 객체 구조의 콘텐츠 유형을 들여쓰기하여 인쇄합니다. 예를 들면:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
    text/plain
    text/plain
    multipart/digest
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
    text/plain
```

선택적 *fp*는 출력을 인쇄할 파일류 객체입니다. 파이썬의 `print()` 함수에 적합해야 합니다. *level*은 내부적으로 사용됩니다. *include_default*가 참이면, 기본 유형도 인쇄합니다.

더 보기

모듈 *smtplib*
SMTP (Simple Mail Transport Protocol) 클라이언트

모듈 *poplib*
POP (Post Office Protocol) 클라이언트

모듈 *imaplib*
IMAP (Internet Message Access Protocol) 클라이언트

모듈 *mailbox*
다양한 표준 형식을 사용하여 디스크에 메시지 모음을 만들고, 읽고, 관리하는 도구.

20.2 `json` — JSON 인코더와 디코더

소스 코드: Lib/json/__init__.py

RFC 7159(RFC 4627을 대체합니다)와 ECMA-404로 규정되는 JSON (JavaScript Object Notation)은 JavaScript 객체 리터럴 문법에서 영감을 얻은 경량 데이터 교환 형식입니다 (JavaScript[1]의 엄격한 부분집합은 아닙니

[1] the errata for RFC 7159에서 언급했듯이, JSON은 문자열에 U+2028(LINE SEPARATOR)과 U+2029(PARAGRAPH SEPARATOR) 문자를 허용하지만, JavaScript(ECMAScript Edition 5.1 기준)는 허용하지 않습니다.

다).

> **참고**
>
> The term "object" in the context of JSON processing in Python can be ambiguous. All values in Python are objects. In JSON, an object refers to any data wrapped in curly braces, similar to a Python dictionary.

> **경고**
>
> Be cautious when parsing JSON data from untrusted sources. A malicious JSON string may cause the decoder to consume considerable CPU and memory resources. Limiting the size of data to be parsed is recommended.

This module exposes an API familiar to users of the standard library *marshal* and *pickle* modules.

기본 파이썬 객체 계층 구조 인코딩:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

간결한 인코딩:

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

예쁜 인쇄:

```
>>> import json
>>> print(json.dumps({'6': 7, '4': 5}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

Customizing JSON object encoding:

```
>>> import json
>>> def custom_json(obj):
...     if isinstance(obj, complex):
...         return {'__complex__': True, 'real': obj.real, 'imag': obj.imag}
...     raise TypeError(f'Cannot serialize object of {type(obj)}')
...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> json.dumps(1 + 2j, default=custom_json)
'{"__complex__": true, "real": 1.0, "imag": 2.0}'
```

JSON 디코딩:

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']
```

Customizing JSON object decoding:

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

JSONEncoder 확장하기:

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # 베이스 클래스의 default 메서드가 TypeError를 발생시키도록 합니다
...         return super().default(obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[2.0', ', 1.0', ']']
```

Using *json* from the shell to validate and pretty-print:

```
$ echo '{"json":"obj"}' | python -m json
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

자세한 설명은 *Command-line interface*를 참조하십시오.

> **참고**
>
> JSON은 YAML 1.2의 부분 집합입니다. 이 모듈의 기본 설정(특히 기본 *separators* 값)으로 생성된 JSON은 YAML 1.0과 1.1의 부분 집합이기도 합니다. 따라서 이 모듈을 YAML 직렬화기로 사용할 수도 있습니다.

> **참고**
>
> 이 모듈의 인코더와 디코더는 기본적으로 입력과 출력 순서를 유지합니다. 하부 컨테이너에 순서가 없을 때만 순서가 손실됩니다.

20.2.1 기본 사용법

`json.dump`(*obj, fp, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw*)

이 파이썬-*to-JSON* 변환표를 사용하여 *obj*를 JSON 형식 스트림으로 *fp*(`.write()`를 지원하는 파일류 객체)로 직렬화합니다.

> **참고**
>
> *`pickle`*과 *`marshal`*과 달리, JSON은 프레임 프로토콜이 아니므로 같은 *fp*를 사용하여 *`dump()`*를 반복 호출하여 여러 객체를 직렬화하려고 하면 잘못된 JSON 파일이 생성됩니다.

매개변수

- **obj** (`object`) – 직렬화될 파이썬 객체.
- **fp** (*file-like object*) – *obj*가 직렬화될 파일류 객체. `json` 모듈은 항상 *`bytes`* 객체가 아니라 *`str`* 객체를 생성합니다, 따라서 `fp.write()`는 *`str`* 입력을 지원해야 합니다.
- **skipkeys** (`bool`) – `True`면, 기본형(*`str`*, *`int`*, *`float`*, *`bool`*, `None`)이 아닌 키는 *`TypeError`*를 발생시키는 대신 건너뜁니다. 기본값 `False`.
- **ensure_ascii** (`bool`) – `True`(기본값)이면, 출력에서 모든 비 ASCII 문자가 이스케이프 되도록 보장됩니다. `False`면, 그 문자들은 있는 그대로 출력됩니다.
- **check_circular** (`bool`) – `False`면, 컨테이너형에 대한 순환 참조 검사를 건너뛰고 순환 참조는 *`RecursionError`*를 일으키게 됩니다 (또는 더 나빠질 수 있습니다). 기본값 `True`.
- **allow_nan** (`bool`) – `False`면, JSON 사양을 엄격히 준수하여, 범위를 벗어난 *`float`* 값(`nan`, `inf`, `-inf`)을 직렬화하면 *`ValueError`*를 일으킵니다. *allow_nan*이 `True`(기본값)면, JavaScript의 대응 물(`NaN`, `Infinity`, `-Infinity`)이 사용됩니다.
- **cls** (a *`JSONEncoder`* subclass) – If set, a custom JSON encoder with the *`default()`* method overridden, for serializing into custom datatypes. If `None` (the default), `JSONEncoder` is used.
- **indent** (*`int`* / *`str`* / *`None`*) – If a positive integer or string, JSON array elements and object members will be pretty-printed with that indent level. A positive integer indents that many spaces per level; a string (such as `"\t"`) is used to indent each level. If zero, negative, or `""` (the empty string), only newlines are inserted. If `None` (the default), the most compact representation is used.
- **separators** (*`tuple`* / *`None`*) – 2-튜플: `(item_separator, key_separator)`. `None`(기본값)이면, *separators*의 기본값은 *indent*가 `None`이면 `(', ', ': ')`이고,

그렇지 않으면 `(',', ': ')` 입니다. 가장 간결한 JSON 을 얻으려면, `(',', ':')` 를 지정하여 공백을 제거해야 합니다.

- **default** (*callable* | None) – 달리 직렬화할 수 없는 객체에 대해 호출되는 함수. 객체의 JSON 인코딩 가능한 버전을 반환하거나 `TypeError`를 발생시켜야 합니다. `None`(기본값) 이면, `TypeError`가 발생합니다.
- **sort_keys** (`bool`) – If `True`, dictionaries will be outputted sorted by key. Default `False`.

버전 3.2에서 변경: *indent*에 정수뿐만 아니라 문자열을 허용합니다.

버전 3.4에서 변경: *indent*가 `None`이 아니면, `(',', ': ')`를 기본값으로 사용합니다.

버전 3.6에서 변경: 모든 선택적 매개 변수는 이제 키워드-전용입니다.

`json.`**`dumps`**(*obj, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw*)

이 변환표를 사용하여 *obj*를 JSON 형식의 `str`로 직렬화합니다. 인자는 `dump()`에서와 같은 의미입니다.

> **참고**
>
> JSON의 키/값 쌍에 있는 키는 항상 `str` 형입니다. 딕셔너리를 JSON으로 변환하면, 딕셔너리의 모든 키가 문자열로 강제 변환됩니다. 이것의 결과로, 딕셔너리를 JSON으로 변환한 다음 다시 딕셔너리로 변환하면, 딕셔너리가 원래의 것과 같지 않을 수 있습니다. 즉, x에 비 문자열 키가 있으면 `loads(dumps(x)) != x`입니다.

`json.`**`load`**(*fp, *, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None, **kw*)

*JSON-to-*파이썬 변환표를 사용하여 *fp*를 파이썬 객체로 역 직렬화합니다.

매개변수

- **fp** (*file-like object*) – 역 직렬화될 JSON 문서를 포함하는 `.read()`를 지원하는 텍스트 파일이나 바이너리 파일.
- **cls** (a `JSONDecoder` subclass) – If set, a custom JSON decoder. Additional keyword arguments to `load()` will be passed to the constructor of *cls*. If `None` (the default), `JSONDecoder` is used.
- **object_hook** (*callable* | None) – If set, a function that is called with the result of any JSON object literal decoded (a `dict`). The return value of this function will be used instead of the `dict`. This feature can be used to implement custom decoders, for example JSON-RPC class hinting. Default `None`.
- **object_pairs_hook** (*callable* | None) – If set, a function that is called with the result of any JSON object literal decoded with an ordered list of pairs. The return value of this function will be used instead of the `dict`. This feature can be used to implement custom decoders. If *object_hook* is also set, *object_pairs_hook* takes priority. Default `None`.
- **parse_float** (*callable* | None) – 설정되면, 디코딩될 모든 JSON float의 문자열로 호출되는 함수. `None`(기본값) 이면, 이것은 `float(num_str)`와 동등합니다. JSON float를 사용자 정의 데이터형으로 구문 분석하는 데 사용될 수 있습니다, 예를 들어 `decimal.Decimal`.
- **parse_int** (*callable* | None) – 설정되면, 디코딩될 모든 JSON int의 문자열로 호출되는 함수. `None`(기본값) 이면, 이것은 `int(num_str)`와 동등합니다. JSON 정수를 사용자 정의 데이터형으로 구문 분석하는 데 사용될 수 있습니다, 예를 들어 `float`.

- **parse_constant** (*callable* | None) – 설정되면, 다음과 같은 문자열 중 하나로 호출되는 함수: `'-Infinity'`, `'Infinity'` 또는 `'NaN'`. 잘못된 JSON 숫자를 만날 때 예외를 발생시키는 데 사용할 수 있습니다. 기본값 `None`.

예외 발생

- ***JSONDecodeError*** – 역 직렬화되는 데이터가 유효한 JSON 문서가 아닐 때.
- ***UnicodeDecodeError*** – When the data being deserialized does not contain UTF-8, UTF-16 or UTF-32 encoded data.

버전 3.1에서 변경:

- 선택적 *object_pairs_hook* 매개 변수가 추가되었습니다.
- *parse_constant*는 더는 'null', 'true', 'false' 에 대해 호출되지 않습니다.

버전 3.6에서 변경:

- 모든 선택적 매개 변수는 이제 키워드-전용입니다.
- *fp*는 이제 바이너리 파일이 될 수 있습니다. 입력 인코딩은 UTF-8, UTF-16 또는 UTF-32 여야 합니다.

버전 3.11에서 변경: The default *parse_int* of `int()` now limits the maximum length of the integer string via the interpreter's *integer string conversion length limitation* to help avoid denial of service attacks.

`json.`**`loads`**(*s*, **, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

`load()`와 동일하지만, 파일류 객체를 역 질렬화하는 대신, 이 변환표를 사용하여 *s*(JSON 문서를 포함하는 `str`, `bytes` 또는 `bytearray` 인스턴스)를 파이썬 객체로 역 직렬화합니다.

버전 3.6에서 변경: *s*는 이제 `bytes`나 `bytearray` 형일 수 있습니다. 입력 인코딩은 UTF-8, UTF-16 또는 UTF-32 여야 합니다.

버전 3.9에서 변경: 키워드 인자 *encoding*이 제거되었습니다.

20.2.2 인코더와 디코더

class `json.`**`JSONDecoder`**(**, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

간단한 JSON 디코더.

기본적으로 디코딩할 때 다음과 같은 변환을 수행합니다:

JSON	파이썬
오브젝트(object)	dict
배열(array)	list
문자열(string)	str
숫자 (정수)	int
숫자 (실수)	float
true	True
false	False
null	None

또한, `NaN`, `Infinity` 및 `-Infinity`를 해당 `float` 값으로 이해합니다. 이 값은 JSON 명세에 속하지 않습니다.

*object_hook*은 지정되면, 모든 JSON 오브젝트의 디코딩된 결과로 호출될 선택적 함수이고, 반환 값을 주어진 `dict` 대신 사용합니다. 사용자 정의 역 직렬화를 제공하는 데 사용할 수 있습니다(예를 들어, JSON-RPC 클래스 힌팅을 지원하기 위해).

*object_pairs_hook*은 모든 JSON 오브젝트가 쌍의 순서 있는 목록으로 디코딩된 결과로 호출될 선택적 함수입니다. `dict` 대신 *object_pairs_hook*의 반환 값이 사용됩니다. 이 기능은 사용자 정의 디코더를

구현하는 데 사용할 수 있습니다. *object_hook*도 정의되어 있으면, *object_pairs_hook*이 우선순위를 갖습니다.

버전 3.1에서 변경: *object_pairs_hook*에 대한 지원이 추가되었습니다.

*parse_float*는 디코딩될 모든 JSON float의 문자열로 호출될 선택적 함수입니다. 기본적으로, 이것은 `float(num_str)`와 동등합니다. JSON float에 대해 다른 데이터형이나 구문 분석기를 사용하고자 할 때 사용될 수 있습니다 (예를 들어, `decimal.Decimal`).

*parse_int*는 디코딩될 모든 JSON int의 문자열로 호출될 선택적 함수입니다. 기본적으로 이것은 `int(num_str)`와 동등합니다. JSON 정수에 대해 다른 데이터형이나 구문 분석기를 사용하고자 할 때 사용될 수 있습니다 (예를 들어 `float`).

*parse_constant*는 다음과 같은 문자열 중 하나로 호출될 선택적 함수입니다: `'-Infinity'`, `'Infinity'`, `'NaN'`. 잘못된 JSON 숫자를 만날 때 예외를 발생시키는 데 사용할 수 있습니다.

*strict*가 거짓이면 (`True`가 기본값입니다), 문자열 안에 제어 문자가 허용됩니다. 이 문맥에서 제어 문자는 0–31 범위의 문자 코드를 가진 것들인데, `'\t'` (탭), `'\n'`, `'\r'` 및 `'\0'`을 포함합니다.

역 직렬화되는 데이터가 유효한 JSON 문서가 아니면, `JSONDecodeError`가 발생합니다.

버전 3.6에서 변경: 모든 매개 변수가 이제 키워드-전용입니다.

decode(*s*)

s(JSON 문서가 포함된 `str` 인스턴스)의 파이썬 표현을 반환합니다.

주어진 JSON 문서가 유효하지 않으면 `JSONDecodeError`가 발생합니다.

raw_decode(*s*)

s(JSON 문서로 시작하는 `str`)에서 JSON 문서를 디코딩하고, 파이썬 표현과 문서가 끝난 *s*에서의 인덱스로 구성된 2-튜플을 반환합니다.

끝에 여분의 데이터가 있을 수 있는 문자열에서 JSON 문서를 디코딩하는 데 사용할 수 있습니다.

class `json.`**JSONEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

파이썬 데이터 구조를 위한 확장 가능한 JSON 인코더

기본적으로 다음 객체와 형을 지원합니다.:

파이썬	JSON
dict	오브젝트(object)
list, tuple	배열(array)
str	문자열(string)
int, float, int와 float에서 파생된 열거형	숫자(number)
True	true
False	false
None	null

버전 3.4에서 변경: int와 float 파생 Enum 클래스에 대한 지원이 추가되었습니다.

다른 객체를 인식하도록 확장하려면, 서브 클래스를 만들고, 가능하면 `o`에 대한 직렬화 가능 객체를 반환하고, 그렇지 않으면 (`TypeError`를 발생시키기 위해) 슈퍼 클래스 구현을 호출하는 다른 메서드로 `default()` 메서드를 구현합니다.

*skipkeys*가 거짓(기본값)이면, `str`, `int`, `float`, `bool` 또는 `None`이 아닌 키를 인코딩하려고 시도할 때 `TypeError`가 발생합니다. *skipkeys*가 참이면 이러한 항목은 단순히 건너뜁니다.

*ensure_ascii*가 참(기본값)이면, 출력에서 모든 비 ASCII 문자가 이스케이프 되도록 보장됩니다. *ensure_ascii*가 거짓이면, 그 문자들은 있는 그대로 출력됩니다.

*check_circular*가 참(기본값)이면, 리스트, 딕셔너리 및 사용자 정의 객체는 무한 재귀 (`RecursionError`를 유발할 수 있습니다)를 방지하기 위해 인코딩 중에 순환 참조를 검사합니다. 그렇지 않으면, 그러한 검사가 수행되지 않습니다.

*allow_nan*이 참(기본값)이면, `NaN`, `Infinity` 및 `-Infinity`는 그 자체로 인코딩됩니다. 이 동작은 JSON 사양을 따르지 않지만, 대부분의 JavaScript 기반 인코더 및 디코더와 일치합니다. 그렇지 않으면, 그러한 float를 인코딩하는 것은 `ValueError`가 됩니다.

*sort_keys*가 참(기본값: `False`)이면, 딕셔너리의 출력이 키로 정렬됩니다; JSON 직렬화를 이전과 비교할 수 있도록 해서 회귀 테스트에 유용합니다.

*indent*가 음이 아닌 정수나 문자열이면, JSON 배열 요소와 오브젝트 멤버가 해당 들여쓰기 수준으로 예쁘게 인쇄됩니다. 0, 음수 또는 `""`의 들여쓰기 수준은 줄 넘김만 삽입합니다. `None`(기본값)은 가장 간결한(compact) 표현을 선택합니다. 양의 정수 indent를 사용하면, 수준 당 그만큼의 스페이스로 들여쓰기합니다. *indent*가 문자열이면 (가령 `"\t"`), 각 수준을 들려 쓰는 데 그 문자열을 사용합니다.

버전 3.2에서 변경: *indent*에 정수뿐만 아니라 문자열을 허용합니다.

지정되면, *separators*는 `(item_separator, key_separator)` 튜플이어야 합니다. 기본값은 *indent*가 `None`이면 `(', ', ': ')`이고, 그렇지 않으면 `(',', ': ')`입니다. 가장 간결한 JSON 표현을 얻으려면, `(',', ':')`를 지정하여 공백을 제거해야 합니다.

버전 3.4에서 변경: *indent*가 `None`이 아니면, `(',', ': ')`를 기본값으로 사용합니다.

지정되면, *default*는 달리 직렬화할 수 없는 객체에 대해 호출되는 함수여야 합니다. 객체의 JSON 인코딩 가능한 버전을 반환하거나 `TypeError`를 발생시켜야 합니다. 지정하지 않으면, `TypeError`가 발생합니다.

버전 3.6에서 변경: 모든 매개 변수가 이제 키워드-전용입니다.

default(*o*)

*o*의 직렬화 가능 객체를 반환하거나 (`TypeError`를 발생시키기 위해서) 베이스 구현을 호출하도록 서브 클래스에 이 메서드를 구현하십시오.

예를 들어, 임의의 이터레이터를 지원하려면, 다음과 같이 `default()`를 구현할 수 있습니다:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # 베이스 클래스의 default 메서드가 TypeError를 발생시키도록 합시다
    return super().default(o)
```

encode(*o*)

파이썬 데이터 구조 *o*의 JSON 문자열 표현을 반환합니다. 예를 들면:

```
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*)

주어진 객체 *o*를 인코딩하고, 준비될 때마다 각 문자열 표현을 산출(yield)합니다. 예를 들면:

```
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

20.2.3 예외

exception `json.`**JSONDecodeError**(*msg*, *doc*, *pos*)

다음 추가 어트리뷰트가 있는 `ValueError`의 서브 클래스:

msg

형식 없는 에러 메시지.

doc

구문 분석 중인 JSON 문서.

pos

구문 분석에 실패한 위치의 시작 부분을 나타내는 *doc*의 인덱스.

lineno

*pos*에 해당하는 줄.

colno

*pos*에 해당하는 열.

Added in version 3.5.

20.2.4 표준 준수와 상호 운용성

JSON 형식은 **RFC 7159**와 ECMA-404에 의해 지정됩니다. 이 절에서는 이 모듈의 RFC 준수 수준에 대해 자세히 설명합니다. 단순화를 위해, `JSONEncoder` 및 `JSONDecoder` 서브 클래스와 명시적으로 언급되지 않은 매개 변수는 고려되지 않습니다.

유효한 JavaScript이지만 유효한 JSON이 아닌 확장을 구현함으로써, 이 모듈은 엄격한 방식으로 RFC를 준수하지는 않습니다. 특히:

- 무한대와 NaN 숫자 값이 받아들여지고 출력됩니다;
- 오브젝트 내에서 반복되는 이름이 허용되고, 마지막 이름-값 쌍의 값만 사용됩니다.

RFC가 RFC를 준수하는 구문 분석기가 RFC를 준수하지 않는 입력 텍스트를 받아들이도록 허용하기 때문에, 이 모듈의 역 직렬화기는 기본 설정에서 기술적으로 RFC를 준수합니다.

문자 인코딩

RFC는 UTF-8, UTF-16 또는 UTF-32를 사용하여 JSON을 표현할 것을 요구하고, 최대 상호 운용성을 위해 권장되는 기본값은 UTF-8입니다.

RFC에 의해 요구되는 것은 아니지만 허용되기 때문에, 이 모듈의 직렬화기는 기본적으로 *ensure_ascii=True*를 설정하므로, 결과 문자열에 ASCII 문자만 포함되도록 출력을 이스케이핑 합니다.

ensure_ascii 매개 변수 외에도, 이 모듈은 파이썬 객체와 유니코드 문자열 사이의 변환으로 엄격하게 정의되어 있으므로, 문자 인코딩 문제를 직접 다루지 않습니다.

RFC는 JSON 텍스트의 시작 부분에 바이트 순서 표시(BOM)를 추가하는 것을 금지하고 있으며, 이 모듈의 직렬화기는 BOM을 출력에 추가하지 않습니다. RFC는 JSON 역 직렬화기가 입력에서 초기 BOM을 무시하는 것을 허용하지만 요구하지는 않습니다. 이 모듈의 역 직렬화기는 초기 BOM이 있을 때 `ValueError`를 발생시킵니다.

RFC는 유효한 유니코드 문자에 해당하지 않는 바이트 시퀀스(예를 들어, 쌍을 이루지 않은 UTF-16 대리 코드(unpaired UTF-16 surrogates))가 포함된 JSON 문자열을 명시적으로 금지하지 않지만, 상호 운용성 문제를 일으킬 수 있다고 지적하고 있습니다. 기본적으로, 이 모듈은 이러한 시퀀스의 코드 포인트를 받아들이고 (원래 `str`에 있을 때) 출력합니다.

무한대와 NaN 숫자 값

RFC는 무한대나 NaN 숫자 값의 표현을 허용하지 않습니다. 그런데도, 기본적으로, 이 모듈은 유효한 JSON 숫자 리터럴 값인 것처럼 `Infinity`, `-Infinity` 및 `NaN`을 받아들이고 출력합니다:

```
>>> # 이 호출 중 어느 것도 예외를 발생시키지 않지만, 결과는 유효한 JSON이 아닙니다
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # 역 직렬화 때도 마찬가지입니다
>>> json.loads('-Infinity')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
-inf
>>> json.loads('NaN')
nan
```

직렬화기에서, *allow_nan* 매개 변수를 사용하여 이 동작을 변경할 수 있습니다. 역 직렬화기에서, *parse_constant* 매개 변수를 사용하여 이 동작을 변경할 수 있습니다.

오브젝트 내에서 반복된 이름

RFC는 JSON 오브젝트 내에서 이름이 고유해야 한다고 지정하지만, JSON 오브젝트 내에서 반복되는 이름을 처리하는 방법을 지정하지는 않습니다. 기본적으로, 이 모듈은 예외를 발생시키지 않습니다; 대신, 주어진 이름에 대한 마지막 이름-값 쌍을 제외한 모든 것을 무시합니다:

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

object_pairs_hook 매개 변수는 이 동작을 변경하는 데 사용할 수 있습니다.

오브젝트나 배열이 아닌 최상윗값

폐지된 **RFC 4627**에 의해 지정된 이전 버전의 JSON은 JSON 텍스트의 최상윗값이 JSON 오브젝트나 배열 (파이썬 `dict`나 `list`)이어야 하고, JSON null, 불리언, 숫자 또는 문자열 값이 될 수 없다고 요구합니다. **RFC 7159**는 그 제한을 제거했으며, 이 모듈은 직렬화기와 역 직렬화기에서 이러한 제한을 구현하지 않으며, 그런 적도 없습니다.

이와 관계없이, 최대한의 상호 운용성을 위해, 여러분은 자발적으로 제한을 준수하기를 원할 수 있습니다.

구현 제약 사항

일부 JSON 역 직렬화기 구현은 다음과 같은 것들에 대한 제한을 설정할 수 있습니다:

- 받아들인 JSON 텍스트의 크기
- JSON 오브젝트와 배열의 최대 중첩 수준
- JSON 숫자의 범위와 정밀도
- JSON 문자열의 내용과 최대 길이

이 모듈은 관련 파이썬 데이터형 자체나 파이썬 인터프리터 자체의 한계 외에는 어떤 제한도 가하지 않습니다.

JSON으로 직렬화할 때, 여러분의 JSON을 사용할 응용 프로그램에 있는 이러한 제한 사항에 주의하십시오. 특히, JSON 숫자가 IEEE 754 배정도 숫자로 역 직렬화되는 것이 일반적이고, 그래서 그 표현의 범위와 정밀도 제한이 적용됩니다. 이것은 매우 큰 규모의 파이썬 `int` 값을 직렬화하거나, `decimal.Decimal`과 같은 "색다른" 숫자 형의 인스턴스를 직렬화할 때 특히 중요합니다.

20.2.5 Command-line interface

소스 코드: Lib/json/tool.py

The `json` module can be invoked as a script via `python -m json` to validate and pretty-print JSON objects. The `json.tool` submodule implements this interface.

선택적 `infile`과 `outfile` 인자가 지정되지 않으면, 각각 `sys.stdin`과 `sys.stdout`이 사용됩니다:

```
$ echo '{"json": "obj"}' | python -m json
{
    "json": "obj"
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
}
$ echo '{1.2:3.4}' | python -m json
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

버전 3.5에서 변경: 출력은 이제 입력과 같은 순서입니다. 딕셔너리의 출력을 키에 대해 알파벳 순으로 정렬하려면 *--sort-keys* 옵션을 사용하십시오.

버전 3.14에서 변경: The *json* module may now be directly executed as `python -m json`. For backwards compatibility, invoking the CLI as `python -m json.tool` remains supported.

Command-line options

infile

유효성을 검사하거나 예쁘게 인쇄할 JSON 파일:

```
$ python -m json mp_films.json
[
    {
        "title": "And Now for Something Completely Different",
        "year": 1971
    },
    {
        "title": "Monty Python and the Holy Grail",
        "year": 1975
    }
]
```

*infile*이 지정되지 않으면, *sys.stdin*에서 읽습니다.

outfile

*infile*의 출력을 지정된 *outfile*에 씁니다. 그렇지 않으면, *sys.stdout*에 씁니다.

--sort-keys

딕셔너리의 출력을 키에 대해 알파벳 순으로 정렬합니다.

Added in version 3.5.

--no-ensure-ascii

비 ASCII 문자의 이스케이프를 비활성화합니다. 자세한 내용은 *json.dumps()*를 참조하십시오.

Added in version 3.9.

--json-lines

모든 입력 행을 별도의 JSON 객체로 구문 분석합니다.

Added in version 3.8.

--indent, --tab, --no-indent, --compact

공백 제어를 위한 상호 배타적 옵션.

Added in version 3.9.

-h, --help

도움말 메시지를 표시합니다.

20.3 mailbox — 다양한 형식의 사서함 조작하기

소스 코드: Lib/mailbox.py

이 모듈은 디스크 상의 사서함과 여기에 포함된 메시지에 액세스하고 조작하기 위한 *Mailbox*와 *Message*라는 두 개의 클래스를 정의합니다. Mailbox는 키에서 메시지로의 딕셔너리와 유사한 매핑을 제공합니다. Message는 형식별 상태와 동작으로 *email.message* 모듈의 *Message* 클래스를 확장합니다. 지원되는 사서함 형식은 Maildir, mbox, MH, Babyl 및 MMDF입니다.

더 보기

모듈 *email*
메시지를 표현하고 조작합니다.

20.3.1 Mailbox 객체

class mailbox.**Mailbox**

검사하고 수정할 수 있는 사서함.

Mailbox 클래스는 인터페이스를 정의하고 인스턴스 화하려는 것은 아닙니다. 대신 형식별 서브 클래스는 Mailbox를 상속해야 하며 코드는 특정 서브 클래스를 인스턴스 화해야 합니다.

Mailbox 인터페이스는 메시지에 해당하는 작은 키가 있는 딕셔너리와 유사합니다. 키는 사용되는 Mailbox 인스턴스에 의해 발급되며 해당 Mailbox 인스턴스에만 의미가 있습니다. 키는 메시지를 다른 메시지로 바꾸는 등 해당 메시지가 수정되어도 메시지를 계속 식별합니다.

집합과 유사한 메서드 *add()*를 사용하여 Mailbox 인스턴스에 메시지를 추가하고 del 문이나 집합과 유사한 메서드 *remove()*와 *discard()*를 사용하여 제거할 수 있습니다.

Mailbox 인터페이스의 의미는 몇 가지 주목할만한 면에서 딕셔너리와 다릅니다. 메시지가 요청될 때마다, 새 표현(보통 *Message* 인스턴스)이 사서함의 현재 상태를 기반으로 생성됩니다. 마찬가지로, Mailbox 인스턴스에 메시지가 추가되면, 제공된 메시지 표현의 내용이 복사됩니다. 두 경우 모두 메시지 표현의 참조는 Mailbox 인스턴스가 유지하는 것이 아닙니다.

기본 Mailbox 이터레이터는 기본 딕셔너리 이터레이터가 수행하는 것처럼 키가 아니라 메시지 표현을 이터레이트 합니다. 또한, 이터레이션 중 사서함 수정은 안전하고 잘 정의되어 있습니다. 이터레이터가 만들어진 후 사서함에 추가된 메시지는 이터레이터가 볼 수 없습니다. 이터레이터가 산출하기 전에 사서함에서 제거된 메시지는 자동으로 건너뛰지만, 이터레이터의 키를 사용하면 해당 메시지가 직후에 제거되는 경우 *KeyError* 예외가 발생할 수 있습니다.

경고

다른 프로세스에 의해 동시에 변경될 수 있는 사서함을 수정할 때는 매우 주의하십시오. 이러한 작업에 사용할 가장 안전한 사서함 형식은 *Maildir*입니다; 동시 쓰기를 위해서는 *mbox*와 같은 단일 파일 형식을 사용하지 마십시오. 사서함을 수정하는 경우, 파일의 메시지를 읽거나 메시지를 추가하거나 삭제하여 변경하기 전에 *lock()*과 *unlock()* 메서드를 호출하여 사서함을 반드시 잠가야 합니다. 사서함을 잠그는 데 실패하면 메시지가 손실되거나 전체 사서함이 손상될 위험이 있습니다.

Mailbox 인스턴스에는 다음 메서드가 있습니다:

add(*message*)

사서함에 *message*를 추가하고 할당된 키를 반환합니다.

매개 변수 *message*는 *Message* 인스턴스, *email.message.Message* 인스턴스, 문자열, 바이트 문자열 또는 파일류 객체(바이너리 모드로 열어야 합니다)일 수 있습니다. *message*가 적절한 형식별 *Message* 서브 클래스의 인스턴스이면 (예를 들어, 이것이 *mbox* 인스턴스일 때 *mboxMessage* 인스턴스이면), 형식별 정보가 사용됩니다. 그렇지 않으면, 형식별 정보에 대한 적절한 기본값이 사용됩니다.

버전 3.2에서 변경: 바이너리 입력에 대한 지원이 추가되었습니다.

remove(*key*)

__delitem__(*key*)

discard(*key*)

사서함에서 *key*에 해당하는 메시지를 삭제합니다.

그러한 메시지가 없으면, 메서드가 `remove()`나 `__delitem__()`으로 호출되면 `KeyError` 예외가 발생하지만, `discard()`로 호출되면 예외가 발생하지 않습니다. 하부 사서함 형식이 다른 프로세스에 의한 동시 수정을 지원하면 `discard()`의 동작을 선호할 수 있습니다.

__setitem__(*key*, *message*)

*key*에 해당하는 메시지를 *message*로 바꿉니다. *key*에 해당하는 메시지가 없으면 `KeyError` 예외를 발생시킵니다.

`add()`와 마찬가지로, *message* 매개 변수는 `Message` 인스턴스, `email.message.Message` 인스턴스, 문자열, 바이트 문자열 또는 파일류 객체(바이너리 모드로 열어야 합니다)일 수 있습니다. *message*가 적절한 형식별 `Message` 서브 클래스의 인스턴스이면 (예를 들어, 이것이 `mbox` 인스턴스일 때 `mboxMessage` 인스턴스이면), 형식별 정보가 사용됩니다. 그렇지 않으면, 현재 *key*에 해당하는 메시지의 형식별 정보가 변경되지 않습니다.

iterkeys()

Return an *iterator* over all keys

keys()

The same as `iterkeys()`, except that a `list` is returned rather than an *iterator*

itervalues()

__iter__()

모든 메시지의 표현에 대한 이터레이터를 반환합니다. `Mailbox` 인스턴스가 초기화될 때 사용자 정의 메시지 팩토리가 지정되지 않는 한, 메시지는 적절한 형식별 `Message` 서브 클래스의 인스턴스로 표현됩니다.

> **참고**
>
> `__iter__()`의 동작은 키를 이터레이트 하는 딕셔너리의 동작과 다릅니다.

values()

The same as `itervalues()`, except that a `list` is returned rather than an *iterator*

iteritems()

(*key*, *message*) 쌍에 대한 이터레이터를 반환합니다. 여기서 *key*는 키이고 *message*는 메시지 표현입니다. `Mailbox` 인스턴스가 초기화될 때 사용자 정의 메시지 팩토리가 지정되지 않는 한 메시지는 적절한 형식별 `Message` 서브 클래스의 인스턴스로 표시됩니다.

items()

The same as `iteritems()`, except that a `list` of pairs is returned rather than an *iterator* of pairs.

get(*key*, *default=None*)

__getitem__(*key*)

*key*에 해당하는 메시지 표현을 반환합니다. 이러한 메시지가 없으면, 메서드가 `get()`으로 호출되면 *default*가 반환되고 메서드가 `__getitem__()`으로 호출되면 `KeyError` 예외가 발생합니다. `Mailbox` 인스턴스가 초기화될 때 사용자 정의 메시지 팩토리가 지정되지 않는 한 메시지는 적절한 형식별 `Message` 서브 클래스의 인스턴스로 표시됩니다.

get_message(*key*)

*key*에 해당하는 메시지의 표현을 적절한 형식별 `Message` 서브 클래스의 인스턴스로 반환하거나, 그러한 메시지가 없으면 `KeyError` 예외를 발생시킵니다.

get_bytes(*key*)

*key*에 해당하는 메시지의 바이트 표현을 반환하거나, 그러한 메시지가 없으면 `KeyError` 예외를 발생시킵니다.

Added in version 3.2.

get_string(*key*)

*key*에 해당하는 메시지의 문자열 표현을 반환하거나 그러한 메시지가 없으면 `KeyError` 예외를 발생시킵니다. 메시지는 `email.message.Message`를 통해 처리되어 7비트(7bit clean) 표현으로 변환됩니다.

get_file(*key*)

*key*에 해당하는 메시지의 파일류 표현을 반환하거나, 그러한 메시지가 없으면 `KeyError` 예외를 발생시킵니다. 파일류 객체는 바이너리 모드로 열린 것처럼 작동합니다. 이 파일이 더는 필요하지 않으면 닫아야 합니다.

버전 3.2에서 변경: 파일 객체는 실제로 바이너리 파일입니다; 이전에는 텍스트 모드로 잘못 반환되었습니다. 또한, 파일류 객체는 이제 컨텍스트 관리자 프로토콜을 지원합니다: `with` 문을 사용하여 자동으로 닫을 수 있습니다.

> **참고**
>
> 메시지의 다른 표현과 달리, 파일류 표현은 메시지를 만든 `Mailbox` 인스턴스나 하부 사서함과 반드시 독립적인 것은 아닙니다. 더욱 자세한 설명서는 각 서브 클래스에서 제공합니다.

__contains__(*key*)

*key*가 메시지에 해당하면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

__len__()

사서함에 있는 메시지 수를 반환합니다.

clear()

사서함에서 모든 메시지를 삭제합니다.

pop(*key*, *default=None*)

*key*에 해당하는 메시지 표현을 반환하고 메시지를 삭제합니다. 이러한 메시지가 없으면, *default*를 반환합니다. `Mailbox` 인스턴스가 초기화될 때 사용자 정의 메시지 팩토리가 지정되지 않는 한 메시지는 적절한 형식별 `Message` 서브 클래스의 인스턴스로 표시됩니다.

popitem()

임의의 (*key*, *message*) 쌍을 반환하고, 여기서 *key*는 키이고 *message*는 메시지 표현입니다, 해당 메시지를 삭제합니다. 사서함이 비어 있으면, `KeyError` 예외를 발생시킵니다. `Mailbox` 인스턴스가 초기화될 때 사용자 정의 메시지 팩토리가 지정되지 않는 한 메시지는 적절한 형식별 `Message` 서브 클래스의 인스턴스로 표시됩니다.

update(*arg*)

매개 변수 *arg*는 *key*에서 *message*로의 매핑이거나 (*key*, *message*) 쌍의 이터러블 이어야 합니다. 주어진 *key*와 *message*에 대해 `__setitem__()`을 사용하는 것처럼 *key*에 해당하는 메시지가 *message*로 설정되도록 사서함을 갱신합니다. `__setitem__()`과 마찬가지로, 각 *key*는 이미 사서함의 메시지에 해당해야 합니다, 그렇지 않으면 `KeyError` 예외가 발생하므로, 일반적으로 *arg*가 `Mailbox` 인스턴스가 되는 것은 올바르지 않습니다.

> **참고**
>
> 딕셔너리와 달리 키워드 인자는 지원되지 않습니다.

flush()

계류 중인 변경 사항을 파일 시스템에 기록합니다. 일부 `Mailbox` 서브 클래스의 경우, 변경 사항이 항상 즉시 기록되고 `flush()`는 아무 작업도 하지 않지만, 이 메서드를 호출하는 습관을 만들어야 합니다.

lock()

다른 프로세스가 수정하지 않아야 한다는 것을 알 수 있도록 사서함에 대한 배타적 권고 잠금(exclusive advisory lock)을 획득합니다. 잠금을 사용할 수 없으면 `ExternalClashError`가 발생

합니다. 사용되는 특정 잠금 메커니즘은 사서함 형식에 따라 다릅니다. 내용을 수정하기 전에 사서함을 항상 잠가야 합니다.

unlock()

사서함의 잠금을 해제합니다 (있다면).

close()

사서함을 플러시하고, 필요하면 잠금을 해제한 다음, 열려있는 모든 파일을 닫습니다. 일부 `Mailbox` 서브 클래스의 경우, 이 메서드는 아무 작업도 수행하지 않습니다.

`Maildir` 객체

class mailbox.**Maildir**(*dirname*, *factory=None*, *create=True*)

Maildir 형식의 사서함에 대한 `Mailbox`의 서브 클래스. 매개 변수 *factory*는 파일류 메시지 표현(바이너리 모드에서 열린 것처럼 동작합니다)을 받아들이고 사용자 정의 표현을 반환하는 콜러블 객체입니다. *factory*가 `None`이면, `MaildirMessage` 가 기본 메시지 표현으로 사용됩니다. *create*가 `True`이면, 사서함이 없으면 만들어집니다.

*create*가 `True`이고 *dirname* 경로가 존재하면, 디렉터리 레이아웃을 확인하지 않고 기존 maildir로 처리됩니다.

역사적인 이유로 *path*가 아니라 *dirname*이라고 이름 붙였습니다.

Maildir은 qmail 메일 전송 에이전트를 위해 고안된 디렉터리 기반 사서함 형식이며 현재 다른 프로그램에서 널리 지원됩니다. Maildir 사서함의 메시지는 공통 디렉터리 구조 내에서 별도의 파일에 저장됩니다. 이 설계를 사용하면 데이터 손상 없이 여러 관련 없는 프로그램에서 Maildir 사서함에 액세스하고 수정할 수 있어서 파일 잠금이 필요하지 않습니다.

Maildir 사서함에는 세 개의 하위 디렉터리가 있습니다, 이름하여 `tmp`, `new` 및 `cur`. 메시지는 일시적으로 `tmp` 하위 디렉터리에 만들어진 다음 `new` 하위 디렉터리로 이동하여 전달을 완료합니다. 메일 사용자 에이전트는 이후에 메시지를 `cur` 하위 디렉터리로 이동하고 파일 이름에 추가된 특수 "info" 섹션에 메시지 상태에 대한 정보를 저장할 수 있습니다.

Courier 메일 전송 에이전트가 도입한 스타일의 폴더도 지원됩니다. `'.'`가 이름의 첫 번째 문자인 경우 주 사서함의 모든 하위 디렉터리는 폴더로 간주합니다. 폴더 이름은 앞의 `'.'`없이 `Maildir`로 표시됩니다. 각 폴더는 그 자체가 Maildir 사서함이지만 다른 폴더를 포함해서는 안 됩니다. 대신, 수준을 구분하기 위해 `'.'`을 사용하여 논리적 중첩이 표시됩니다, 예를 들어, "Archived.2005.07".

colon

Maildir 명세에서는 특정 메시지 파일 이름에 콜론(`':'`)을 사용하도록 요구합니다. 그러나 일부 운영 체제에서는 파일 이름에 이 문자를 허용하지 않습니다. 이러한 운영 체제에서 Maildir과 유사한 형식을 사용하려면, 대신 사용할 다른 문자를 지정해야 합니다. 느낌표(`'!'`)는 인기 있는 선택입니다. 예를 들면:

```
import mailbox
mailbox.Maildir.colon = '!'
```

`colon` 어트리뷰트는 인스턴스별로 설정할 수도 있습니다.

버전 3.13에서 변경: `Maildir` now ignores files with a leading dot.

`Maildir` 인스턴스에는 `Mailbox`의 모든 메서드 외에도 다름과 같은 메서드가 있습니다:

list_folders()

모든 폴더의 이름 리스트를 반환합니다.

get_folder(*folder*)

이름이 *folder*인 폴더를 나타내는 `Maildir` 인스턴스를 반환합니다. 폴더가 없으면 `NoSuchMailboxError` 예외가 발생합니다.

add_folder(*folder*)

이름이 *folder*인 폴더를 만들고 이를 나타내는 `Maildir` 인스턴스를 반환합니다.

remove_folder(*folder*)

이름이 *folder*인 폴더를 삭제합니다. 폴더에 메시지가 포함되었으면, `NotEmptyError` 예외가 발생하고 폴더가 삭제되지 않습니다.

clean()

지난 36시간 동안 액세스하지 않은 사서함에서 임시 파일을 삭제합니다. Maildir 명세는 메일을 읽는 프로그램이 이 작업을 가끔 수행해야 한다고 말합니다.

get_flags(*key*)

Return as a string the flags that are set on the message corresponding to *key*. This is the same as `get_message(key).get_flags()` but much faster, because it does not open the message file. Use this method when iterating over the keys to determine which messages are interesting to get.

If you do have a `MaildirMessage` object, use its `get_flags()` method instead, because changes made by the message' s `set_flags()`, `add_flag()` and `remove_flag()` methods are not reflected here until the mailbox' s `__setitem__()` method is called.

Added in version 3.13.

set_flags(*key*, *flags*)

On the message corresponding to *key*, set the flags specified by *flags* and unset all others. Calling `some_mailbox.set_flags(key, flags)` is similar to

```
one_message = some_mailbox.get_message(key)
one_message.set_flags(flags)
some_mailbox[key] = one_message
```

but faster, because it does not open the message file.

If you do have a `MaildirMessage` object, use its `set_flags()` method instead, because changes made with this mailbox method will not be visible to the message object' s method, `get_flags()`.

Added in version 3.13.

add_flag(*key*, *flag*)

*key*에 해당하는 메시지에 대해, 다른 플래그를 변경하지 않고 *flag*로 지정된 플래그를 설정합니다. 한 번에 둘 이상의 플래그를 추가하기 위해, *flag*가 둘 이상의 문자로 구성된 문자열일 수 있습니다.

Considerations for using this method versus the message object' s `add_flag()` method are similar to those for `set_flags()`; see the discussion there.

Added in version 3.13.

remove_flag(*key*, *flag*)

*key*에 해당하는 메시지에 대해, 다른 플래그를 변경하지 않고 *flag*에 의해 지정된 플래그를 설정해제합니다. 한 번에 둘 이상의 플래그를 제거하기 위해, *flag*는 둘 이상의 문자로 구성된 문자열일 수 있습니다.

Considerations for using this method versus the message object' s `remove_flag()` method are similar to those for `set_flags()`; see the discussion there.

Added in version 3.13.

get_info(*key*)

Return a string containing the info for the message corresponding to *key*. This is the same as `get_message(key).get_info()` but much faster, because it does not open the message file. Use this method when iterating over the keys to determine which messages are interesting to get.

If you do have a `MaildirMessage` object, use its `get_info()` method instead, because changes made by the message' s `set_info()` method are not reflected here until the mailbox' s `__setitem__()` method is called.

Added in version 3.13.

set_info(*key*, *info*)

Set the info of the message corresponding to *key* to *info*. Calling `some_mailbox.set_info(key, flags)` is similar to

```
one_message = some_mailbox.get_message(key)
one_message.set_info(info)
some_mailbox[key] = one_message
```

but faster, because it does not open the message file.

If you do have a `MaildirMessage` object, use its `set_info()` method instead, because changes made with this mailbox method will not be visible to the message object' s method, `get_info()`.

Added in version 3.13.

`Maildir`에 의해 구현된 일부 `Mailbox` 메서드는 특별한 주의가 필요합니다:

add(*message*)

__setitem__(*key*, *message*)

update(*arg*)

> **경고**
>
> 이 메서드들은 현재 프로세스 ID를 기반으로 고유한 파일 이름을 생성합니다. 다중 스레드를 사용할 때, 이러한 메서드를 사용하여 같은 사서함을 동시에 조작하지 않도록 스레드를 조정하지 않으면 감지되지 않은 이름 충돌이 발생하여 사서함이 손상될 수 있습니다.

flush()

Maildir 사서함에 대한 모든 변경 사항은 즉시 적용되므로, 이 메서드는 아무 작업도 수행하지 않습니다.

lock()

unlock()

Maildir 사서함은 잠금을 지원(또는 필요로)하지 않아서, 이 메서드들은 아무 작업도 수행하지 않습니다.

close()

`Maildir` 인스턴스는 아무런 열린 파일도 유지하지 않으며 하부 사서함은 잠금을 지원하지 않아서, 이 메서드는 아무 작업도 수행하지 않습니다.

get_file(*key*)

호스트 플랫폼에 따라, 반환된 파일이 열려있는 동안 하부 메시지를 수정하거나 제거하지 못할 수 있습니다.

> **더 보기**
>
> **Courier의 maildir 매뉴얼 페이지**
> 형식의 명세. 지원 폴더에 대한 공통 확장을 설명합니다.
>
> **Using maildir format**
> 발명가의 Maildir에 대한 노트. 개정된 이름 생성 체계와 "info" 의미론에 대한 세부 정보를 포함합니다.

mbox 객체

class mailbox.**mbox**(*path*, *factory=None*, *create=True*)

mbox 형식의 사서함에 대한 `Mailbox`의 서브 클래스. 매개 변수 *factory*는 파일류 메시지 표현(바이너리 모드에서 열린 것처럼 동작합니다)을 받아들이고 사용자 정의 표현을 반환하는 콜러블 객체입니다. *factory*가 `None`이면, `mboxMessage`가 기본 메시지 표현으로 사용됩니다. *create*가 `True`이면, 사서함이 없으면 만들어집니다.

mbox 형식은 유닉스 시스템에서 메일을 저장하기 위한 고전적인 형식입니다. mbox 사서함의 모든 메시지는 처음 5개의 문자가 "From "인 줄로 표시되는 각 메시지의 시작 부분과 함께 단일 파일에 저장됩니다.

원래 형식에서 인식된 단점을 해결하기 위해 mbox 형식의 여러 변형이 존재합니다. 호환성을 위해, `mbox`는 *mboxo*라고도 하는 원래 형식을 구현합니다. 즉, `Content-Length` 헤더(있다면)는 무시되고 메시지 본문의 줄 시작 부분에 있는 "From "은 메시지를 저장할 때 ">From "으로 변환됩니다, 하지만 메시지를 읽을 때 ">From "은 "From "으로 변환되지 않습니다.

`mbox`에 의해 구현된 일부 `Mailbox` 메서드는 특별한 주의가 필요합니다:

get_bytes(*key*, *from_=False*)

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix "From " line. If *from_* is False, the first line of the file is dropped.

get_file(*key*, *from_=False*)

`mbox` 인스턴스에서 `flush()`나 `close()`를 호출한 후 파일을 사용하면 예기치 않은 결과가 발생하거나 예외가 발생할 수 있습니다.

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix "From " line. If *from_* is False, the first line of the file is dropped.

get_string(*key*, *from_=False*)

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix "From " line. If *from_* is False, the first line of the file is dropped.

lock()

unlock()

세 가지 잠금 메커니즘이 사용됩니다—점 잠금(dot locking)과, 사용할 수 있다면, `flock()`과 `lockf()` 시스템 호출.

더 보기

tin의 mbox 매뉴얼 페이지
형식의 명세, 잠금에 대한 세부 정보가 있습니다.

유닉스에서 Netscape Mail 구성하기: 왜 Content-Length 형식이 나쁜가
변형이 아닌 원래 mbox 형식을 사용하는 것에 대한 논의.

"mbox"는 서로 호환되지 않는 여러 사서함 형식의 집합입니다
mbox 변형의 역사.

MH 객체

class mailbox.**MH**(*path*, *factory=None*, *create=True*)

MH 형식의 사서함에 대한 `Mailbox`의 서브 클래스. 매개 변수 *factory*는 파일류 메시지 표현(바이너리 모드에서 열린 것처럼 동작합니다)을 받아들이고 사용자 정의 표현을 반환하는 콜러블 객체입니다. *factory*가 `None`이면, `MHMessage`가 기본 메시지 표현으로 사용됩니다. *create*가 `True`이면, 사서함이 없으면 만들어집니다.

MH는 메일 사용자 에이전트인 MH 메시지 처리 시스템(MH Message Handling System)을 위해 개발된 디렉터리 기반 사서함 형식입니다. MH 사서함의 각 메시지는 각자의 파일에 있습니다. MH 사서함에는 메시지 외에 다른 MH 사서함(폴더(*folders*)라고 합니다)이 포함될 수 있습니다. 폴더는 무제한 중첩될 수 있습니다. MH 사서함은 또한 메시지를 서브 폴더로 이동하지 않고 논리적으로 그룹화하는

데 사용되는 명명된 목록인 시퀀스(*sequences*)를 지원합니다. 시퀀스는 각 폴더의 `.mh_sequences` 라는 파일에 정의됩니다.

`MH` 클래스는 MH 사서함을 조작하지만, **mh**의 모든 동작을 모사하려고 시도하지는 않습니다. 특히, **mh**가 상태와 구성을 저장하는 데 사용하는 `context`나 `.mh_profile` 파일을 수정하지 않고 영향받지 않습니다.

`MH` 인스턴스에는 *Mailbox*의 모든 메서드 외에도 다음 메서드가 있습니다:

버전 3.13에서 변경: Supported folders that don' t contain a `.mh_sequences` file.

list_folders()

모든 폴더의 이름 리스트를 반환합니다.

get_folder(*folder*)

이름이 *folder*인 폴더를 나타내는 `MH` 인스턴스를 반환합니다. 폴더가 없으면 *NoSuchMailboxError* 예외가 발생합니다.

add_folder(*folder*)

이름이 *folder*인 폴더를 만들고 이를 나타내는 `MH` 인스턴스를 반환합니다.

remove_folder(*folder*)

이름이 *folder*인 폴더를 삭제합니다. 폴더에 메시지가 포함되었으면, *NotEmptyError* 예외가 발생하고 폴더가 삭제되지 않습니다.

get_sequences()

키 리스트에 매핑된 시퀀스 이름의 딕셔너리를 반환합니다. 시퀀스가 없으면, 빈 딕셔너리가 반환됩니다.

set_sequences(*sequences*)

*get_sequences()*에서 반환하는 것과 같이, 키 리스트에 매핑된 이름의 딕셔너리인, *sequences*를 기반으로 사서함에 존재하는 시퀀스를 다시 정의합니다.

pack()

번호 매기기의 간격을 없애기 위해 필요에 따라 사서함의 메시지 이름을 변경합니다. 시퀀스 리스트의 항목이 그에 따라 갱신됩니다.

> **참고**
>
> 이미 발급된 키는 이 작업에 의해 무효가 되며 이후에 사용해서는 안 됩니다.

`MH`에 의해 구현된 일부 *Mailbox* 메서드는 특별한 주의가 필요합니다:

remove(*key*)

__delitem__(*key*)

discard(*key*)

이 메서드들은 메시지를 즉시 삭제합니다. 이름 앞에 쉼표를 추가하여 메시지를 삭제하도록 표시하는 MH 규칙은 사용되지 않습니다.

lock()

unlock()

세 가지 잠금 메커니즘이 사용됩니다—점 잠금(dot locking)과, 사용할 수 있다면, `flock()`과 `lockf()` 시스템 호출. MH 사서함의 경우, 사서함을 잠그는 것은 `.mh_sequences` 파일을 잠그는 것을 의미하며, 해당 파일에 영향을 주는 작업 기간에만 개별 메시지 파일을 잠급니다.

get_file(*key*)

호스트 플랫폼에 따라, 반환된 파일이 열려있는 동안 하부 메시지를 제거하지 못할 수도 있습니다.

flush()

MH 사서함에 대한 모든 변경 사항이 즉시 적용되므로, 이 메서드는 아무 작업도 수행하지 않습니다.

close()

`MH` 인스턴스는 열린 파일을 유지하지 않아서, 이 메서드는 *`unlock()`*과 동등합니다.

더 보기

nmh - 메시지 처리 시스템
nmh의 홈페이지, 원래 **mh**의 갱신된 버전.

MH & nmh: 사용자와 프로그래머를 위한 이메일
mh와 **nmh**에 대한 GPL 라이선스 책, 사서함 형식에 대한 정보가 있습니다.

`Babyl` 객체

class mailbox.**Babyl**(*path*, *factory=None*, *create=True*)

Babyl 형식의 사서함에 대한 *`Mailbox`*의 서브 클래스. 매개 변수 *factory*는 파일류 메시지 표현(바이너리 모드에서 열린 것처럼 동작합니다)을 받아들이고 사용자 정의 표현을 반환하는 콜러블 객체입니다. *factory*가 `None`이면, *`BabylMessage`*가 기본 메시지 표현으로 사용됩니다. *create*가 `True`이면, 사서함이 없으면 만들어집니다.

Babyl은 Emacs에 포함된 Rmail 메일 사용자 에이전트에서 사용하는 단일 파일 사서함 형식입니다. 메시지의 시작 부분은 Control-Underscore(`'\037'`)와 Control-L(`'\014'`) 두 문자가 포함된 줄로 표시됩니다. 메시지의 끝은 다음 메시지의 시작이나, 마지막 메시지의 경우 Control-Underscore(`'\037'`) 문자가 포함된 줄로 표시됩니다.

Babyl 사서함의 메시지는 두 집합의 헤더가 있습니다, 원래 헤더와 소위 가시적(visible) 헤더가 있습니다. 가시적 헤더는 일반적으로 더 매력적으로 다시 포맷되거나 요약된 원래 헤더의 부분 집합입니다. Babyl 사서함의 각 메시지에는 레이블(*labels*) 리스트, 또는 메시지에 대한 추가 정보를 기록하는 짧은 문자열이 있으며, 사서함에 있는 모든 사용자 정의 레이블 목록은 Babyl 옵션 섹션에 보관됩니다.

`Babyl` 인스턴스에는 *`Mailbox`*의 모든 메서드 외에도 다음과 같은 메서드가 있습니다:

get_labels()

사서함에 사용된 모든 사용자 정의 레이블의 이름 리스트를 반환합니다.

> **참고**
> Babyl 옵션 섹션의 레이블 목록을 참조하지 않고 사서함에 있는 레이블을 확인하기 위해 실제 메시지를 검사하지만, 사서함이 수정될 때마다 Babyl 섹션이 갱신됩니다.

`Babyl`에 의해 구현된 일부 *`Mailbox`* 메서드는 특별한 주의가 필요합니다:

get_file(*key*)

Babyl 사서함에서, 메시지 헤더는 메시지 본문과 연속적으로 저장되지 않습니다. 파일류 표현을 생성하기 위해, 헤더와 본문이 파일과 동일한 API를 가진 *`io.BytesIO`* 인스턴스에 함께 복사됩니다. 결과적으로, 파일류 객체는 하부 사서함과는 진짜로 독립적이지만 문자열 표현보다 메모리를 절약하지 않습니다.

lock()

unlock()

세 가지 잠금 메커니즘이 사용됩니다—점 잠금(dot locking)과, 사용할 수 있다면, `flock()`과 `lockf()` 시스템 호출.

더 보기

Format of Version 5 Babyl Files
Babyl 형식의 명세.

Rmail로 메일 읽기
Rmail 매뉴얼, Babyl 의미론에 대한 정보가 있습니다.

MMDF 객체

class mailbox.**MMDF**(*path*, *factory=None*, *create=True*)

MMDF 형식의 사서함에 대한 *Mailbox*의 서브 클래스. 매개 변수 *factory*는 파일류 메시지 표현(바이너리 모드에서 열린 것처럼 동작합니다)을 받아들이고 사용자 정의 표현을 반환하는 콜러블 객체입니다. *factory*가 `None`이면, *MMDFMessage*가 기본 메시지 표현으로 사용됩니다. *create*가 `True`이면, 사서함이 없으면 만들어집니다.

MMDF는 메일 전송 에이전트인 Multichannel Memorandum Distribution Facility를 위해 개발된 단일 파일 사서함 형식입니다. 각 메시지는 mbox 메시지와 같은 형식이지만 4개의 Control-A (`'\001'`) 문자를 포함하는 줄로 앞뒤로 둘러쌉니다. mbox 형식과 마찬가지로, 각 메시지의 시작 부분은 처음 5개의 문자가 "From "인 줄로 표시되지만, 메시지를 저장할 때 추가로 등장하는 "From "을 ">From "으로 변환하지 않습니다, 메시지를 구분하는 추가적인 줄로 인해 후속 메시지의 시작으로 착각할 우려가 없기 때문입니다.

`MMDF`에 의해 구현된 일부 *Mailbox* 메서드는 특별한 주의가 필요합니다:

get_bytes(*key*, *from_=False*)

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix "From " line. If *from_* is False, the first line of the file is dropped.

get_file(*key*, *from_=False*)

`MMDF` 인스턴스에서 *flush()*나 *close()*를 호출한 후 파일을 사용하면 예기치 않은 결과가 발생하거나 예외가 발생할 수 있습니다.

Note: This method has an extra parameter (*from_*) compared with other classes. The first line of an mbox file entry is the Unix "From " line. If *from_* is False, the first line of the file is dropped.

lock()

unlock()

세 가지 잠금 메커니즘이 사용됩니다—점 잠금(dot locking)과, 사용할 수 있다면, `flock()`과 `lockf()` 시스템 호출.

더 보기

tin의 mmdf 매뉴얼 페이지
뉴스 리더인 tin의 설명서에 있는 MMDF 형식의 명세.

MMDF
Multichannel Memorandum Distribution Facility를 설명하는 위키피디아 기사.

20.3.2 Message 객체

class mailbox.**Message**(*message=None*)

email.message 모듈 *Message*의 서브 클래스. `mailbox.Message`의 서브 클래스는 사서함 형식별 상태와 동작을 추가합니다.

*message*를 생략하면, 새 인스턴스가 기본 빈 상태로 만들어집니다. *message*가 *email.message.Message* 인스턴스이면, 그 내용이 복사됩니다; 또한, *message*가 `Message` 인스턴스이면 가능한 한 모든 형식별 정보가 변환됩니다. *message*가 문자열, 바이트 문자열 또는 파일이면, **RFC 2822** 호환 메시지를 포함해야 하는데, 읽고 구문 분석됩니다. 파일은 바이너리 모드로 열려야 하지만, 이전 버전과의 호환성을 위해 텍스트 모드 파일이 허용됩니다.

서브 클래스에서 제공하는 형식별 상태와 동작은 다양하지만, 일반적으로 지원되는 특정 사서함에만 고유하지는 않은 속성입니다 (아마도 속성은 특정 사서함 형식에 고유합니다). 예를 들어, 단일 파일 사서함 형식에 대한 파일 오프셋과 디렉터리 기반 사서함 형식에 대한 파일 이름은 유지되지 않는데, 원래 사서함에만 적용되기 때문입니다. 그러나 사용자가 메시지를 읽었는지나 중요하다고 표시되었는지와 같은 상태는 메시지 자체에 적용되므로 유지됩니다.

`Mailbox` 인스턴스를 사용하여 꺼낸 메시지를 나타내는 데 `Message` 인스턴스를 사용할 필요는 없습니다. 때에 따라, `Message` 표현을 생성하는 데 필요한 시간과 메모리가 허용되지 않을 수 있습니다. 이러한 상황에서, `Mailbox` 인스턴스는 문자열과 파일류 표현도 제공하며, `Mailbox` 인스턴스가 초기화될 때 사용자 정의 메시지 팩토리를 지정할 수 있습니다.

`MaildirMessage` 객체

class mailbox.**MaildirMessage**(*message=None*)

Maildir 특정 동작을 갖는 메시지. 매개 변수 *message*는 `Message` 생성자와 같은 의미입니다.

일반적으로, 메일 사용자 에이전트 응용 프로그램은 사용자가 사서함을 처음 열고 닫은 후 `new` 하위 디렉터리의 모든 메시지를 `cur` 하위 디렉터리로 이동하여, 메시지가 실제로 읽혔는지에 관계없이 오래되었음을 기록합니다. `cur`의 각 메시지에는 상태에 대한 정보를 저장하기 위해 파일 이름에 추가된 "info" 섹션이 있습니다. (일부 메일 리더는 `new`의 메시지에 "info" 섹션을 추가할 수도 있습니다.) "info" 섹션은 두 가지 형식 중 하나를 취할 수 있습니다: "2," 와 그 뒤로 표준화된 플래그 목록이 오거나 (예를 들어, "2,FR"), "1," 과 그 뒤로 소위 실험적 정보를 포함할 수 있습니다. Maildir 메시지의 표준 플래그는 다음과 같습니다:

플래그	의미	설명
D	초안	작성 중
F	깃발 표시	중요하다고 표시됨
P	전달됨	전달, 재전송 또는 반송됨
R	답장함	답장함
S	보았음	읽었음
T	휴지통	후에 삭제하기 위해 표시함

`MaildirMessage` 인스턴스는 다음 메서드를 제공합니다:

get_subdir()

"new" (메시지가 `new` 하위 디렉터리에 저장되어야 하면) 나 "cur" (메시지가 `cur` 하위 디렉터리에 저장되어야 하면) 를 반환합니다.

> **참고**
>
> 메시지를 읽었는지에 관계없이, 일반적으로 사서함에 액세스한 후 메시지는 `new`에서 `cur`로 이동됩니다. `"S" in msg.get_flags()`가 `True`이면 메시지 `msg`를 읽은 것입니다.

set_subdir(*subdir*)

메시지를 저장할 하위 디렉터리를 설정합니다. 매개 변수 *subdir*은 "new" 나 "cur" 여야 합니다.

get_flags()

현재 설정된 플래그를 지정하는 문자열을 반환합니다. 메시지가 표준 Maildir 형식을 준수하면, 결과는 `'D'`, `'F'`, `'P'`, `'R'`, `'S'` 및 `'T'`의 각 항목이 알파벳 순서로 0이나 1개 등장하는 이어붙이기입니다. 플래그가 설정되지 않았거나 "info" 에 실험적 의미가 포함되어 있으면 빈 문자열이 반환됩니다.

set_flags(*flags*)

*flags*로 지정된 플래그를 설정하고 다른 모든 플래그를 설정 해제합니다.

add_flag(*flag*)

다른 플래그를 변경하지 않고 *flag*로 지정된 플래그를 설정합니다. 한 번에 둘 이상의 플래그를

추가하기 위해, *flag*가 둘 이상의 문자로 구성된 문자열일 수 있습니다. 플래그 대신 실험적 정보를 포함하는지와 관계없이 현재 "info"를 덮어씁니다.

remove_flag(*flag*)

다른 플래그를 변경하지 않고 *flag*에 의해 지정된 플래그를 설정 해제합니다. 한 번에 둘 이상의 플래그를 제거하기 위해, *flag*는 둘 이상의 문자로 구성된 문자열일 수 있습니다. "info"에 플래그 대신 실험적 정보가 포함되어 있으면, 현재 "info"가 수정되지 않습니다.

get_date()

메시지의 배달 날짜를 에포크(Epoch) 이후 초를 나타내는 부동 소수점 숫자로 반환합니다.

set_date(*date*)

메시지의 배달 날짜를 *date*로 설정합니다. 이는 에포크(Epoch) 이후 초를 나타내는 부동 소수점 숫자입니다.

get_info()

메시지에 대한 "info"를 포함하는 문자열을 반환합니다. 이것은 실험적인 (즉, 플래그 목록이 아닌) "info"를 액세스하고 수정하는 데 유용합니다.

set_info(*info*)

"info"를 문자열이어야 하는 *info*로 설정합니다.

*mboxMessage*나 *MMDFMessage* 인스턴스를 기반으로 `MaildirMessage` 인스턴스가 만들어질 때, *Status*와 *X-Status* 헤더가 생략되고 다음 변환이 발생합니다:

결과 상태	*mboxMessage*나 *MMDFMessage* 상태
"cur" 하위 디렉터리	O 플래그
F 플래그	F 플래그
R 플래그	A 플래그
S 플래그	R 플래그
T 플래그	D 플래그

MHMessage 인스턴스를 기반으로 `MaildirMessage` 인스턴스가 만들어지면, 다음 변환이 발생합니다:

결과 상태	*MHMessage* 상태
"cur" 하위 디렉터리	"unseen" 시퀀스
"cur" 하위 디렉터리와 S 플래그	"unseen" 시퀀스 없음
F 플래그	"flagged" 시퀀스
R 플래그	"replied" 시퀀스

BabylMessage 인스턴스를 기반으로 `MaildirMessage` 인스턴스가 만들어지면 다음 변환이 발생합니다:

결과 상태	*BabylMessage* 상태
"cur" 하위 디렉터리	"unseen" 레이블
"cur" 하위 디렉터리와 S 플래그	"unseen" 레이블 없음
P 플래그	"forwarded" 나 "resent" 레이블
R 플래그	"answered" 레이블
T 플래그	"deleted" 레이블

mboxMessage 객체

class mailbox.**mboxMessage**(*message=None*)

mbox 특정 동작을 갖는 메시지. 매개 변수 *message*는 *Message* 생성자와 같은 의미입니다.

mbox 사서함의 메시지는 단일 파일에 함께 저장됩니다. 보낸 사람의 봉투 주소(envelope address)와 배달 시간은 일반적으로 메시지의 시작을 나타내는 데 사용되는 "From "으로 시작하는 줄에 저장되

지만, mbox 구현 간에 이 데이터의 정확한 형식에는 상당한 차이가 있습니다. 읽었는지나 중요하다고 표시되었는지와 같은 메시지 상태를 나타내는 플래그는 일반적으로 `Status`와 `X-Status` 헤더에 저장됩니다.

mbox 메시지의 전통적인 플래그는 다음과 같습니다:

플래그	의미	설명
R	읽었음	읽었음
O	오래되었음	전에 MUA에서 감지됨
D	삭제됨	후에 삭제하기 위해 표시함
F	깃발 표시	중요하다고 표시됨
A	답변함	답장함

"R" 과 "O" 플래그는 `Status` 헤더에 저장되고, "D", "F" 및 "A" 플래그는 `X-Status` 헤더에 저장됩니다. 플래그와 헤더는 일반적으로 언급된 순서대로 나타납니다.

`mboxMessage` 인스턴스는 다음 메서드를 제공합니다:

get_from()

mbox 사서함에서 메시지의 시작을 표시하는 "From " 줄을 나타내는 문자열을 반환합니다. 선행 "From "과 후행 줄넘김은 제외됩니다.

set_from(*from_*, *time_=None*)

"From " 줄을 *from_*으로 설정합니다. *from_*은 선행 "From "이나 후행 줄넘김 없이 지정되어야 합니다. 편의를 위해, *time_*을 지정할 수 있으며 적절하게 포맷해서 *from_*에 추가합니다. *time_*이 지정되면, `time.struct_time` 인스턴스, `time.strftime()`로 전달하기에 적합한 튜플 또는 `True`(`time.gmtime()` 사용)여야 합니다.

get_flags()

현재 설정된 플래그를 지정하는 문자열을 반환합니다. 메시지가 전통적인 형식을 준수하면, 결과는 `'R'`, `'O'`, `'D'`, `'F'` 및 `'A'` 각각이 이 순서로 0이나 1회 등장하도록 이어붙인 것입니다.

set_flags(*flags*)

*flags*에서 지정한 플래그를 설정하고 다른 모든 플래그를 설정 해제합니다. 매개 변수 *flags*는 `'R'`, `'O'`, `'D'`, `'F'` 및 `'A'` 각각이 0개 이상 등장하도록 임의의 순서로 이어붙인 것이어야 합니다.

add_flag(*flag*)

다른 플래그를 변경하지 않고 *flag*로 지정된 플래그를 설정합니다. 한 번에 둘 이상의 플래그를 추가하기 위해, *flag*가 둘 이상의 문자로 구성된 문자열일 수 있습니다.

remove_flag(*flag*)

다른 플래그를 변경하지 않고 *flag*에 의해 지정된 플래그를 설정 해제합니다. 한 번에 둘 이상의 플래그를 제거하기 위해, *flag*는 둘 이상의 문자로 구성된 문자열일 수 있습니다.

`MaildirMessage` 인스턴스를 기반으로 `mboxMessage` 인스턴스가 만들어지면, `MaildirMessage` 인스턴스의 배달 날짜를 기반으로 "From " 줄이 생성되고, 다음과 같은 변환이 발생합니다:

결과 상태	`MaildirMessage` 상태
R 플래그	S 플래그
O 플래그	"cur" 하위 디렉터리
D 플래그	T 플래그
F 플래그	F 플래그
A 플래그	R 플래그

`MHMessage` 인스턴스를 기반으로 `mboxMessage` 인스턴스가 만들어지면, 다음과 같은 변환이 발생합니다:

결과 상태	MHMessage 상태
R 플래그와 O 플래그	"unseen" 시퀀스 없음
O 플래그	"unseen" 시퀀스
F 플래그	"flagged" 시퀀스
A 플래그	"replied" 시퀀스

BabylMessage 인스턴스를 기반으로 mboxMessage 인스턴스가 만들어지면, 다음과 같은 변환이 발생합니다:

결과 상태	BabylMessage 상태
R 플래그와 O 플래그	"unseen" 레이블 없음
O 플래그	"unseen" 레이블
D 플래그	"deleted" 레이블
A 플래그	"answered" 레이블

MMDFMessage 인스턴스를 기반으로 mboxMessage 인스턴스가 만들어지면, "From " 줄이 복사되고 모든 플래그가 직접 대응됩니다:

결과 상태	MMDFMessage 상태
R 플래그	R 플래그
O 플래그	O 플래그
D 플래그	D 플래그
F 플래그	F 플래그
A 플래그	A 플래그

MHMessage 객체

class mailbox.**MHMessage**(*message=None*)

MH 특정 동작을 갖는 메시지. 매개 변수 *message*는 Message 생성자와 같은 의미입니다.

MH 메시지는 전통적인 의미에서 마크나 플래그를 지원하지 않지만, 임의 메시지의 논리적 그룹인 시퀀스를 지원합니다. 일부 메일 읽기 프로그램(표준 **mh**와 **nmh**는 아니지만)은 다음과 같이 다른 형식에서 플래그를 사용하는 것과 거의 같은 방식으로 시퀀스를 사용합니다:

시퀀스	설명
unseen	읽지 않았지만, 이전에 MUA에서 감지했습니다.
replied	답장함
flagged	중요하다고 표시됨

MHMessage 인스턴스는 다음 메서드를 제공합니다:

get_sequences()

이 메시지를 포함하는 시퀀스의 이름 리스트를 반환합니다.

set_sequences(*sequences*)

이 메시지를 포함하는 시퀀스의 리스트를 설정합니다.

add_sequence(*sequence*)

이 메시지를 포함하는 시퀀스의 리스트에 *sequence*를 추가합니다.

remove_sequence(*sequence*)

이 메시지를 포함하는 시퀀스의 리스트에서 *sequence*를 제거합니다.

MaildirMessage 인스턴스를 기반으로 MHMessage 인스턴스가 만들어지면, 다음과 같은 변환이 발생합니다:

결과 상태	*MaildirMessage* 상태
"unseen" 시퀀스	S 플래그 없음
"replied" 시퀀스	R 플래그
"flagged" 시퀀스	F 플래그

*mboxMessage*나 *MMDFMessage* 인스턴스를 기반으로 MHMessage 인스턴스가 만들어지면, *Status*와 *X-Status* 헤더가 생략되고 다음과 같은 변환이 발생합니다:

결과 상태	*mboxMessage*나 *MMDFMessage* 상태
"unseen" 시퀀스	R 플래그 없음
"replied" 시퀀스	A 플래그
"flagged" 시퀀스	F 플래그

BabylMessage 인스턴스를 기반으로 MHMessage 인스턴스가 만들어지면, 다음과 같은 변환이 발생합니다:

결과 상태	*BabylMessage* 상태
"unseen" 시퀀스	"unseen" 레이블
"replied" 시퀀스	"answered" 레이블

BabylMessage 객체

class mailbox.**BabylMessage**(*message=None*)

Babyl 특정 동작을 갖는 메시지. 매개 변수 *message*는 *Message* 생성자와 같은 의미입니다.

어트리뷰트(*attributes*)라고 부르는 특정 메시지 레이블은 관례에 따라 특별한 의미를 갖도록 정의됩니다. 어트리뷰트는 다음과 같습니다:

레이블	설명
unseen	읽지 않았지만, 이전에 MUA에서 감지했습니다.
deleted	후에 삭제하기 위해 표시함
filed	다른 파일이나 사서함에 복사됨
answered	답장함
forwarded	전달됨
edited	사용자가 수정했음
resent	다시 보냈음

기본적으로, Rmail은 가시적 헤더만 표시합니다. 그러나 BabylMessage 클래스는 더 완전하기 때문에 원래 헤더를 사용합니다. 원하면 가시적 헤더에 명시적으로 액세스 할 수 있습니다.

BabylMessage 인스턴스는 다음 메서드를 제공합니다:

get_labels()

메시지의 레이블 리스트를 반환합니다.

set_labels(*labels*)

메시지의 레이블 리스트를 *labels*로 설정합니다.

add_label(*label*)

메시지의 레이블 리스트에 *label*을 추가합니다.

remove_label(*label*)

메시지의 레이블 리스트에서 *label*을 제거합니다.

get_visible()

헤더가 메시지의 가시적 헤더이고 본문이 비어있는 `Message` 인스턴스를 반환합니다.

set_visible(*visible*)

메시지의 가시적 헤더를 *message*의 헤더와 같게 설정합니다. 매개 변수 *visible*은 `Message` 인스턴스, `email.message.Message` 인스턴스, 문자열 또는 파일류 객체(텍스트 모드로 열어야 합니다)여야 합니다.

update_visible()

`BabylMessage` 인스턴스의 원래 헤더가 수정될 때, 가시적 헤더는 일치하도록 자동으로 수정되지 않습니다. 이 메서드는 다음과 같이 가시적 헤더를 갱신합니다: 해당 원본 헤더가 있는 각 가시적 헤더는 원래 헤더의 값으로 설정되고, 해당 원본 헤더가 없는 각 가시적 헤더는 제거되며 원래 헤더에는 있지만, 가시적 헤더에는 없는 `Date`, `From`, `Reply-To`, `To`, `CC` 및 `Subject`는 모두 가시적 헤더에 추가됩니다.

`MaildirMessage` 인스턴스를 기반으로 `BabylMessage` 인스턴스가 만들어지면, 다음과 같은 변환이 발생합니다:

결과 상태	`MaildirMessage` 상태
"unseen" 레이블	S 플래그 없음
"deleted" 레이블	T 플래그
"answered" 레이블	R 플래그
"forwarded" 레이블	P 플래그

`mboxMessage`나 `MMDFMessage` 인스턴스를 기반으로 `BabylMessage` 인스턴스가 만들어지면, `Status`와 `X-Status` 헤더가 생략되고 다음과 같은 변환이 발생합니다:

결과 상태	`mboxMessage`나 `MMDFMessage` 상태
"unseen" 레이블	R 플래그 없음
"deleted" 레이블	D 플래그
"answered" 레이블	A 플래그

`MHMessage` 인스턴스를 기반으로 `BabylMessage` 인스턴스가 만들어지면, 다음과 같은 변환이 발생합니다:

결과 상태	`MHMessage` 상태
"unseen" 레이블	"unseen" 시퀀스
"answered" 레이블	"replied" 시퀀스

MMDFMessage 객체

class mailbox.**MMDFMessage**(*message=None*)

MMDF 특정 동작을 갖는 메시지. 매개 변수 *message*는 `Message` 생성자와 같은 의미입니다.

mbox 사서함의 메시지와 마찬가지로, MMDF 메시지는 보낸 사람의 주소와 배달 날짜가 "From "으로 시작하는 첫 줄에 저장됩니다. 마찬가지로, 메시지의 상태를 나타내는 플래그는 일반적으로 `Status`와 `X-Status` 헤더에 저장됩니다.

MMDF 메시지의 전통적인 플래그는 mbox 메시지의 플래그와 동일하며 다음과 같습니다:

플래그	의미	설명
R	읽었음	읽었음
O	오래되었음	전에 MUA에서 감지됨
D	삭제됨	후에 삭제하기 위해 표시함
F	깃발 표시	중요하다고 표시됨
A	답변함	답장함

"R"과 "O" 플래그는 `Status` 헤더에 저장되고, "D", "F" 및 "A" 플래그는 `X-Status` 헤더에 저장됩니다. 플래그와 헤더는 일반적으로 언급된 순서대로 나타납니다.

`MMDFMessage` 인스턴스는 `mboxMessage`에서 제공하는 것과 동일한 다음 메서드를 제공합니다:

get_from()

mbox 사서함에서 메시지의 시작을 표시하는 "From " 줄을 나타내는 문자열을 반환합니다. 선행 "From "과 후행 줄넘김은 제외됩니다.

set_from(*from_*, *time_=None*)

"From " 줄을 *from_*으로 설정합니다. *from_*은 선행 "From "이나 후행 줄넘김 없이 지정되어야 합니다. 편의를 위해, *time_*을 지정할 수 있으며 적절하게 포맷해서 *from_*에 추가합니다. *time_*이 지정되면, `time.struct_time` 인스턴스, `time.strftime()`로 전달하기에 적합한 튜플 또는 `True`(`time.gmtime()` 사용)여야 합니다.

get_flags()

현재 설정된 플래그를 지정하는 문자열을 반환합니다. 메시지가 전통적인 형식을 준수하면, 결과는 `'R'`, `'O'`, `'D'`, `'F'` 및 `'A'` 각각이 이 순서로 0이나 1회 등장하도록 이어붙인 것입니다.

set_flags(*flags*)

*flags*에서 지정한 플래그를 설정하고 다른 모든 플래그를 설정 해제합니다. 매개 변수 *flags*는 `'R'`, `'O'`, `'D'`, `'F'` 및 `'A'` 각각이 0개 이상 등장하도록 임의의 순서로 이어붙인 것이어야 합니다.

add_flag(*flag*)

다른 플래그를 변경하지 않고 *flag*로 지정된 플래그를 설정합니다. 한 번에 둘 이상의 플래그를 추가하기 위해, *flag*가 둘 이상의 문자로 구성된 문자열일 수 있습니다.

remove_flag(*flag*)

다른 플래그를 변경하지 않고 *flag*에 의해 지정된 플래그를 설정 해제합니다. 한 번에 둘 이상의 플래그를 제거하기 위해, *flag*는 둘 이상의 문자로 구성된 문자열일 수 있습니다.

`MaildirMessage` 인스턴스를 기반으로 `MMDFMessage` 인스턴스가 만들어지면, `MaildirMessage` 인스턴스의 배달 날짜를 기반으로 "From " 줄이 생성되고, 다음과 같은 변환이 발생합니다:

결과 상태	`MaildirMessage` 상태
R 플래그	S 플래그
O 플래그	"cur" 하위 디렉터리
D 플래그	T 플래그
F 플래그	F 플래그
A 플래그	R 플래그

`MHMessage` 인스턴스를 기반으로 `MMDFMessage` 인스턴스가 만들어지면, 다음과 같은 변환이 발생합니다:

결과 상태	`MHMessage` 상태
R 플래그와 O 플래그	"unseen" 시퀀스 없음
O 플래그	"unseen" 시퀀스
F 플래그	"flagged" 시퀀스
A 플래그	"replied" 시퀀스

BabylMessage 인스턴스를 기반으로 MMDFMessage 인스턴스가 만들어지면, 다음과 같은 변환이 발생합니다:

결과 상태	*BabylMessage* 상태
R 플래그와 O 플래그	"unseen" 레이블 없음
O 플래그	"unseen" 레이블
D 플래그	"deleted" 레이블
A 플래그	"answered" 레이블

mboxMessage 인스턴스를 기반으로 MMDFMessage 인스턴스가 만들어지면, "From " 줄이 복사되고 모든 플래그가 직접 대응됩니다:

결과 상태	*mboxMessage* 상태
R 플래그	R 플래그
O 플래그	O 플래그
D 플래그	D 플래그
F 플래그	F 플래그
A 플래그	A 플래그

20.3.3 예외

다음 예외 클래스가 mailbox 모듈에 정의되어 있습니다:

exception mailbox.**Error**

기타 모든 다른 모듈 특정 예외에 대한 베이스 클래스.

exception mailbox.**NoSuchMailboxError**

사서함이 기대되지만 찾을 수 없을 때 발생합니다, 가령 존재하지 않는 경로로 (그리고 False로 설정된 *create* 매개 변수를 사용하여) *Mailbox* 서브 클래스를 인스턴스 화하거나, 존재하지 않는 폴더를 열 때.

exception mailbox.**NotEmptyError**

사서함이 비어 있지 않지만 비어있을 것으로 기대될 때 발생합니다, 가령 메시지가 포함된 폴더를 삭제할 때.

exception mailbox.**ExternalClashError**

프로그램의 제어를 벗어난 일부 사서함 관련 조건으로 인해 진행할 수 없을 때 발생합니다, 가령 다른 프로그램이 이미 잠금을 보유하고 있는 잠금을 획득하지 못할 때나 고유하게 생성된 파일 이름이 이미 존재할 때.

exception mailbox.**FormatError**

파일의 데이터를 구문 분석할 수 없을 때 발생합니다, 가령 *MH* 인스턴스가 손상된 .mh_sequences 파일을 읽으려고 할 때.

20.3.4 예

사서함에 있는 모든 흥미롭게 보이는 메시지의 제목을 인쇄하는 간단한 예:

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']        # None일 수 있습니다.
    if subject and 'python' in subject.lower():
        print(subject)
```

변환할 수 있는 모든 형식별 정보를 변환하면서, Babyl 사서함에서 MH 사서함으로 모든 메일을 복사하려면:

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

이 예에서는 여러 메일링 리스트로부터의 메일을 다른 사서함으로 정렬하여 넣고, 다른 프로그램에 의한 동시 수정으로 인한 메일 손상, 프로그램 중단으로 인한 메일 손실 또는 사서함에 있는 잘못된 메시지로 인한 조기 종료를 방지하도록 주의합니다:

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = {name: mailbox.mbox('~/email/%s' % name) for name in list_names}
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue                # 메시지가 잘못되었습니다. 그냥 둡니다.

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # 사용할 사서함을 얻습니다
            box = boxes[name]

            # 원본을 제거하기 전에 사본을 디스크에 기록합니다.
            # 충돌이 있으면, 메시지가 중복될 수 있지만, 메시지를 완전히 잃어버리는
            # 것보다는 낫습니다.
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # 원본 메시지 제거
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break               # 목적지를 찾았으니, 그만 찾습니다.

for box in boxes.itervalues():
    box.close()
```

20.4 mimetypes — 파일명을 MIME 유형에 매핑

소스 코드: Lib/mimetypes.py

---

mimetypes 모듈은 파일명이나 URL과 파일명 확장자와 연관된 MIME 유형 간의 변환을 제공합니다. 변

환은 파일명에서 MIME 유형으로, MIME 유형에서 파일 이름 확장자로 제공됩니다; 후자의 변환에서는 인코딩이 지원되지 않습니다.

이 모듈은 하나의 클래스와 여러 편리 함수를 제공합니다. 함수가 이 모듈에 대한 일반 인터페이스이지만, 일부 응용 프로그램은 클래스에도 관심이 있을 수 있습니다.

아래에 설명된 함수는 이 모듈의 기본 인터페이스를 제공합니다. 모듈이 초기화되지 않았으면, 함수가 `init()`가 설정하는 정보에 의존하면 `init()`를 호출합니다.

`mimetypes.`**`guess_type`**(*url*, *strict=True*)

*url*로 주어진 파일명, 경로 또는 URL에 기반한 파일의 유형을 추측합니다. URL은 문자열이나 경로류 객체일 수 있습니다.

반환 값은 튜플 `(type, encoding)` 인데, *type*은 MIME `content-type` 헤더에 사용할 수 있는 `'type/subtype'` 형식의 문자열이거나, 유형을 추측할 수 없으면 (없거나 알려지지 않은 확장자) `None` 입니다.

*encoding*은 인코딩에 사용된 프로그램의 이름(예를 들어, **compress**나 **gzip**)이거나, 인코딩이 없으면 `None` 입니다. 인코딩은 `Content-Encoding` 헤더로 사용하기에 적합합니다, `Content-Transfer-Encoding` 헤더가 **아닙니다**. 매핑은 테이블 기반입니다. 인코딩 접미사는 대소문자를 구분합니다; 유형 접미사는 먼저 대소 문자를 구분해서 시도한 후에, 대소 문자를 구분하지 않고 시도합니다.

The optional *strict* argument is a flag specifying whether the list of known MIME types is limited to only the official types registered with IANA. However, the behavior of this module also depends on the underlying operating system. Only file types recognized by the OS or explicitly registered with Python' s internal database can be identified. When *strict* is `True` (the default), only the IANA types are supported; when *strict* is `False`, some additional non-standard but commonly used MIME types are also recognized.

버전 3.8에서 변경: 경로류 객체 *url*에 대한 지원이 추가되었습니다.

버전 3.13부터 폐지됨: Passing a file path instead of URL is *soft deprecated*. Use `guess_file_type()` for this.

`mimetypes.`**`guess_file_type`**(*path*, **, strict=True*)

*path*로 주어진 경로에 기반하여 파일의 유형을 추측합니다. `guess_type()` 함수와 유사하지만, URL 대신 경로를 받습니다. 경로는 문자열, 바이트열 객체 혹은 경로류 객체일 수 있습니다.

Added in version 3.13.

`mimetypes.`**`guess_all_extensions`**(*type*, *strict=True*)

*type*으로 주어진 MIME 유형을 기반으로 파일의 확장자를 추측합니다. 반환 값은 가능한 모든 파일명 확장자를 제공하는 문자열 리스트인데, 선행 점(`'.'`)을 포함합니다. 확장자는 특정 데이터 스트림과 연관되었음이 보장되지는 않지만, `guess_type()`과 `guess_file_type()`에 의해 MIME 유형 *type*으로 매핑됩니다.

선택적 *strict* 인자는 `guess_type()` 함수에서와 같은 의미를 가집니다.

`mimetypes.`**`guess_extension`**(*type*, *strict=True*)

*type*으로 주어진 MIME 유형을 기반으로 파일의 확장자를 추측합니다. 반환 값은 파일명 확장자를 제공하는 문자열인데, 선행 점(`'.'`)을 포함합니다. 확장자는 특정 데이터 스트림과 연관되었음이 보장되지는 않지만, `guess_type()`과 `guess_file_type()`에 의해 MIME 유형 *type*으로 매핑됩니다. *type*에 대해 추측할 수 있는 확장이 없으면, `None`이 반환됩니다.

선택적 *strict* 인자는 `guess_type()` 함수에서와 같은 의미를 가집니다.

일부 추가 함수와 데이터 항목은 모듈의 동작을 제어하는 데 사용할 수 있습니다.

`mimetypes.`**`init`**(*files=None*)

내부 데이터 구조를 초기화합니다. 주어지면, *files*는 기본 유형 맵을 보강하는 데 사용해야 하는 파일 이름의 시퀀스여야 합니다. 생략하면, 사용할 파일 이름은 `knownfiles`에서 가져옵니다; 윈도우에서는, 현재 레지스트리 설정이 로드됩니다. *files*나 `knownfiles`에서 명명된 각 파일은 그 앞에서 명명된 파일보다 우선합니다. 반복적으로 `init()`를 호출할 수 있습니다.

*files*에 빈 리스트를 지정하면 시스템 기본값이 적용되지 않습니다: 내장 리스트로부터 온 잘 알려진 값만 나타납니다.

*files*가 `None`이면 내부 데이터 구조가 초기 기본값으로 완전히 다시 만들어집니다. 이것은 안정적인 연산이며 여러 번 호출 될 때 같은 결과를 생성합니다.

버전 3.2에서 변경: 이전에는, 윈도우 레지스트리 설정이 무시되었습니다.

`mimetypes.`**`read_mime_types`**(*filename*)

filename 파일이 있으면, 그 파일에 주어진 유형 맵을 로드합니다. 유형 맵은 선행 점(`'.'`)을 포함하는 파일명 확장자를 `'type/subtype'` 형식의 문자열로 매핑하는 딕셔너리로 반환됩니다. *filename* 파일이 없거나 읽을 수 없으면 `None`이 반환됩니다.

`mimetypes.`**`add_type`**(*type*, *ext*, *strict=True*)

MIME 유형 *type*에서 확장자 *ext*로의 매핑을 추가합니다. 확장자가 이미 알려져 있으면, 새 유형이 이전 유형을 대체합니다. 유형이 이미 알려져 있으면, 확장이 알려진 확장 리스트에 추가됩니다.

*strict*가 `True`(기본값)이면, 매핑이 공식 MIME 유형에 추가되고, 그렇지 않으면 비표준 MIME 유형에 추가됩니다.

`mimetypes.`**`inited`**

전역 데이터 구조가 초기화되었는지를 나타내는 플래그. 이것은 *init()*에 의해 `True`로 설정됩니다.

`mimetypes.`**`knownfiles`**

일반적으로 설치된 유형 맵 파일 이름의 리스트입니다. 이 파일들은 일반적으로 `mime.types`로 명명되며 패키지별로 다른 위치에 설치됩니다.

`mimetypes.`**`suffix_map`**

접미사를 접미사에 매핑하는 딕셔너리. 인코딩과 유형이 같은 확장자로 표시되는 인코딩된 파일을 인식하도록 하는 데 사용됩니다. 예를 들어, `.tgz` 확장자는 인코딩과 유형을 별도로 인식 할 수 있도록, `.tar.gz`에 매핑됩니다.

`mimetypes.`**`encodings_map`**

파일명 확장자를 인코딩 유형에 매핑하는 딕셔너리.

`mimetypes.`**`types_map`**

파일명 확장자를 MIME 유형에 매핑하는 딕셔너리.

`mimetypes.`**`common_types`**

파일명 확장자를 비표준이지만 일반적으로 발견되는 MIME 유형에 매핑하는 딕셔너리.

모듈의 사용 예:

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

20.4.1 MimeTypes objects

MimeTypes 클래스는 하나 이상의 MIME 유형 데이터베이스가 필요한 응용 프로그램에 유용 할 수 있습니다. *mimetypes* 모듈과 유사한 인터페이스를 제공합니다.

class `mimetypes.`**`MimeTypes`**(*filenames=()*, *strict=True*)

이 클래스는 MIME 유형 데이터베이스를 나타냅니다. 기본적으로, 이 모듈의 나머지 부분과 같은 데이터베이스에 대한 액세스를 제공합니다. 초기 데이터베이스는 모듈이 제공하는 것의 사본이며, *read()*나 *readfp()* 메서드를 사용하여 추가 `mime.types`-스타일 파일을 데이터베이스에 로드하여 확장할 수 있습니다. 기본 데이터가 필요하지 않으면, 추가 데이터를 로드하기 전에 매핑 딕셔너리를 지울 수도 있습니다.

선택적 *filenames* 매개 변수는 기본 데이터베이스의 "위"에 추가 파일을 로드하게 하는 데 사용할 수 있습니다.

suffix_map

접미사를 접미사에 매핑하는 딕셔너리. 인코딩과 유형이 같은 확장자로 표시되는 인코딩된 파일을 인식하도록 하는 데 사용됩니다. 예를 들어, `.tgz` 확장자는 인코딩과 유형을 별도로 인식 할 수 있도록, `.tar.gz`에 매핑됩니다. 이것은 초기에는 모듈에 정의된 전역 `suffix_map`의 사본입니다.

encodings_map

파일명 확장자를 인코딩 유형에 매핑하는 딕셔너리. 이것은 초기에는 모듈에 정의된 전역 `encodings_map`의 사본입니다.

types_map

파일명 확장자를 MIME 유형으로 매핑하는 두 개의 딕셔너리를 포함하는 튜플: 첫 번째 딕셔너리는 비표준 유형 용이고 두 번째는 표준 유형 용입니다. `common_types`와 `types_map`으로 초기화됩니다.

types_map_inv

MIME 타입을 파일명 확장자 리스트로 매핑하는 두 개의 딕셔너리를 포함하는 튜플: 첫 번째 딕셔너리는 비표준 유형 용이고 두 번째는 표준 유형 용입니다. `common_types`와 `types_map`으로 초기화됩니다.

guess_extension(*type*, *strict=True*)

`guess_extension()` 함수와 유사하고, 객체의 일부로 저장된 테이블을 사용합니다.

guess_type(*url*, *strict=True*)

`guess_type()` 함수와 유사하고, 객체의 일부로 저장된 테이블을 사용합니다.

guess_file_type(*path*, ***, *strict=True*)

`guess_file_type()` 함수와 유사하고, 객체의 일부로 저장된 테이블을 사용합니다.

Added in version 3.13.

guess_all_extensions(*type*, *strict=True*)

`guess_all_extensions()` 함수와 유사하고, 객체의 일부로 저장된 테이블을 사용합니다.

read(*filename*, *strict=True*)

*filename*이라는 이름의 파일에서 MIME 정보를 로드합니다. `readfp()`를 사용하여 파일을 구문 분석합니다.

*strict*가 `True`이면, 정보는 표준 유형 리스트에 추가되고, 그렇지 않으면 비표준 유형 리스트에 추가됩니다.

readfp(*fp*, *strict=True*)

열린 파일 *fp*에서 MIME 유형 정보를 로드합니다. 파일은 표준 `mime.types` 파일의 형식이어야 합니다.

*strict*가 `True`이면, 정보는 표준 유형 리스트에 추가되고, 그렇지 않으면 비표준 유형 리스트에 추가됩니다.

read_windows_registry(*strict=True*)

윈도우 레지스트리에서 MIME 유형 정보를 로드합니다.

가용성: Windows.

*strict*가 `True`이면, 정보는 표준 유형 리스트에 추가되고, 그렇지 않으면 비표준 유형 리스트에 추가됩니다.

Added in version 3.2.

add_type(*type*, *ext*, *strict=True*)

Add a mapping from the MIME type *type* to the extension *ext*. Valid extensions start with a '.' or are empty. When the extension is already known, the new type will replace the old one. When the type is already known the extension will be added to the list of known extensions.

*strict*가 `True`(기본값)이면, 매핑이 공식 MIME 유형에 추가되고, 그렇지 않으면 비표준 MIME 유형에 추가됩니다.

Deprecated since version 3.14, will be removed in version 3.16: Invalid, undotted extensions will raise a `ValueError` in Python 3.16.

20.4.2 Command-line usage

The `mimetypes` module can be executed as a script from the command line.

```
python -m mimetypes [-h] [-e] [-l] type [type ...]
```

The following options are accepted:

-h
--help

Show the help message and exit.

-e
--extension

Guess extension instead of type.

-l
--lenient

Additionally search for some common, but non-standard types.

By default the script converts MIME types to file extensions. However, if `--extension` is specified, it converts file extensions to MIME types.

For each `type` entry, the script writes a line into the standard output stream. If an unknown type occurs, it writes an error message into the standard error stream and exits with the return code `1`.

20.4.3 Command-line example

Here are some examples of typical usage of the `mimetypes` command-line interface:

```
$ # get a MIME type by a file name
$ python -m mimetypes filename.png
type: image/png encoding: None

$ # get a MIME type by a URL
$ python -m mimetypes https://example.com/filename.txt
type: text/plain encoding: None

$ # get a complex MIME type
$ python -m mimetypes filename.tar.gz
type: application/x-tar encoding: gzip

$ # get a MIME type for a rare file extension
$ python -m mimetypes filename.pict
error: unknown extension of filename.pict

$ # now look in the extended database built into Python
$ python -m mimetypes --lenient filename.pict
type: image/pict encoding: None
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
$ # get a file extension by a MIME type
$ python -m mimetypes --extension text/javascript
.js

$ # get a file extension by a rare MIME type
$ python -m mimetypes --extension text/xul
error: unknown type text/xul

$ # now look in the extended database again
$ python -m mimetypes --extension --lenient text/xul
.xul

$ # try to feed an unknown file extension
$ python -m mimetypes filename.sh filename.nc filename.xxx filename.txt
type: application/x-sh encoding: None
type: application/x-netcdf encoding: None
error: unknown extension of filename.xxx

$ # try to feed an unknown MIME type
$ python -m mimetypes --extension audio/aac audio/opus audio/future audio/x-wav
.aac
.opus
error: unknown type audio/future
```

20.5 base64 — Base16, Base32, Base64, Base85 데이터 인코딩

소스 코드: Lib/base64.py

This module provides functions for encoding binary data to printable ASCII characters and decoding such encodings back to binary data. This includes the *encodings specified in* **RFC 4648** (Base64, Base32 and Base16) and the non-standard *Base85 encodings*.

이 모듈은 두 개의 인터페이스를 제공합니다. 최신 인터페이스는 바이트열류 객체를 ASCII `bytes`로 인코딩하고, 바이트열류 객체나 ASCII를 포함하는 문자열을 `bytes`로 디코딩하는 것을 지원합니다. **RFC 4648**에 정의된 두 가지(일반, 그리고 URL과 파일 시스템에서 안전한) base-64 알파벳이 모두 지원됩니다.

The *legacy interface* does not support decoding from strings, but it does provide functions for encoding and decoding to and from *file objects*. It only supports the Base64 standard alphabet, and it adds newlines every 76 characters as per **RFC 2045**. Note that if you are looking for **RFC 2045** support you probably want to be looking at the `email` package instead.

버전 3.3에서 변경: ASCII 전용 유니코드 문자열은 이제 최신 인터페이스의 디코딩 함수가 받아들입니다.

버전 3.4에서 변경: 모든 바이트열류 객체는 이제 이 모듈의 모든 인코딩과 디코딩 함수가 받아들입니다. Ascii85/Base85 지원이 추가되었습니다.

20.5.1 RFC 4648 Encodings

The **RFC 4648** encodings are suitable for encoding binary data so that it can be safely sent by email, used as parts of URLs, or included as part of an HTTP POST request.

`base64.b64encode`(*s*, *altchars=None*)

Base64를 사용하여 바이트열류 객체 *s*를 인코딩하고 인코딩된 `bytes`를 반환합니다.

선택적 *altchars*는 +와 / 문자의 대체 알파벳을 지정하는 길이 2의 바이트열류 객체여야 합니다. 이를 통해 응용 프로그램은 URL이나 파일 시스템에서 안전한 Base64 문자열을 생성할 수 있습니다. 기본값은 `None`이며, 표준 Base64 알파벳이 사용됩니다.

May assert or raise a `ValueError` if the length of *altchars* is not 2. Raises a `TypeError` if *altchars* is not a *bytes-like object*.

base64.**b64decode**(*s*, *altchars=None*, *validate=False*)

Base64로 인코딩된 바이트열류 객체나 ASCII 문자열 *s*를 디코딩하고 디코딩된 `bytes`를 반환합니다.

선택적 *altchars*는 +와 / 문자 대신에 사용되는 대체 알파벳을 지정하는 길이 2의 바이트열류 객체나 ASCII 문자열이어야 합니다.

*s*가 잘못 채워지면(padded) `binascii.Error` 예외가 발생합니다.

*validate*가 `False`(기본값)면, 일반 base-64 알파벳도 대체 알파벳도 아닌 문자는 채우기(padding) 검사 전에 버려집니다. *validate*가 `True`면, 입력에 이 알파벳이 아닌 문자가 있으면 `binascii.Error`가 발생합니다.

For more information about the strict base64 check, see `binascii.a2b_base64()`

May assert or raise a `ValueError` if the length of *altchars* is not 2.

base64.**standard_b64encode**(*s*)

표준 Base64 알파벳을 사용하여 바이트열류 객체 *s*를 인코딩하고 인코딩된 `bytes`를 반환합니다.

base64.**standard_b64decode**(*s*)

표준 Base64 알파벳을 사용하여 바이트열류 객체나 ASCII 문자열 *s*를 디코딩하고 디코딩된 `bytes`를 반환합니다.

base64.**urlsafe_b64encode**(*s*)

표준 Base64 알파벳에서 + 대신 -를, / 대신 _를 사용하도록 대체한 URL과 파일 시스템에서 안전한 알파벳을 사용하여 바이트열류 객체 *s*를 인코딩하고, 인코딩된 `bytes`를 반환합니다. 결과에는 여전히 =가 포함될 수 있습니다.

base64.**urlsafe_b64decode**(*s*)

표준 Base64 알파벳에서 + 대신 -를, / 대신 _를 사용하도록 대체한 URL과 파일 시스템에서 안전한 알파벳을 사용하여 바이트열류 객체나 ASCII 문자열 *s*를 디코딩하고, 디코딩된 `bytes`를 반환합니다.

base64.**b32encode**(*s*)

Base32를 사용하여 바이트열류 객체 *s*를 인코딩하고 인코딩된 `bytes`를 반환합니다.

base64.**b32decode**(*s*, *casefold=False*, *map01=None*)

Base32로 인코딩된 바이트열류 객체나 ASCII 문자열 *s*를 디코딩하고 디코딩된 `bytes`를 반환합니다.

선택적 *casefold*는 소문자 알파벳을 입력으로 사용할 수 있는지를 지정하는 플래그입니다. 보안을 위해, 기본값은 `False`입니다.

RFC 4648은 숫자 0(영)을 글자 O(오)로 선택적으로 매핑하고, 숫자 1(일)을 글자 I(아이)나 글자 L(엘)로 선택적으로 매핑할 수 있게 합니다. 선택적 인자 *map01*이 `None`이 아니면, 숫자 1이 매핑되어야 하는 글자를 지정합니다 (*map01*이 `None`이 아닐 때, 숫자 0은 항상 글자 O로 매핑됩니다). 보안상의 이유로 기본값은 `None`이고, 0과 1은 입력에 허용되지 않습니다.

*s*가 잘못 채워졌거나(padded) 입력에 알파벳이 아닌 문자가 있으면 `binascii.Error`가 발생합니다.

base64.**b32hexencode**(*s*)

Similar to `b32encode()` but uses the Extended Hex Alphabet, as defined in **RFC 4648**.

Added in version 3.10.

base64.**b32hexdecode**(*s*, *casefold=False*)

Similar to `b32decode()` but uses the Extended Hex Alphabet, as defined in **RFC 4648**.

This version does not allow the digit 0 (zero) to the letter O (oh) and digit 1 (one) to either the letter I (eye) or letter L (el) mappings, all these characters are included in the Extended Hex Alphabet and are not interchangeable.

Added in version 3.10.

base64.**b16encode**(*s*)

Base16을 사용하여 바이트열류 객체 *s*를 인코딩하고 인코딩된 `bytes`를 반환합니다.

base64.**b16decode**(*s*, *casefold=False*)

Base16으로 인코딩된 바이트열류 객체나 ASCII 문자열 *s*를 디코딩하고 디코딩된 `bytes`를 반환합니다.

선택적 *casefold*는 소문자 알파벳을 입력으로 사용할 수 있는지를 지정하는 플래그입니다. 보안을 위해, 기본값은 `False`입니다.

*s*가 잘못 채워졌거나(padded) 입력에 알파벳이 아닌 문자가 있으면 `binascii.Error`가 발생합니다.

20.5.2 Base85 Encodings

Base85 encoding is not formally specified but rather a de facto standard, thus different systems perform the encoding differently.

The `a85encode()` and `b85encode()` functions in this module are two implementations of the de facto standard. You should call the function with the Base85 implementation used by the software you intend to work with.

The two functions present in this module differ in how they handle the following:

- Whether to include enclosing <~ and ~> markers
- Whether to include newline characters
- The set of ASCII characters used for encoding
- Handling of null bytes

Refer to the documentation of the individual functions for more information.

base64.**a85encode**(*b*, ***, *foldspaces=False*, *wrapcol=0*, *pad=False*, *adobe=False*)

Ascii85를 사용하여 바이트열류 객체 *b*를 인코딩하고 인코딩된 `bytes`를 반환합니다.

*foldspaces*는 'btoa' 가 지원하듯이 4개의 연속된 스페이스 (ASCII 0x20) 대신 특수한 짧은 시퀀스 'y'를 사용하는 선택적 플래그입니다. 이 기능은 "표준" Ascii85 인코딩에서 지원되지 않습니다.

*wrapcol*은 출력에 줄 바꿈 (`b'\n'`) 문자가 추가되어야 하는지를 제어합니다. 이것이 0이 아니면, 각 출력 줄의 길이는 후행 줄 바꿈을 제외하고 이 문자 수를 넘지 않습니다.

*pad*는 인코딩하기 전에 입력이 4의 배수로 채워졌는지를 제어합니다. `btoa` 구현은 항상 채운다는 것에 유의하십시오.

*adobe*는 인코딩된 바이트 시퀀스가 Adobe 구현에서 사용되는 <~와 ~>로 프레임화되는지를 제어합니다.

Added in version 3.4.

base64.**a85decode**(*b*, ***, *foldspaces=False*, *adobe=False*, *ignorechars=b' \t\n\r\x0b'*)

Ascii85로 인코딩된 바이트열류 객체나 ASCII 문자열 *b*를 디코딩하고 디코딩된 `bytes`를 반환합니다.

*foldspaces*는 짧은 시퀀스 'y' 를 4개의 연속 된 스페이스(ASCII 0x20)에 대한 축약으로 받아들여야 하는지를 지정하는 플래그입니다. 이 기능은 "표준" Ascii85 인코딩에서 지원되지 않습니다.

*adobe*는 입력 시퀀스가 Adobe Ascii85 형식인지(즉 <~ 와 ~> 로 프레임화되었는지)를 제어합니다.

*ignorechars*는 입력에서 무시할 문자가 포함된 바이트열류 객체나 ASCII 문자열이어야 합니다. 여기에는 공백 문자만 포함되어야 하며, 기본적으로 ASCII의 모든 공백 문자가 포함됩니다.

Added in version 3.4.

base64.**b85encode**(*b*, *pad=False*)

base85를 사용하여 바이트열류 객체 *b*를 인코딩하고 (예를 들어 git 스타일 바이너리 diff에서 사용되는 것처럼), 인코딩된 `bytes`를 반환합니다.

*pad*가 참이면, 입력은 `b'\0'`으로 채워져서 길이는 인코딩 전에 4바이트의 배수가 됩니다.

Added in version 3.4.

base64.**b85decode**(*b*)

base85로 인코딩된 바이트열류 객체나 ASCII 문자열 *b*를 디코딩하고 디코딩된 *bytes*를 반환합니다. 필요하면, 채우기는 묵시적으로 제거됩니다.

Added in version 3.4.

base64.**z85encode**(*s*)

(ZeroMQ 에서 사용된) Z85를 사용하여 바이트열류 객체 *s*를 인코딩하고, 인코딩된 *bytes*를 반환합니다. 더 자세한 내용은 Z85 명세를 참조하세요.

Added in version 3.13.

base64.**z85decode**(*s*)

Z85로 인코딩된 바이트열류 객체나 ASCII 문자열 *s*를 디코딩하고 디코딩된 *bytes*를 반환합니다. 더 자세한 내용은 Z85 명세를 참조하세요.

Added in version 3.13.

20.5.3 Legacy Interface

base64.**decode**(*input*, *output*)

바이너리 *input* 파일의 내용을 디코딩하고 결과 바이너리 데이터를 *output* 파일에 씁니다. *input*과 *output*은 파일 객체여야 합니다. *input*은 `input.readline()` 이 빈 바이트열 객체를 반환할 때까지 읽힙니다.

base64.**decodebytes**(*s*)

하나 이상의 base64 인코딩된 데이터 줄을 포함해야 하는 바이트열류 객체 *s*를 디코딩하고 디코딩된 *bytes*를 반환합니다.

Added in version 3.1.

base64.**encode**(*input*, *output*)

바이너리 *input* 파일의 내용을 인코딩하고 base64로 인코딩된 결과 데이터를 *output* 파일에 씁니다. *input*과 *output*은 파일 객체여야 합니다. *input*은 `input.read()` 이 빈 바이트열 객체를 반환할 때까지 읽힙니다. *encode()*는 **RFC 2045**(MIME)에 따라 출력의 76바이트마다 개행 문자(`b'\n'`)를 삽입할 뿐만 아니라, 항상 출력이 개행 문자로 끝나도록 합니다.

base64.**encodebytes**(*s*)

임의의 바이너리 데이터를 포함할 수 있는 바이트열류 객체 *s*를 인코딩하고, **RFC 2045** (MIME)에 따라 76바이트의 출력마다 개행(`b'\n'`)이 삽입되고, 후행 개행이 붙은 base64 인코딩된 데이터를 포함하는 *bytes*를 반환합니다.

Added in version 3.1.

모듈 사용 예:

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSB0byBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

20.5.4 Security Considerations

A new security considerations section was added to **RFC 4648** (section 12); it' s recommended to review the security section for any code deployed to production.

더 보기

모듈 *binascii*
ASCII에서 바이너리로, 바이너리에서 ASCII로의 변환이 포함된 지원 모듈.

RFC 1521 - MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies
5.2 절, "Base64 Content-Transfer-Encoding" 은 base64 인코딩의 정의를 제공합니다.

20.6 binascii — 바이너리와 ASCII 간의 변환

binascii 모듈에는 바이너리와 다양한 ASCII 인코딩 바이너리 표현 사이를 변환하는 여러 가지 방법이 포함되어 있습니다. 일반적으로 이러한 함수는 직접 사용하지 않고, 대신 *base64*와 같은 래퍼 모듈을 사용합니다. *binascii* 모듈에는 고수준 모듈에서 사용하는 보다 빠른 속도를 위해 C로 작성된 저수준 함수가 들어 있습니다.

참고

`a2b_*` 함수는 ASCII 문자만 포함하는 유니코드 문자열을 받아들입니다. 다른 함수는 바이트열류 객체(가령 *bytes*, *bytearray* 및 버퍼 프로토콜을 지원하는 다른 객체)만 받아들입니다.

버전 3.3에서 변경: ASCII로만 이루어진 유니코드 문자열은 이제 `a2b_*` 함수에서 허용됩니다.

binascii 모듈은 다음 함수를 정의합니다:

binascii.**a2b_uu**(*string*)
한 줄의 UU 인코딩된 데이터 string을 바이너리로 역변환하고 바이너리 데이터를 반환합니다. 마지막 줄을 제외하고는, 줄은 보통 45 (바이너리) 바이트를 포함합니다. 줄 데이터 뒤에는 공백 문자가 올 수 있습니다.

binascii.**b2a_uu**(*data*, **, *backtick=False*)
바이너리 data를 ASCII 문자의 줄로 변환합니다, 반환 값은 개행 문자를 포함하는 변환 된 줄입니다. *data*의 길이는 최대 45이어야 합니다. *backtick*가 참이면, 0은 스페이스 대신 `'`'`으로 표현됩니다.

버전 3.7에서 변경: *backtick* 매개 변수가 추가되었습니다.

binascii.**a2b_base64**(*string*, */*, **, *strict_mode=False*)
base64 데이터 블록을 바이너리로 역변환하고 바이너리 데이터를 반환합니다. 한 번에 한 줄 이상을 전달할 수 있습니다.

If *strict_mode* is true, only valid base64 data will be converted. Invalid base64 data will raise *binascii.Error*.

Valid base64:

- Conforms to **RFC 3548**.
- Contains only characters from the base64 alphabet.
- Contains no excess data after padding (including excess padding, newlines, etc.).
- Does not start with a padding.

버전 3.11에서 변경: *strict_mode* 매개 변수가 추가되었습니다.

binascii.**b2a_base64**(*data*, **, *newline=True*)
바이너리 data를 base64 코딩으로 ASCII 문자의 줄로 변환합니다. 반환 값은 변환된 줄인데, *newline*이 참이면, 개행 문자를 포함합니다. 이 함수의 출력은 **RFC 3548**을 따릅니다.

버전 3.6에서 변경: *newline* 매개 변수가 추가되었습니다.

binascii.**a2b_qp**(*data*, *header=False*)

quoted-printable data 블록을 바이너리로 역변환하고 바이너리 데이터를 반환합니다. 한 번에 한 줄 이상을 전달할 수 있습니다. 선택적 인자 *header*가 있고 참이면, 밑줄(underscore)은 스페이스로 디코딩됩니다.

binascii.**b2a_qp**(*data*, *quotetabs=False*, *istext=True*, *header=False*)

바이너리 data를 quoted-printable 인코딩으로 ASCII 문자의 줄로 변환합니다. 반환 값은 변환된 줄입니다. 선택적 인자 *quotetabs*가 있고 참이면, 모든 탭과 스페이스가 인코딩됩니다. 선택적 인자 *istext*가 있고 참이면, 개행 문자는 인코딩되지 않지만, 후행 공백은 인코딩됩니다. 선택적 인자 *header*가 있고 참이면, 스페이스는 **RFC 1522**에 따라 밑줄로 인코딩됩니다. 선택적 인자 *header*가 있고 거짓이면, 개행 문자도 함께 인코딩됩니다; 그렇지 않으면 라인 피드(linefeed) 변환이 바이너리 데이터 스트림을 손상할 수 있습니다.

binascii.**crc_hqx**(*data*, *value*)

초기 CRC *value*로 시작하는, *data*의 16비트 CRC 값을 계산하고 결과를 반환합니다. 종종 0x1021로 표시되는, CRC-CCITT 다항식 $x^{16} + x^{12} + x^5 + 1$을 사용합니다. 이 CRC는 binhex4 형식에서 사용됩니다.

binascii.**crc32**(*data*[, *value*])

초기 CRC *value*로 시작하는, *data*의 부호 없는 32비트 체크섬인 CRC-32를 계산합니다. 기본 초기 CRC는 0입니다. 이 알고리즘은 ZIP 파일 체크섬과 일치합니다. 이 알고리즘은 체크섬 알고리즘으로 사용하도록 설계되었으므로, 일반 해시 알고리즘으로 사용하기에 적합하지 않습니다. 다음과 같이 사용하십시오:

```
print(binascii.crc32(b"hello world"))
# 또는, 두 조각으로:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

버전 3.0에서 변경: The result is always unsigned.

binascii.**b2a_hex**(*data*[, *sep*[, *bytes_per_sep=1*]])

binascii.**hexlify**(*data*[, *sep*[, *bytes_per_sep=1*]])

바이너리 *data*의 16진수 표현을 반환합니다. *data*의 모든 바이트는 해당 2자리 16진수 표현으로 변환됩니다. 따라서 반환된 바이트열 객체의 길이는 *data* 의 두 배입니다.

비슷한 기능(하지만 텍스트 문자열을 반환하는)을 `bytes.hex()` 메서드를 사용하여 편리하게 액세스할 수도 있습니다.

*sep*이 지정되면, 단일 문자 문자열이나 바이트열 객체여야 합니다. *bytes_per_sep* 입력 바이트마다 출력에 삽입됩니다. 구분자 배치는 기본적으로 출력의 오른쪽 끝에서부터 계산됩니다. 왼쪽부터 계산하려면, 음수의 *bytes_per_sep* 값을 제공하십시오.

```
>>> import binascii
>>> binascii.b2a_hex(b'\xb9\x01\xef')
b'b901ef'
>>> binascii.hexlify(b'\xb9\x01\xef', '-')
b'b9-01-ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b'_', 2)
b'b9_01ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b' ', -2)
b'b901 ef'
```

버전 3.8에서 변경: *sep*과 *bytes_per_sep* 매개 변수가 추가되었습니다.

binascii.**a2b_hex**(*hexstr*)

binascii.**unhexlify**(*hexstr*)

16진수 문자열 *hexstr*로 표현된 바이너리 데이터를 반환합니다. 이 함수는 `b2a_hex()`의 역함수입

니다. *hexstr*는 짝수개의 16진수(대소문자 모두 가능합니다)를 포함해야 하며, 그렇지 않으면 `Error` 예외가 발생합니다.

비슷한 기능(텍스트 문자열 인자만 받아들이지만, 공백에 대해 더 느슨한)을 `bytes.fromhex()` 클래스 메서드를 사용하여 액세스할 수도 있습니다.

exception binascii.**Error**

에러 시 발생하는 예외. 이들은 대개 프로그래밍 에러입니다.

exception binascii.**Incomplete**

불완전한 데이터에서 발생하는 예외. 이들은 일반적으로 프로그래밍 에러가 아니지만, 조금 더 많은 데이터를 읽고 다시 시도하면 처리될 수 있습니다.

더 보기

모듈 ***base64***
RFC 호환 base64 스타일 인코딩으로, 베이스 16, 32, 64 및 85를 지원합니다.

모듈 ***quopri***
MIME 전자 우편 메시지에 사용되는 quoted-printable 인코딩 지원.

20.7 quopri — MIME quoted-printable 데이터 인코딩과 디코딩

소스 코드: Lib/quopri.py

이 모듈은 **RFC 1521**: "MIME (Multipurpose Internet Mail Extensions) 1부: 인터넷 메시지 본문의 형식을 지정하고 설명하기 위한 메커니즘"에 정의된 대로, quoted-printable 전송 인코딩과 디코딩을 수행합니다. quoted-printable 인코딩은 인쇄할 수 없는 문자가 비교적 적은 데이터를 위해 설계되었습니다; `base64` 모듈을 통해 사용할 수 있는 base64 인코딩 체계는 그래픽 파일을 보낼 때와 같이 그런 문자가 많은 경우 더 압축적입니다.

quopri.**decode**(*input*, *output*, *header=False*)

input 파일의 내용을 디코딩하고 결과로 디코딩된 바이너리 데이터를 *output* 파일에 씁니다. *input* 과 *output*는 바이너리 파일 객체 여야 합니다. 선택적 인자 *header*가 있고 참이면, 밑줄은 스페이스로 디코딩됩니다. 이것은 **RFC 1522**: "MIME (Multipurpose Internet Mail Extensions) 2부: 비 ASCII 텍스트를 위한 메시지 헤더 확장"에서 설명한 대로 "Q"-인코딩된 헤더를 디코딩하는 데 사용됩니다.

quopri.**encode**(*input*, *output*, *quotetabs*, *header=False*)

input 파일의 내용을 인코딩하고 결과 quoted-printable 데이터를 *output* 파일에 씁니다. *input* 과 *output*은 바이너리 파일 객체 여야 합니다. *quotetabs*는 포함 된 스페이스와 탭을 인코딩할지를 제어하는 비 선택적 플래그입니다; 참이면 그러한 공백 문자를 인코드하고, 거짓이면 인코드하지 않고 남겨둡니다. 줄 끝에 나타나는 공백과 탭은 **RFC 1521**에 따라 항상 인코딩됨에 유의하십시오. *header*는 스페이스를 **RFC 1522**에 따라 밑줄로 인코딩할지를 제어하는 플래그입니다.

quopri.**decodestring**(*s*, *header=False*)

`decode()`와 비슷하지만, 소스 `bytes`를 받아들이고 해독된 해당 `bytes`를 반환합니다.

quopri.**encodestring**(*s*, *quotetabs=False*, *header=False*)

`encode()`와 비슷하지만, 소스 `bytes`를 받아들이고 인코딩된 해당 `bytes`를 반환합니다. 기본적으로 `encode()` 함수의 *quotetabs* 매개 변수에 `False` 값을 보냅니다.

더 보기

모듈 ***base64***
MIME base64 데이터 인코딩과 디코딩

CHAPTER 21

구조화된 마크업 처리 도구

파이썬은 다양한 형태의 구조화된 데이터 마크업을 다루는 다양한 모듈을 지원합니다. 여기에는 SGML (Standard Generalized Markup Language) 및 HTML (Hypertext Markup Language) 을 다루는 모듈과 XML (Extensible Markup Language) 작업을 위한 여러 인터페이스가 포함됩니다.

21.1 `html` — 하이퍼텍스트 마크업 언어 지원

소스 코드: Lib/html/__init__.py

이 모듈은 HTML을 조작하는 유틸리티를 정의합니다.

`html.escape`(*s*, *quote=True*)

Convert the characters `&`, < and > in string *s* to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag *quote* is true (the default), the characters (`"`) and (`'`) are also translated; this helps for inclusion in an HTML attribute value delimited by quotes, as in `<a href="...">`. If *quote* is set to false, the characters (`"`) and (`'`) are not translated.

Added in version 3.2.

`html.unescape`(*s*)

문자열 *s*의 모든 이름과 숫자 문자 참조(예를 들어, `>`, `>`, `>`)를 해당 유니코드 문자로 변환합니다. 이 함수는 유효하거나 유효하지 않은 문자 참조 모두에 대해 HTML 5 표준에 정의된 규칙과 *HTML 5 이름 문자 참조 목록*을 사용합니다.

Added in version 3.4.

`html` 패키지의 서브 모듈은 다음과 같습니다:

- *`html.parser`* – 관대한 구문 분석 모드가 있는 HTML/XHTML 구문 분석기
- *`html.entities`* – HTML 엔티티 정의

21.2 html.parser — 간단한 HTML과 XHTML 구문 분석기

소스 코드: Lib/html/parser.py

이 모듈은 HTML(HyperText Mark-up Language)와 XHTML 형식의 텍스트 파일을 구문 분석하기 위한 기초로 사용되는 클래스 *HTMLParser*를 정의합니다.

class html.parser.**HTMLParser**(*, *convert_charrefs=True*)

잘못된 마크업을 구문 분석할 수 있는 구문 분석기 인스턴스를 만듭니다.

*convert_charrefs*가 True(기본값)이면, (script/style 요소에 있는 것을 제외한) 모든 문자 참조(character references)가 자동으로 해당 유니코드 문자로 변환됩니다.

HTMLParser 인스턴스는 HTML 데이터를 받아서 시작 태그, 종료 태그, 텍스트, 주석 및 기타 마크업 요소를 만날 때마다 처리기 메서드를 호출합니다. 사용자는 원하는 동작을 구현하기 위해 *HTMLParser*의 서브 클래스를 만들고 해당 메서드를 재정의해야 합니다.

이 구문 분석기는 종료 태그가 시작 태그와 일치하는지 검사하거나, 바깥(outer) 요소를 닫음으로써 묵시적으로 닫힌 요소에 대해 종료 태그 처리기를 호출하지 않습니다.

버전 3.4에서 변경: *convert_charrefs* 키워드 인자가 추가되었습니다.

버전 3.5에서 변경: 인자 *convert_charrefs*의 기본값은 이제 True입니다.

21.2.1 HTML 구문 분석기 응용 프로그램 예제

As a basic example, below is a simple HTML parser that uses the *HTMLParser* class to print out start tags, end tags, and data as they are encountered:

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag:", tag)

    def handle_endtag(self, tag):
        print("Encountered an end tag :", tag)

    def handle_data(self, data):
        print("Encountered some data  :", data)

parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

출력은 다음과 같습니다:

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

21.2.2 HTMLParser 메서드

HTMLParser 인스턴스에는 다음과 같은 메서드가 있습니다:

HTMLParser.**feed**(*data*)

구문 분석기에 텍스트를 입력합니다. 완전한 요소로 구성되어있는 부분까지 처리됩니다; 불완전한 데이터는 더 많은 데이터가 공급되거나 *close()* 가 호출될 때까지 버퍼링 됩니다. *data* 는 *str* 이어야 합니다.

HTMLParser.**close**()

버퍼링 된 모든 데이터를 마치 파일 끝(end-of-file) 표시가 붙은 것처럼 처리합니다. 이 메서드는 파생 클래스에 의해 입력 끝에서의 추가 처리를 정의하기 위해 재정의될 수 있지만, 재정의된 버전에서는 항상 *HTMLParser* 베이스 클래스 메서드인 *close()* 를 호출해야 합니다.

HTMLParser.**reset**()

인스턴스를 재설정합니다. 처리되지 않은 모든 데이터를 잃습니다. 이것은 인스턴스 생성 시에 묵시적으로 호출됩니다.

HTMLParser.**getpos**()

현재의 줄 번호와 오프셋(offset)을 반환합니다.

HTMLParser.**get_starttag_text**()

가장 최근에 열렸던 시작 태그의 텍스트를 반환합니다. 이것은 일반적으로 구조화된 처리에 필요하지 않지만, "배치된 대로(as deployed)" HTML을 다루거나 최소한의 변경(어트리뷰트 사이의 공백을 보존할 수 있음, 등등)으로 입력을 다시 생성하는 데 유용할 수 있습니다.

다음 메서드는 데이터나 마크업 요소를 만날 때 호출되며 서브 클래스에서 재정의하려는 용도입니다. 베이스 클래스 구현은 아무 일도 하지 않습니다 (*handle_startendtag()* 는 예외입니다).:

HTMLParser.**handle_starttag**(*tag*, *attrs*)

이 메서드는 엘리먼트의 시작 태그(예를 들어, `<div id="main">`)를 처리하기 위해 호출됩니다.

tag 인자는 소문자로 변환된 태그의 이름입니다. *attrs* 인자는 태그의 <> 화살괄호 안에 있는 어트리뷰트를 포함하는 `(name, value)` 쌍의 리스트입니다. *name* 은 소문자로 변환되고, *value* 의 따옴표는 제거되고, 문자와 엔티티 참조는 치환됩니다.

예를 들어, 태그 `<A HREF="https://www.cwi.nl/">` 의 경우, 이 메서드는 `handle_starttag('a', [('href', 'https://www.cwi.nl/')])` 로 호출됩니다.

html.entities 의 모든 엔티티 참조가 어트리뷰트 값에서 치환됩니다.

HTMLParser.**handle_endtag**(*tag*)

이 메서드는 요소의 종료 태그(예를 들어, `</div>`)를 처리하기 위해 호출됩니다.

tag 인자는 소문자로 변환된 태그의 이름입니다.

HTMLParser.**handle_startendtag**(*tag*, *attrs*)

handle_starttag() 와 비슷하지만, 구문 분석기가 XHTML 스타일의 빈 태그(`<img ... />`)를 만날 때 호출됩니다. 이 메서드는 이 특정의 어휘 정보(lexical information)가 필요한 서브 클래스에 의해 재정의될 수 있습니다; 기본 구현은 단순히 *handle_starttag()* 와 *handle_endtag()* 를 호출합니다.

HTMLParser.**handle_data**(*data*)

이 메서드는 임의의 데이터(예를 들어, 텍스트 노드와 `<script>...</script>` 및 `<style>...</style>` 의 내용)를 처리하기 위해 호출됩니다.

HTMLParser.**handle_entityref**(*name*)

이 메서드는 `&name;` 형식(예를 들어, `>`)의 이름있는 문자 참조를 처리하기 위해 호출됩니다. 여기서 *name* 은 일반 엔티티 참조(예를 들어, `'gt'`)입니다. *convert_charrefs* 가 `True` 이면, 이 메서드는 호출되지 않습니다.

HTMLParser.**handle_charref**(*name*)

이 메서드는 `&#NNN;` 과 `&#xNNN;` 형식의 10진수 및 16진수 문자 참조를 처리하기 위해 호출됩니다. 예를 들어, `>` 에 해당하는 10진수는 `>` 이고, 반면에 16진수는 `>` 입니다; 이때 메서드는 `'62'` 나 `'x3E'` 를 받습니다. 이 메서드는 *convert_charrefs* 가 `True` 이면 호출되지 않습니다.

HTMLParser.**handle_comment**(*data*)

이 메서드는 주석을 만날 때 호출됩니다 (예를 들어, `<!--comment-->`).

예를 들어, 주석 `<!-- comment -->`는 이 메서드가 인자 `' comment '`로 호출되도록 합니다.

Internet Explorer 조건부 주석 (condcoms) 의 내용도 이 메서드로 보내지므로, `<!--[if IE 9]>IE9-specific content<![endif]-->`의 경우, 이 메서드는 `'[if IE 9]>IE9-specific content<![endif]'`를 받습니다.

HTMLParser.**handle_decl**(*decl*)

이 메서드는 HTML doctype 선언 (예를 들어, `<!DOCTYPE html>`)을 처리하기 위해 호출됩니다.

decl 매개 변수는 `<!...>` 마크업 내의 선언 전체 내용입니다 (예를 들어, `'DOCTYPE html'`).

HTMLParser.**handle_pi**(*data*)

처리 명령 (processing instruction) 을 만날 때 호출되는 메서드. *data* 매개 변수에는 전체 처리 명령이 포함됩니다. 예를 들어, 처리 명령 `<?proc color='red'>`의 경우, 이 메서드는 `handle_pi("proc color='red'")`로 호출됩니다. 파생 클래스에 의해 재정의되려는 목적입니다; 베이스 클래스 구현은 아무것도 수행하지 않습니다.

> **참고**
>
> *HTMLParser* 클래스는 처리 명령에 대해 SGML 구문 규칙을 사용합니다. 후행 `'?'`를 사용하는 XHTML 처리 명령은 `'?'`가 *data*에 포함되도록 합니다.

HTMLParser.**unknown_decl**(*data*)

이 메서드는 구문 분석기가 인식할 수 없는 선언을 읽었을 때 호출됩니다.

data 매개 변수는 `<![...]>` 마크업 안에 있는 선언의 전체 내용입니다. 파생 클래스가 재정의하는 것이 때때로 유용합니다. 베이스 클래스 구현은 아무것도 수행하지 않습니다.

21.2.3 예제

The following class implements a parser that will be used to illustrate more examples:

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print("     attr:", attr)

    def handle_endtag(self, tag):
        print("End tag  :", tag)

    def handle_data(self, data):
        print("Data     :", data)

    def handle_comment(self, data):
        print("Comment  :", data)

    def handle_entityref(self, name):
        c = chr(name2codepoint[name])
        print("Named ent:", c)

    def handle_charref(self, name):
        if name.startswith('x'):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
            c = chr(int(name[1:], 16))
        else:
            c = chr(int(name))
        print("Num ent  :", c)

    def handle_decl(self, data):
        print("Decl     :", data)

parser = MyHTMLParser()
```

Parsing a doctype:

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl     : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
↪html4/strict.dtd"
```

Parsing an element with a few attributes and a title:

```
>>> parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
     attr: ('src', 'python-logo.png')
     attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

The content of `script` and `style` elements is returned as is, without further parsing:

```
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
     attr: ('type', 'text/css')
Data     : #python { color: green }
End tag  : style

>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello!</strong>");</script>')
Start tag: script
     attr: ('type', 'text/javascript')
Data     : alert("<strong>hello!</strong>");
End tag  : script
```

Parsing comments:

```
>>> parser.feed('<!--a comment-->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment  : a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

Parsing named and numeric character references and converting them to the correct char (note: these 3 references are all equivalent to `'>'`):

```
>>> parser = MyHTMLParser()
>>> parser.feed('&gt;&#62;&#x3E;')
Data     : >>>
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> parser = MyHTMLParser(convert_charrefs=False)
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent  : >
Num ent  : >
```

Feeding incomplete chunks to *feed()* works, but *handle_data()* might be called more than once (unless *convert_charrefs* is set to `True`):

```
>>> for chunk in ['<sp', 'an>buff', 'ered', ' text</s', 'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data     : buff
Data     : ered
Data     :  text
End tag  : span
```

Parsing invalid HTML (e.g. unquoted attributes) also works:

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
     attr: ('class', 'link')
     attr: ('href', '#main')
Data     : tag soup
End tag  : p
End tag  : a
```

21.3 html.entities — HTML 일반 엔티티의 정의

소스 코드: Lib/html/entities.py

이 모듈은 네 가지 딕셔너리 *html5*, *name2codepoint*, *codepoint2name* 와 *entitydefs*를 정의합니다.

html.entities.**html5**

HTML5 이름 문자 참조[1] 를 해당 유니코드 문자에 매핑하는 딕셔너리, 예를 들어 `html5['gt;'] == '>'`. 후미 세미콜론이 이름에 포함됨에 유의하십시오 (예를 들어 `'gt;'`). 하지만 일부 이름은 세미콜론 없이도 표준에서 허용됩니다: 이럴 때 이름은 `';'`가 있거나 없는 형태가 모두 포함됩니다. *html.unescape()*도 참조하십시오.

Added in version 3.3.

html.entities.**entitydefs**

XHTML 1.0 엔티티 정의를 ISO Latin-1의 치환 텍스트에 매핑하는 딕셔너리.

html.entities.**name2codepoint**

HTML4 엔티티 이름을 유니코드 코드 포인트에 매핑하는 딕셔너리.

html.entities.**codepoint2name**

유니코드 코드 포인트를 HTML4 엔티티 이름에 매핑하는 딕셔너리.

[1] https://html.spec.whatwg.org/multipage/named-characters.html#named-character-references 를 참조하십시오.

21.4 XML 처리 모듈

소스 코드: Lib/xml/

XML 처리를 위한 파이썬의 인터페이스는 `xml` 패키지로 묶여있습니다.

> **참고**
>
> If you need to parse untrusted or unauthenticated data, see *XML security*.

`xml` 패키지의 모듈들은 최소한 하나의 SAX 호환 XML 구문 분석기가 있도록 요구함에 유의해야 합니다. Expat 구문 분석기가 파이썬에 포함되어 있으므로, `xml.parsers.expat` 모듈을 항상 사용할 수 있습니다.

`xml.dom` 과 `xml.sax` 패키지에 대한 설명서는 DOM과 SAX 인터페이스에 대한 파이썬 바인딩의 정의입니다.

XML 처리 서브 모듈은 다음과 같습니다:

- `xml.etree.ElementTree`: ElementTree API, 간단하고 가벼운 XML 프로세서
- `xml.dom`: DOM API 정의
- `xml.dom.minidom`: 최소 DOM 구현
- `xml.dom.pulldom`: 부분 DOM 트리 구축 지원
- `xml.sax`: SAX2 베이스 클래스와 편리 함수
- `xml.parsers.expat`: Expat 구문 분석기 바인딩

21.4.1 XML security

An attacker can abuse XML features to carry out denial of service attacks, access local files, generate network connections to other machines, or circumvent firewalls.

Expat versions lower than 2.6.0 may be vulnerable to "billion laughs", "quadratic blowup" and "large tokens". Python may be vulnerable if it uses such older versions of Expat as a system-provided library. Check `pyexpat.EXPAT_VERSION`.

`xmlrpc` is **vulnerable** to the "decompression bomb" attack.

billion laughs(억만 웃음) / exponential entity expansion(지수적 엔티티 확장)
: 지수적 엔티티 확장으로도 알려진, Billion Laughs 공격은 여러 수준의 중첩된 엔티티를 사용합니다. 각 엔티티는 다른 엔티티를 여러 번 참조하며, 최종 엔티티 정의에는 작은 문자열이 포함됩니다. 지수적인 확장으로 수 기가바이트의 텍스트가 생성되고 많은 메모리와 CPU 시간이 소모됩니다.

quadratic blowup entity expansion(이차 폭발 엔티티 확장)
: 이차 폭발 공격은 Billion Laughs 공격과 유사합니다; 이 역시 엔티티 확장을 남용합니다. 중첩된 엔티티 대신 2천 개 이상의 문자를 갖는 커다란 엔티티 하나를 계속 반복합니다. 공격은 지수적인 경우만큼 효율적이지 않지만 깊이 중첩된 엔티티를 금지하는 구문 분석기 대응책을 우회합니다.

decompression bomb(압축해제 폭탄)
: 압축 해제 폭탄(일명 ZIP bomb)은 gzip 압축된 HTTP 스트림이나 LZMA 압축 파일과 같은, 압축된 XML 스트림을 구문 분석할 수 있는 모든 XML 라이브러리에 적용됩니다. 공격자는 전송된 데이터의 양을 3배 이상 줄일 수 있습니다.

large tokens(대형 토큰)
: Expat needs to re-parse unfinished tokens; without the protection introduced in Expat 2.6.0, this can lead to quadratic runtime that can be used to cause denial of service in the application parsing XML. The issue is known as **CVE 2023-52425**.

21.5 xml.etree.ElementTree — ElementTree XML API

소스 코드: Lib/xml/etree/ElementTree.py

---

xml.etree.ElementTree 모듈은 XML 데이터를 구문 분석하고 만들기 위한 단순하고 효율적인 API를 구현합니다.

버전 3.3에서 변경: 이 모듈은 가능할 때마다 빠른 구현을 사용합니다.

버전 3.3부터 폐지됨: `xml.etree.cElementTree` 모듈은 폐지되었습니다.

> **참고**
>
> If you need to parse untrusted or unauthenticated data, see *XML security*.

21.5.1 자습서

이것은 *xml.etree.ElementTree*(줄여서 `ET`)를 사용하기 위한 간단한 자습서입니다. 목표는 모듈의 일부 빌딩 블록과 기본 개념을 예시하는 것입니다.

XML 트리와 엘리먼트

XML은 본질적으로 위계적(hierarchical) 데이터 형식이며, 이를 나타내는 가장 자연스러운 방법은 트리를 사용하는 것입니다. `ET`에는 이 목적을 위한 두 가지 클래스가 있습니다 - *ElementTree*는 전체 XML 문서를 트리로 나타내고, *Element*는 이 트리에 있는 단일 노드를 나타냅니다. 전체 문서와의 상호 작용(파일 읽기와 쓰기)은 일반적으로 *ElementTree* 수준에서 수행됩니다. 단일 XML 엘리먼트와 해당 서브 엘리먼트와의 상호 작용은 *Element* 수준에서 수행됩니다.

XML 구문 분석하기

이 섹션의 샘플 데이터로 가상의 `country_data.xml` XML 문서를 사용합니다:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

파일을 읽어서 이 데이터를 가져올 수 있습니다:

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

또는 문자열에서 직접:

```
root = ET.fromstring(country_data_as_string)
```

*fromstring()*은 문자열에서 *Element*로 XML을 직접 구문 분석하는데, 구문 분석된 트리의 루트 엘리먼트입니다. 다른 구문 분석 함수는 *ElementTree*를 만들 수 있습니다. 설명서를 확인하십시오.

*Element*로서, `root`에는 태그(tag)와 어트리뷰트 딕셔너리가 있습니다:

```
>>> root.tag
'data'
>>> root.attrib
{}
```

또한 우리가 이터레이트 할 수 있는 자식 노드가 있습니다:

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

자식은 중첩되며, 인덱스로 특정 자식 노드에 액세스 할 수 있습니다:

```
>>> root[0][1].text
'2008'
```

참고

XML 입력의 모든 엘리먼트가 구문 분석된 트리의 엘리먼트가 되는 것은 아닙니다. 현재, 이 모듈은 입력에서 XML 주석, 처리 명령(processing instructions) 및 문서 형 선언(document type declarations)을 건너뜁니다. 그런데도, XML 텍스트를 구문 분석하는 대신 이 모듈의 API를 사용하여 만들어진 트리에는 주석과 처리 명령이 있을 수 있습니다; 이들은 XML 출력을 생성할 때 포함됩니다. 사용자 정의 *TreeBuilder* 인스턴스를 *XMLParser* 생성자에 전달하여 문서 형 선언에 액세스 할 수 있습니다.

비 블로킹 구문 분석을 위한 풀(pull) API

이 모듈이 제공하는 대부분의 구문 분석 함수는 결과를 반환하기 전에 전체 문서를 한 번에 읽도록 요구합니다. *XMLParser*를 사용하고 점진적으로 데이터를 공급하는 것이 가능하지만, 콜백 대상에 메서드를 호출하는 푸시(push) API로, 대부분의 경우 너무 저 수준이고 불편합니다. 때로 사용자가 실제로 원하는 것은 완전히 구성된 *Element* 객체의 편리함을 즐기면서 연산을 블로킹하지 않고 XML을 점진적으로 구문 분석할 수 있는 것입니다.

이를 위한 가장 강력한 도구는 *XMLPullParser* 입니다. XML 데이터를 얻기 위해 블로킹 읽기가 필요하지 않으며, 대신 *XMLPullParser.feed()* 호출을 통해 점진적으로 데이터가 제공됩니다. 구문 분석된 XML 엘리먼트를 얻으려면, *XMLPullParser.read_events()*를 호출하십시오. 예를 들면 다음과 같습니다:

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
end
mytag text= sometext more text
```

명백한 사용 사례는 XML 데이터가 소켓에서 수신되거나 일부 저장 장치에서 점진적으로 읽히는 비 블로킹 방식으로 작동하는 응용 프로그램입니다. 이럴 때, 블로킹 읽기는 허용되지 않습니다.

*XMLPullParser*는 매우 유연하기 때문에 더 단순한 사용 사례에 사용하기 불편할 수 있습니다. 응용 프로그램이 XML 데이터 읽기를 블로킹해도 상관없지만, 여전히 점진적인 구문 분석 기능이 필요하면, *iterparse()*를 살펴보십시오. 큰 XML 문서를 읽을 때 메모리에 전체를 저장하지 않으려는 경우 유용할 수 있습니다.

Where *immediate* feedback through events is wanted, calling method *XMLPullParser.flush()* can help reduce delay; please make sure to study the related security notes.

흥미로운 엘리먼트 찾기

*Element*에는 그 아래의 모든 서브 트리(자식, 자식의 자식 등)를 재귀적으로 이터레이트 하는 데 도움을 주는 유용한 메서드가 있습니다. 예를 들어, *Element.iter()*:

```
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

*Element.findall()*은 현재 엘리먼트의 직접적인 자식인 태그가 있는 엘리먼트만 찾습니다. *Element.find()*는 특정 태그가 있는 첫 번째 자식을 찾고, *Element.text*는 엘리먼트의 텍스트 내용에 액세스합니다. *Element.get()*은 엘리먼트의 어트리뷰트에 액세스합니다:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

*XPath*를 사용하면 찾을 엘리먼트를 더 정교하게 지정할 수 있습니다.

XML 파일 수정하기

*ElementTree*는 XML 문서를 구축하고 파일에 쓰는 간단한 방법을 제공합니다. *ElementTree.write()* 메서드가 이 용도입니다.

일단 만들어지면, *Element* 객체는 필드(가령 *Element.text*)를 직접 변경하고, 어트리뷰트를 추가하고 수정(*Element.set()* 메서드)하는 것뿐만 아니라 새로운 자식을 추가하여 (예를 들어 *Element.append()*) 조작 할 수 있습니다.

각각의 국가(country)의 순위(rank)에 1을 더하고, rank 엘리먼트에 `updated` 어트리뷰트를 추가하고 싶다고 합시다:

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

우리의 XML은 이제 다음과 같습니다:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank updated="yes">69</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

*Element.remove()*를 사용하여 엘리먼트를 제거할 수 있습니다. rank가 50보다 큰 모든 국가를 제거하려고 한다고 합시다:

```
>>> for country in root.findall('country'):
...     # root.findall() 를 사용하여 순회 중 제거를 피합니다
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

이터레이션 하는 동안 동시 수정은 파이썬 리스트나 딕셔너리를 이터레이션 할 때와 마찬가지로 문제를 유발할 수 있음에 유의하십시오. 따라서, 이 예제에서는 먼저 `root.findall()`로 일치하는 모든 엘리먼트를 수집한 다음, 일치 항목 리스트를 이터레이트 합니다.

우리의 XML은 이제 다음과 같습니다:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```

XML 문서 구축하기

SubElement() 함수는 주어진 엘리먼트에 대해 새로운 서브 엘리먼트를 만드는 편리한 방법을 제공합니다:

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

이름 공간이 있는 XML 구문 분석하기

XML 입력에 이름 공간(namespaces)이 있으면, `prefix:sometag` 형식의 접두사가 있는 태그와 어트리뷰트는 `{uri}sometag`로 확장되는데, 여기서 *prefix*는 전체 *URI*로 대체됩니다. 또한 기본 이름 공간(default namespace)이 있으면, 그 전체 URI가 접두사가 없는 모든 태그 앞에 추가됩니다.

다음은 두 개의 이름 공간을 통합한 XML 예제입니다, 하나는 접두사가 "fictional" 이고 다른 하나는 기본 이름 공간으로 사용됩니다:

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
    <actor>
        <name>John Cleese</name>
        <fictional:character>Lancelot</fictional:character>
        <fictional:character>Archie Leach</fictional:character>
    </actor>
    <actor>
        <name>Eric Idle</name>
        <fictional:character>Sir Robin</fictional:character>
        <fictional:character>Gunther</fictional:character>
        <fictional:character>Commander Clement</fictional:character>
    </actor>
</actors>
```

이 XML 예제를 검색하고 탐색하는 한 가지 방법은 *find()*나 *findall()*의 xpath에 있는 모든 태그나 어트리뷰트에 URI를 수동으로 추가하는 것입니다:

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print(name.text)
    for char in actor.findall('{http://characters.example.com}character'):
        print(' |-->', char.text)
```

이름 공간이 있는 XML 예제를 검색하는 더 좋은 방법은 여러분 자신의 접두사가 담긴 딕셔너리를 만들고 검색 함수에서 사용하는 것입니다:

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print(name.text)
    for char in actor.findall('role:character', ns):
        print(' |-->', char.text)
```

이 두 가지 방법 모두 다음과 같이 출력합니다:

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

21.5.2 XPath 지원

이 모듈은 트리에서 엘리먼트를 찾기 위해 XPath 표현식을 제한적으로 지원합니다. 목표는 축약된 문법의 작은 부분 집합을 지원하는 것입니다; 완전한 XPath 엔진은 이 모듈의 범위를 벗어납니다.

예

다음은 모듈의 일부 XPath 기능을 보여주는 예입니다. *XML 구문 분석하기* 섹션의 `countrydata` XML 문서를 사용할 것입니다:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# 최상위 엘리먼트
root.findall(".")

# 최상위 엘리먼트의 모든 'country' 자식의 'neighbor' 손자
root.findall("./country/neighbor")

# 'year' 자식이 있고 name='Singapore'인 노드
root.findall(".//year/..[@name='Singapore']")

# name='Singapore'인 노드의 자식인 'year' 노드
root.findall(".//*[@name='Singapore']/year")

# 부모의 두 번째 자식인 모든 'neighbor' 노드
root.findall(".//neighbor[2]")
```

이름 공간이 있는 XML의 경우, 일반적인 정규화된 `{namespace}tag` 표기법을 사용하십시오:

```
# 문서의 모든 더블린 코어(dublin-core) "title" 태그
root.findall(".//{http://purl.org/dc/elements/1.1/}title")
```

지원되는 XPath 문법

문법	의미
`tag`	주어진 태그를 가진 모든 자식 엘리먼트를 선택합니다. 예를 들어, `spam`은 `spam`이라는 모든 자식 엘리먼트를 선택하고, `spam/egg`는 `spam`이라는 모든 자식의 `egg`라는 모든 손자를 선택합니다. `{namespace}*`는 지정된 이름 공간의 모든 태그를 선택하고, `{*}spam`은 모든 이름 공간의 (또는 이름 공간이 없는) `spam`이라는 태그를 선택하고, `{}*`은 이름 공간이 없는 태그만 선택합니다. 버전 3.8에서 변경: 애스터리스크 와일드카드 지원이 추가되었습니다.
`*`	주석과 처리 명령을 포함한 모든 자식 엘리먼트를 선택합니다. 예를 들어, `*/egg`는 `egg`라는 모든 손자를 선택합니다.
`.`	현재 노드를 선택합니다. 상대 경로임을 나타내기 위해 경로의 시작 부분에서 주로 유용합니다.
`//`	현재 엘리먼트 아래의 모든 수준에서 모든 서브 엘리먼트를 선택합니다. 예를 들어, `.//egg`는 전체 트리에서 모든 `egg` 엘리먼트를 선택합니다.
`..`	부모 엘리먼트를 선택합니다. 경로가 (엘리먼트 `find`가 호출된) 시작 엘리먼트의 조상에 도달하려고 하면 `None`을 반환합니다.
`[@attrib]`	주어진 어트리뷰트를 가진 모든 엘리먼트를 선택합니다.
`[@attrib='value']`	주어진 어트리뷰트가 주어진 값을 갖는 모든 엘리먼트를 선택합니다. 값은 따옴표를 포함할 수 없습니다.
`[@attrib!='value']`	주어진 어트리뷰트가 주어진 값을 갖지 않는 모든 엘리먼트를 선택합니다. 값은 따옴표를 포함할 수 없습니다. Added in version 3.10.
`[tag]`	`tag`라는 자식이 있는 모든 엘리먼트를 선택합니다. 직계 자식만 지원됩니다.
`[.='text']`	자손을 포함한 전체 텍스트 내용이 주어진 `text`와 같은 모든 엘리먼트를 선택합니다. Added in version 3.7.
`[.!='text']`	자손을 포함한 전체 텍스트 내용이 주어진 `text`와 다른 모든 엘리먼트를 선택합니다. Added in version 3.10.
`[tag='text']`	자손을 포함한 전체 텍스트 내용이 지정된 `text`와 같은 이름이 `tag`인 자식이 있는 모든 엘리먼트를 선택합니다.
`[tag!='text']`	자손을 포함한 전체 텍스트 내용이 지정된 `text`와 다른 이름이 `tag`인 자식이 있는 모든 엘리먼트를 선택합니다. Added in version 3.10.
`[position]`	주어진 위치(position)에 있는 모든 엘리먼트를 선택합니다. 위치(position)는 정수(1이 첫 번째 위치입니다), 표현식 `last()` (마지막 위치) 또는 마지막 위치에 상대적인 위치(예를 들어 `last()-1`)일 수 있습니다.

술어 (대괄호 안에 있는 표현식) 앞에는 태그 이름, 애스터리스크 또는 다른 술어가 와야 합니다. `position` 술어 앞에는 태그 이름이 와야 합니다.

21.5.3 레퍼런스

함수

`xml.etree.ElementTree.`**`canonicalize`**(*xml_data=None, *, out=None, from_file=None, **options*)

C14N 2.0 변환 함수.

규범화(canonicalization)는 바이트 단위 비교와 디지털 서명을 허용하는 방식으로 XML 출력을 정규화하는 방법입니다. XML 직렬화기가 갖는 자유도를 줄이고 대신 더 제한된 XML 표현을 생성합니다. 주요 제한 사항은 이름 공간 선언의 배치, 어트리뷰트의 순서 및 무시할 수 있는 공백입니다.

이 함수는 XML 데이터 문자열(*xml_data*)이나 파일 경로 또는 파일류 객체(*from_file*)를 입력으로 받아서, 규범적 형식으로 변환한 후, 제공된다면 *out* 파일(류) 객체를 사용하여 기록하고, 그렇지 않으면 텍스트 문자열로 반환합니다. 출력 파일은 바이트열이 아닌 텍스트를 받습니다. 따라서 `utf-8` 인코딩을 사용하여 텍스트 모드로 열어야 합니다.

일반적인 사용:

```
xml_data = "<root>...</root>"
print(canonicalize(xml_data))

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(xml_data, out=out_file)

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(from_file="inputfile.xml", out=out_file)
```

구성 *options*는 다음과 같습니다:

- *with_comments*: 주석을 포함하려면 참으로 설정합니다 (기본값: 거짓).
- ***strip_text*: 텍스트 내용 전후의 공백을 제거하려면 참으로 설정합니다**
  (기본값: 거짓)
- ***rewrite_prefixes*: 이름 공간 접두사를 "n{number}" 로 바꾸려면 참으로 설정합니다**
  (기본값: 거짓)
- ***qname_aware_tags*: 텍스트 내용에서 접두사를 대체해야 하는 qname 인식 태그 이름 집합**
  (기본값: 비어 있음)
- ***qname_aware_attrs*: 텍스트 내용에서 접두사를 대체해야 하는 qname 인식 어트리뷰트 이름 집합**
  (기본값: 비어 있음)
- *exclude_attrs*: 직렬화하면 안 되는 어트리뷰트 이름 집합
- *exclude_tags*: 직렬화하면 안 되는 태그 이름 집합

위의 옵션 목록에서, "집합"은 문자열의 모든 컬렉션이나 이터러블을 가리키며, 순서는 고려하지 않습니다.

Added in version 3.8.

`xml.etree.ElementTree.`**`Comment`**(*text=None*)

주석 엘리먼트 팩토리. 이 팩토리 함수는 표준 직렬화기가 XML 주석으로 직렬화할 특수 엘리먼트를 만듭니다. 주석 문자열은 바이트 문자열이나 유니코드 문자열일 수 있습니다. *text*는 주석 문자열이 포함된 문자열입니다. 주석을 나타내는 엘리먼트 인스턴스를 반환합니다.

`XMLParser`는 주석 객체를 만드는 대신 입력에서 주석을 건너뜀에 유의하십시오. `ElementTree`는 `Element` 메서드 중 하나를 사용하여 트리에 삽입된 주석 노드만 포함합니다.

`xml.etree.ElementTree.`**`dump`**(*elem*)

엘리먼트 트리나 엘리먼트 구조를 sys.stdout에 씁니다. 이 함수는 디버깅에만 사용해야 합니다.

정확한 출력 형식은 구현에 따라 다릅니다. 이 버전에서는, 일반 XML 파일로 기록됩니다.

*elem*은 엘리먼트 트리나 개별 엘리먼트입니다.

버전 3.8에서 변경: `dump()` 함수는 이제 사용자가 지정한 어트리뷰트 순서를 유지합니다.

`xml.etree.ElementTree.`**`fromstring`**(*text, parser=None*)

문자열 상수에서 XML 섹션을 구문 분석합니다. `XML()`과 같습니다. *text*는 XML 데이터를 포함하는 문자열입니다. *parser*는 선택적 구문 분석기 인스턴스입니다. 지정하지 않으면, 표준 `XMLParser` 구문 분석기가 사용됩니다. `Element` 인스턴스를 반환합니다.

`xml.etree.ElementTree.`**`fromstringlist`**(*sequence, parser=None*)

문자열 조각의 시퀀스에서 XML 문서를 구문 분석합니다. *sequence*는 XML 데이터 조각을 포함하는 리스트나 다른 시퀀스입니다. *parser*는 선택적 구문 분석기 인스턴스입니다. 지정하지 않으면 표준 `XMLParser` 구문 분석기가 사용됩니다. `Element` 인스턴스를 반환합니다.

Added in version 3.2.

xml.etree.ElementTree.**indent**(*tree*, *space=' '*, *level=0*)

트리를 시각적으로 들여쓰기하기 위해 서브 트리에 공백을 추가합니다. 이것은 예쁘게 인쇄된 XML 출력을 생성하는 데 사용될 수 있습니다. *tree*는 Element나 ElementTree 일 수 있습니다. *space*는 각 들여쓰기 수준에 삽입되는 공백 문자열이며 기본적으로 두 개의 스페이스 문자입니다. 이미 들여쓰기된 트리 내부에서 부분 서브 트리를 들여 쓰려면, 초기 들여쓰기 수준을 *level*로 전달하십시오.

Added in version 3.9.

xml.etree.ElementTree.**iselement**(*element*)

객체가 유효한 엘리먼트 객체로 보이는지 확인합니다. *element*는 엘리먼트 인스턴스입니다. 이것이 엘리먼트 객체이면 `True`를 반환합니다.

xml.etree.ElementTree.**iterparse**(*source*, *events=None*, *parser=None*)

XML 섹션을 엘리먼트 트리로 점진적으로 구문 분석하고, 사용자에게 진행 중인 작업을 보고합니다. *source*는 파일명이나 XML 데이터를 포함하는 파일 객체입니다. *events*는 보고할 이벤트의 시퀀스입니다. 지원되는 이벤트는 문자열 `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` 및 `"end-ns"`입니다 ("ns" 이벤트는 자세한 이름 공간 정보를 얻는 데 사용됩니다). *events*를 생략하면, `"end"` 이벤트만 보고됩니다. *parser*는 선택적 구문 분석기 인스턴스입니다. 지정하지 않으면 표준 `XMLParser` 구문 분석기가 사용됩니다. *parser*는 `XMLParser`의 서브 클래스여야 하며 기본 `TreeBuilder` 만 대상으로 사용할 수 있습니다. `(event, elem)` 쌍을 제공하는 이터레이터를 반환합니다; *source*가 완전히 읽힌 후 결과 XML 트리의 루트 엘리먼트를 참조하는 `root` 어트리뷰트가 있습니다. 이터레이터는 *source*가 파일명일 때, 내부 파일 객체를 닫는 `close()` 메서드를 제공합니다.

`iterparse()`는 점진적으로 트리를 구축하지만, *source*(또는 그 이름의 파일)에 대한 블로킹 읽기를 유발함에 유의하십시오. 따라서, 블로킹 읽기를 할 수 없는 응용 프로그램에는 적합하지 않습니다. 완전한 비 블로킹 구문 분석에 대해서는 `XMLPullParser`를 참조하십시오.

> **참고**
>
> `iterparse()`는 "start" 이벤트를 방출할 때 시작 태그의 ">" 문자를 보았다는 것만 보장해서, 어트리뷰트는 정의되지만, 텍스트의 내용과 테일(tail) 어트리뷰트는 그 시점에 정의되지 않습니다. 자식 엘리먼트에도 마찬가지로 적용됩니다; 그들은 존재할 수도 그렇지 않을 수도 있습니다.
>
> 완전히 채워진 엘리먼트가 필요하면, 대신 "end" 이벤트를 찾으십시오.

버전 3.4부터 폐지됨: *parser* 인자.

버전 3.8에서 변경: `comment`와 `pi` 이벤트가 추가되었습니다.

버전 3.13에서 변경: Added the `close()` method.

xml.etree.ElementTree.**parse**(*source*, *parser=None*)

XML 섹션을 엘리먼트 트리로 구문 분석합니다. *source*는 XML 데이터를 포함하는 파일명이나 파일 객체입니다. *parser*는 선택적 구문 분석기 인스턴스입니다. 지정하지 않으면 표준 `XMLParser` 구문 분석기가 사용됩니다. `ElementTree` 인스턴스를 반환합니다.

xml.etree.ElementTree.**ProcessingInstruction**(*target*, *text=None*)

PI 엘리먼트 팩토리. 이 팩토리 함수는 XML 처리 명령으로 직렬화될 특수 엘리먼트를 만듭니다. *target*은 PI 대상을 포함하는 문자열입니다. 주어지면, *text*는 PI 내용을 포함하는 문자열입니다. 처리 명령을 나타내는 엘리먼트 인스턴스를 반환합니다.

`XMLParser`는 PI 객체를 작성하는 대신 입력에서 처리 명령을 건너뜀에 유의하십시오. `ElementTree`는 `Element` 메서드 중 하나를 사용하여 트리에 삽입된 처리 명령 노드만 포함합니다.

xml.etree.ElementTree.**register_namespace**(*prefix*, *uri*)

이름 공간 접두사를 등록합니다. 레지스트리는 전역적이며, 지정된 접두사나 이름 공간 URI에 대한 기존 매핑이 제거됩니다. *prefix*는 이름 공간 접두사입니다. *uri*는 이름 공간 URI입니다. 이 이름 공간의 태그와 어트리뷰트는 가능하다면 주어진 접두사로 직렬화됩니다.

Added in version 3.2.

xml.etree.ElementTree.**SubElement**(*parent*, *tag*, *attrib*={}, ***extra*)

서브 엘리먼트 팩토리. 이 함수는 엘리먼트 인스턴스를 만들어 기존 엘리먼트에 추가합니다.

엘리먼트 이름, 어트리뷰트 이름 및 어트리뷰트 값은 바이트 문자열이나 유니코드 문자열일 수 있습니다. *parent*는 부모 엘리먼트입니다. *tag*는 서브 엘리먼트 이름입니다. *attrib*는 엘리먼트 어트리뷰트를 포함하는 선택적 딕셔너리입니다. *extra*에는 키워드 인자로 지정된 추가 어트리뷰트가 포함됩니다. 엘리먼트 인스턴스를 반환합니다.

xml.etree.ElementTree.**tostring**(*element*, *encoding='us-ascii'*, *method='xml'*, *, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

모든 서브 엘리먼트를 포함하는, XML 엘리먼트의 문자열 표현을 생성합니다. *element*는 `Element` 인스턴스입니다. *encoding*[1] 은 출력 인코딩입니다 (기본값은 US-ASCII 입니다). `encoding="unicode"` 를 사용하여 유니코드 문자열을 생성하십시오 (그렇지 않으면 바이트 문자열이 생성됩니다). *method* 는 `"xml"`, `"html"` 또는 `"text"` 입니다 (기본값은 `"xml"` 입니다). *xml_declaration*, *default_namespace* 및 *short_empty_elements*는 `ElementTree.write()` 에서와 같은 의미입니다. XML 데이터를 포함하는 (선택적으로) 인코딩된 문자열을 반환합니다.

버전 3.4에서 변경: *short_empty_elements* 매개 변수를 추가했습니다.

버전 3.8에서 변경: *xml_declaration*과 *default_namespace* 매개 변수를 추가했습니다.

버전 3.8에서 변경: `tostring()` 함수는 이제 사용자가 지정한 어트리뷰트 순서를 유지합니다.

xml.etree.ElementTree.**tostringlist**(*element*, *encoding='us-ascii'*, *method='xml'*, *, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

모든 서브 엘리먼트를 포함하는, XML 엘리먼트의 문자열 표현을 생성합니다. *element*는 `Element` 인스턴스입니다. *encoding*[1] 은 출력 인코딩입니다 (기본값은 US-ASCII 입니다). `encoding="unicode"` 를 사용하여 유니코드 문자열을 생성하십시오 (그렇지 않으면 바이트 문자열이 생성됩니다). *method* 는 `"xml"`, `"html"` 또는 `"text"` 입니다 (기본값은 `"xml"` 입니다). *xml_declaration*, *default_namespace* 및 *short_empty_elements*는 `ElementTree.write()` 에서와 같은 의미입니다. XML 데이터가 포함된 (선택적으로) 인코딩된 문자열의 리스트를 반환합니다. `b"".join(tostringlist(element)) == tostring(element)` 라는 사실을 제외하고는, 특정 시퀀스를 보장하지는 않습니다.

Added in version 3.2.

버전 3.4에서 변경: *short_empty_elements* 매개 변수를 추가했습니다.

버전 3.8에서 변경: *xml_declaration*과 *default_namespace* 매개 변수를 추가했습니다.

버전 3.8에서 변경: `tostringlist()` 함수는 이제 사용자가 지정한 어트리뷰트 순서를 유지합니다.

xml.etree.ElementTree.**XML**(*text*, *parser=None*)

문자열 상수에서 XML 섹션을 구문 분석합니다. 이 함수는 "XML 리터럴" 을 파이썬 코드에 내장시키는 데 사용할 수 있습니다. *text*는 XML 데이터를 포함하는 문자열입니다. *parser*는 선택적 구문 분석기 인스턴스입니다. 지정하지 않으면 표준 `XMLParser` 구문 분석기가 사용됩니다. `Element` 인스턴스를 반환합니다.

xml.etree.ElementTree.**XMLID**(*text*, *parser=None*)

문자열 상수에서 XML 섹션을 구문 분석하고, 엘리먼트 id:s 를 엘리먼트로 매핑하는 딕셔너리도 반환합니다. *text*는 XML 데이터를 포함하는 문자열입니다. *parser*는 선택적 구문 분석기 인스턴스입니다. 지정하지 않으면 표준 `XMLParser` 구문 분석기가 사용됩니다. `Element` 인스턴스와 딕셔너리를 포함하는 튜플을 반환합니다.

21.5.4 XInclude 지원

이 모듈은 `xml.etree.ElementInclude` 도우미 모듈을 통해 XInclude 지시어를 제한적으로 지원합니다. 이 모듈은 트리의 정보를 기반으로, 서브 트리와 텍스트 문자열을 엘리먼트 트리에 삽입하는 데 사용할 수 있습니다.

[1] XML 출력에 포함된 인코딩 문자열은 적절한 표준을 준수해야 합니다. 예를 들어, "UTF-8" 은 유효하지만, "UTF8" 은 유효하지 않습니다. https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl and https://www.iana.org/assignments/character-sets/character-sets.xhtml 을 참조하십시오.

예

다음은 XInclude 모듈의 사용법을 보여주는 예입니다. 현재 문서에 XML 문서를 포함 시키려면, `{http://www.w3.org/2001/XInclude}include` 엘리먼트를 사용하고 **parse** 어트리뷰트를 `"xml"`로 설정하고, **href** 어트리뷰트를 사용하여 포함할 문서를 지정하십시오.

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="source.xml" parse="xml" />
</document>
```

기본적으로, **href** 어트리뷰트는 파일 이름으로 취급됩니다. 사용자 정의 로더를 사용하여 이 동작을 대체할 수 있습니다. 또한 표준 도우미는 XPointer 문법을 지원하지 않음에 유의하십시오.

이 파일을 처리하려면, 평소와 같이 로드하고, 루트 엘리먼트를 *xml.etree.ElementTree* 모듈에 전달하십시오:

```
from xml.etree import ElementTree, ElementInclude

tree = ElementTree.parse("document.xml")
root = tree.getroot()

ElementInclude.include(root)
```

ElementInclude 모듈은 `{http://www.w3.org/2001/XInclude}include` 엘리먼트를 **source.xml** 문서의 루트 엘리먼트로 바꿉니다. 결과는 다음과 같습니다:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <para>This is a paragraph.</para>
</document>
```

parse 어트리뷰트가 생략되면, 기본값은 "xml" 입니다. href 어트리뷰트는 필수입니다.

텍스트 문서를 포함 시키려면, `{http://www.w3.org/2001/XInclude}include` 엘리먼트를 사용하고, **parse** 어트리뷰트를 "text" 로 설정하십시오:

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) <xi:include href="year.txt" parse="text" />.
</document>
```

결과는 다음과 같습니다:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) 2003.
</document>
```

21.5.5 레퍼런스

함수

`xml.etree.ElementInclude.`**`default_loader`**(*href*, *parse*, *encoding=None*)

기본 로더. 이 기본 로더는 디스크에서 포함되는 리소스를 읽습니다. *href*는 URL 입니다. *parse*는 구문 분석 모드로 "xml" 이나 "text" 입니다. *encoding*은 선택적 텍스트 인코딩입니다. 지정하지 않으면, 인코딩은 `utf-8` 입니다. 확장된 리소스를 반환합니다. 구문 분석 모드가 `"xml"` 이면, *Element* 인스턴스입니다. 구문 분석 모드가 `"text"` 이면, 문자열입니다. 로더가 실패하면, `None`을 반환하거나 예외를 발생시킬 수 있습니다.

`xml.etree.ElementInclude.`**`include`**(*elem*, *loader=None*, *base_url=None*, *max_depth=6*)

이 함수는 *elem*이 가리키는 트리 내에서 XInclude 지시어를 제자리에서(in-place) 확장합니다. *elem*은

루트 *Element*이거나 그런 엘리먼트를 찾기 위한 *ElementTree* 인스턴스입니다. *loader*는 선택적 리소스 로더입니다. 생략하면, 기본값은 *default_loader()* 입니다. 주어지면, *default_loader()*와 같은 인터페이스를 구현하는 콜러블 이어야 합니다. *base_url*은 상대적인 포함 파일 참조를 결정하기 위한 원본 파일의 베이스 URL입니다. *max_depth*는 최대 재귀 포함 수입니다. 악의적인 내용 폭발의 위험을 줄이도록 제한되었습니다. 제한을 비활성화하려면 `None`을 전달하십시오.

버전 3.9에서 변경: *base_url*과 *max_depth* 매개 변수를 추가했습니다.

Element 객체

class `xml.etree.ElementTree.`**`Element`**(*tag*, *attrib*={}, ***extra*)

Element 클래스. 이 클래스는 Element 인터페이스를 정의하고, 이 인터페이스의 참조 구현을 제공합니다.

엘리먼트 이름, 어트리뷰트 이름 및 어트리뷰트 값은 바이트 문자열이나 유니코드 문자열일 수 있습니다. *tag*는 엘리먼트 이름입니다. *attrib*는 엘리먼트 어트리뷰트를 포함하는 선택적 딕셔너리입니다. *extra*에는 키워드 인자로 지정된 추가 어트리뷰트가 포함되어 있습니다.

tag

이 엘리먼트가 나타내는 데이터 종류(즉, 엘리먼트 유형)를 식별하는 문자열.

text

tail

이 어트리뷰트는 엘리먼트와 연관된 추가 데이터를 담는 데 사용될 수 있습니다. 해당 값은 일반적으로 문자열이지만 임의의 응용 프로그램별 객체일 수 있습니다. 엘리먼트가 XML 파일에서 만들어지면, *text* 어트리뷰트는 엘리먼트의 시작 태그와 첫 번째 자식이나 종료 태그 사이의 텍스트를 담거나 `None`이고, *tail* 어트리뷰트는 엘리먼트의 종료 태그와 다음 태그 사이의 텍스트를 담거나 `None`입니다. 다음과 같은 XML 데이터의 경우

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

a 엘리먼트는 *text*와 *tail* 어트리뷰트 모두에 대해 `None`을 갖고, *b* 엘리먼트는 *text* `"1"`과 *tail* `"4"`를 갖고, *c* 엘리먼트는 *text* `"2"`와 *tail* `None`을 갖고, *d* 엘리먼트는 *text* `None`과 *tail* `"3"`을 갖습니다.

엘리먼트의 내부 텍스트를 수집하려면, *itertext()*를 참조하십시오, 예를 들어 `"".join(element.itertext())`.

응용 프로그램은 이 어트리뷰트들에 임의의 객체를 저장할 수 있습니다.

attrib

엘리먼트의 어트리뷰트를 포함하는 딕셔너리. *attrib* 값은 항상 진짜 가변 파이썬 딕셔너리이지만, ElementTree 구현은 다른 내부 표현을 사용하도록 선택하고, 누군가가 요청할 때만 딕셔너리를 만들 수 있습니다. 이러한 구현의 이점을 활용하려면, 가능한 한 아래의 딕셔너리 메서드를 사용하십시오.

다음과 같은 딕셔너리와 유사한 메서드가 엘리먼트 어트리뷰트에서 작동합니다.

clear()

엘리먼트를 재설정합니다. 이 함수는 모든 서브 엘리먼트를 제거하고, 모든 어트리뷰트를 지우고, text 및 tail 어트리뷰트를 `None`으로 설정합니다.

get(*key*, *default=None*)

*key*라는 이름의 엘리먼트 어트리뷰트를 가져옵니다.

어트리뷰트 값을 반환하거나, 어트리뷰트를 찾을 수 없으면 *default*를 반환합니다.

items()

엘리먼트 어트리뷰트를 (이름, 값) 쌍의 시퀀스로 반환합니다. 어트리뷰트는 임의의 순서로 반환됩니다.

keys()

엘리먼트 어트리뷰트 이름을 리스트로 반환합니다. 이름은 임의의 순서로 반환됩니다.

set(*key*, *value*)

엘리먼트의 *key* 어트리뷰트를 *value*로 설정합니다.

다음 메서드는 엘리먼트의 자식(서브 엘리먼트)에서 작동합니다.

append(*subelement*)

이 엘리먼트의 내부 서브 엘리먼트 리스트 끝에 엘리먼트 *subelement*를 추가합니다. *subelement*가 `Element`가 아니면 `TypeError`를 발생시킵니다.

extend(*subelements*)

엘리먼트의 이터러블에서 *subelements*를 추가합니다. 서브 엘리먼트가 `Element`가 아니면 `TypeError`를 발생시킵니다.

Added in version 3.2.

find(*match*, *namespaces=None*)

*match*와 일치하는 첫 번째 서브 엘리먼트를 찾습니다. *match*는 태그 이름이나 경로일 수 있습니다. 엘리먼트 인스턴스나 `None`을 반환합니다. *namespaces*는 이름 공간 접두사에서 전체 이름으로의 선택적 매핑입니다. 표현식에서 접두사가 없는 모든 태그 이름을 지정된 이름 공간으로 이동하려면 `''`를 접두사로 전달하십시오.

findall(*match*, *namespaces=None*)

태그 이름이나 경로로 일치하는 모든 서브 엘리먼트를 찾습니다. 일치하는 모든 엘리먼트가 문서 순서로 포함된 리스트를 반환합니다. *namespaces*는 이름 공간 접두사에서 전체 이름으로의 선택적 매핑입니다. 표현식에서 접두사가 없는 모든 태그 이름을 지정된 이름 공간으로 이동하려면 `''`를 접두사로 전달하십시오.

findtext(*match*, *default=None*, *namespaces=None*)

*match*와 일치하는 첫 번째 서브 엘리먼트의 텍스트를 찾습니다. *match*는 태그 이름이나 경로일 수 있습니다. 일치하는 첫 번째 엘리먼트의 텍스트 내용을 반환하거나, 엘리먼트가 없으면 *default*를 반환합니다. 일치하는 엘리먼트에 텍스트 내용이 없으면 빈 문자열이 반환됨에 유의하십시오. *namespaces*는 이름 공간 접두사에서 전체 이름으로의 선택적 매핑입니다. 표현식에서 접두사가 없는 모든 태그 이름을 지정된 이름 공간으로 이동하려면 `''`를 접두사로 전달하십시오.

insert(*index*, *subelement*)

이 엘리먼트의 지정된 위치에 *subelement*를 삽입합니다. *subelement*가 `Element`가 아니면 `TypeError`를 발생시킵니다.

iter(*tag=None*)

현재 엘리먼트를 루트로 하여 트리 이터레이터를 만듭니다. 이터레이터는 이 엘리먼트와 그 아래의 모든 엘리먼트를 문서 순서로 (깊이 우선) 이터레이트 합니다. *tag*가 `None`이나 `'*'`가 아니면, 태그가 *tag*와 같은 엘리먼트만 이터레이터에서 반환됩니다. 이터레이션 중에 트리 구조가 수정되면, 결과는 정의되지 않습니다.

Added in version 3.2.

iterfind(*match*, *namespaces=None*)

태그 이름이나 경로로 일치하는 모든 서브 엘리먼트를 찾습니다. 일치하는 모든 엘리먼트를 문서 순서로 산출하는 이터러블을 반환합니다. *namespaces*는 이름 공간 접두사에서 전체 이름으로의 선택적 매핑입니다.

Added in version 3.2.

itertext()

텍스트 이터레이터를 만듭니다. 이터레이터는 이 엘리먼트와 모든 서브 엘리먼트를 문서 순서대로 루핑하고, 모든 내부 텍스트를 반환합니다.

Added in version 3.2.

makeelement(*tag*, *attrib*)

이 엘리먼트와 같은 유형의 새 엘리먼트 객체를 만듭니다. 이 메서드를 호출하지 말고, 대신 `SubElement()` 팩토리 함수를 사용하십시오.

remove(*subelement*)

엘리먼트에서 *subelement*를 제거합니다. find* 메서드와 달리 이 메서드는 태그값이나 내용이 아닌 인스턴스 아이덴티티를 기준으로 엘리먼트를 비교합니다.

Element 객체는 서브 엘리먼트 작업을 위한 `__delitem__()`, `__getitem__()`, `__setitem__()`, `__len__()` 시퀀스 형 메서드도 지원합니다.

주의: 서브 엘리먼트가 없는 엘리먼트는 `False`로 테스트 됩니다. 미래에 출시되는 파이썬에서, 모든 엘리먼트는 서브 엘리먼트가 존재하든 안 하든 관계없이 `True`로 테스트될 것입니다. 대신, 명시적인 `len(elem)` 이나 `elem is not None` 테스트를 사용하십시오.:

```
element = root.find('foo')

if not element:  # 주의!
    print("element not found, or element has no subelements")

if element is None:
    print("element not found")
```

버전 3.12에서 변경: Testing the truth value of an Element emits *DeprecationWarning*.

파이썬 3.8 이전에는, 어트리뷰트를 이름으로 정렬하여 엘리먼트의 XML 어트리뷰트의 직렬화 순서를 인위적으로 예측할 수 있도록 했습니다. 이제 보장되는 딕셔너리 순서를 기반으로, 이 임의 재정렬은 파이썬 3.8에서 제거되어 어트리뷰트가 원래 구문 분석되거나 사용자 코드에 의해 만들어진 순서를 유지합니다.

일반적으로, 사용자 코드는 XML Information Set이 정보 전달에서 어트리뷰트 순서를 명시적으로 제외한다는 점에서 구체적인 어트리뷰트 순서에 의존하지 않아야 합니다. 입력의 모든 순서를 다룰 수 있도록 코드를 준비해야 합니다. 예를 들어 암호화 서명이나 테스트 데이터 집합과 같이 결정론적 XML 출력이 필요한 경우, *canonicalize()* 함수로 규범적 직렬화를 사용할 수 있습니다.

규범적 출력이 적용되지는 않지만, 출력에서 특정 어트리뷰트 순서가 여전히 필요하면, 코드는 코드를 읽는 사람의 지각 불일치를 피하고자, 원하는 순서로 어트리뷰트를 직접 만드는 것을 목표로 해야 합니다. 이를 달성하기 어려운 경우, Element 생성과 독립적으로 순서를 강제하기 위해 직렬화 전에 다음과 같은 조리법을 적용 할 수 있습니다:

```
def reorder_attributes(root):
    for el in root.iter():
        attrib = el.attrib
        if len(attrib) > 1:
            # 어트리뷰트 순서를 조정합니다, 예를 들어 정렬로
            attribs = sorted(attrib.items())
            attrib.clear()
            attrib.update(attribs)
```

ElementTree 객체

class xml.etree.ElementTree.**ElementTree**(*element=None*, *file=None*)

ElementTree 래퍼 클래스. 이 클래스는 전체 엘리먼트 위계를 나타내며, 표준 XML과의 직렬화에 대한 추가 지원을 추가합니다.

*element*는 루트 엘리먼트입니다. 주어지면 XML *file*의 내용으로 트리가 초기화됩니다.

_setroot(*element*)

이 트리의 루트 엘리먼트를 교체합니다. 이것은 트리의 현재 내용을 버리고, 주어진 엘리먼트로 대체합니다. 주의해서 사용하십시오. *element*는 엘리먼트 인스턴스입니다.

find(*match*, *namespaces=None*)

*Element.find()*와 같습니다, 트리의 루트에서 시작합니다.

findall(*match*, *namespaces=None*)

`Element.findall()`과 같습니다, 트리의 루트에서 시작합니다.

findtext(*match*, *default=None*, *namespaces=None*)

`Element.findtext()`와 같습니다, 트리의 루트에서 시작합니다.

getroot()

이 트리의 루트 엘리먼트를 반환합니다.

iter(*tag=None*)

루트 엘리먼트에 대한 트리 이터레이터를 만들고 반환합니다. 이터레이터는 이 트리의 모든 엘리먼트를 섹션 순서대로 루핑합니다. *tag*는 찾을 태그입니다 (기본값은 모든 엘리먼트를 반환하는 것입니다).

iterfind(*match*, *namespaces=None*)

`Element.iterfind()`와 같습니다, 트리의 루트에서 시작합니다.

Added in version 3.2.

parse(*source*, *parser=None*)

이 엘리먼트 트리에 외부 XML 섹션을 로드합니다. *source*는 파일 이름이나 파일 객체입니다. *parser*는 선택적 구문 분석기 인스턴스입니다. 지정하지 않으면 표준 `XMLParser` 구문 분석기가 사용됩니다. 섹션 루트 엘리먼트를 반환합니다.

write(*file*, *encoding='us-ascii'*, *xml_declaration=None*, *default_namespace=None*, *method='xml'*, **, short_empty_elements=True*)

엘리먼트 트리를 XML로 파일에 씁니다. *file*은 파일 이름이거나 쓰기 위해 열린 파일 객체입니다. *encoding*[Page 1417, 1] 은 출력 인코딩입니다 (기본값은 US-ASCII 입니다). *xml_declaration*은 파일에 XML 선언을 추가해야 하는지를 제어합니다. 추가하지 말아야 하면 `False`, 항상 추가하면 `True`, US-ASCII나 UTF-8이나 유니코드가 아닐 때만 추가하면 `None`을 사용하십시오 (기본값은 `None` 입니다). *default_namespace*는 기본 XML 이름 공간을 설정합니다 ("xmlns"). *method*는 `"xml"`, `"html"` 또는 `"text"` 입니다 (기본값은 `"xml"` 입니다). 키워드 전용 *short_empty_elements* 매개 변수는 내용이 없는 엘리먼트의 포매팅을 제어합니다. `True`(기본값)이면, 단일 스스로 닫힌 태그로 방출되고, 그렇지 않으면 한 쌍의 시작/종료 태그로 방출됩니다.

출력은 문자열(`str`)이나 바이너리(`bytes`)입니다. 이것은 *encoding* 인자에 의해 제어됩니다. *encoding*이 `"unicode"`이면, 출력은 문자열입니다; 그렇지 않으면 바이너리입니다. *file*이 열린 파일 객체이면 *file*의 유형과 충돌할 수 있음에 유의하십시오; 문자열을 바이너리 스트림에 쓰거나 그 반대로 하지 않도록 하십시오.

버전 3.4에서 변경: *short_empty_elements* 매개 변수를 추가했습니다.

버전 3.8에서 변경: `write()` 메서드는 이제 사용자가 지정한 어트리뷰트 순서를 유지합니다.

이것은 조작될 XML 파일입니다:

```
<html>
    <head>
        <title>Example page</title>
    </head>
    <body>
        <p>Moved to <a href="http://example.org/">example.org</a>
        or <a href="http://example.com/">example.com</a>.</p>
    </body>
</html>
```

첫 번째 문단에 있는 모든 링크의 어트리뷰트 "target"을 변경하는 예:

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")     # body에서 처음 등장하는 태그 p를 찾습니다
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))   # 모든 링크의 리스트를 반환합니다
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:             # 발견된 모든 링크를 이터레이트 합니다
...     i.attrib["target"] = "blank"
...
>>> tree.write("output.xhtml")
```

QName 객체

class xml.etree.ElementTree.**QName**(*text_or_uri*, *tag=None*)

QName 래퍼. 출력에서 이름 공간을 올바르게 처리하기 위해, QName 어트리뷰트 값을 래핑하는 데 사용할 수 있습니다. *text_or_uri*는 {uri}local 형식으로 QName 값을 포함하는 문자열이거나, tag 인자가 제공되면 QName의 URI 부분입니다. *tag*가 제공되면, 첫 번째 인자는 URI로 해석되고, 이 인자는 로컬 이름으로 해석됩니다. `QName` 인스턴스는 불투명합니다.

TreeBuilder 객체

class xml.etree.ElementTree.**TreeBuilder**(*element_factory=None*, **, comment_factory=None*, *pi_factory=None*, *insert_comments=False*, *insert_pis=False*)

일반 엘리먼트 구조 빌더. 이 빌더는 일련의 start, data, end, comment 및 pi 메서드 호출을 올바른 형식의 엘리먼트 구조로 변환합니다. 이 클래스를 사용하여 사용자 정의 XML 구문 분석기를 사용하거나 다른 XML 형식의 구문 분석기를 사용하는 엘리먼트 구조를 구축할 수 있습니다.

주어지면 *element_factory*는 두 개의 위치 인자를 받아들이는 콜러블 이어야 합니다: 태그와 어트리뷰트 딕셔너리. 새로운 엘리먼트 인스턴스를 반환할 것으로 기대합니다.

주어지면, *comment_factory*와 *pi_factory* 함수는 주석과 처리 명령을 만들기 위해 `Comment()`와 `ProcessingInstruction()` 함수처럼 동작해야 합니다. 지정하지 않으면, 기본 팩토리가 사용됩니다. *insert_comments* 및/또는 *insert_pis*가 참이면, 주석/처리 명령이 루트 엘리먼트 내에 있으면 (하지만 외부에 있지 않으면) 트리에 삽입됩니다.

close()

빌더 버퍼를 플러시하고, 최상위 문서 엘리먼트를 반환합니다. `Element` 인스턴스를 반환합니다.

data(*data*)

현재 엘리먼트에 텍스트를 추가합니다. *data*는 문자열입니다. 바이트 문자열이거나 유니코드 문자열이어야 합니다.

end(*tag*)

현재 엘리먼트를 닫습니다. *tag*는 엘리먼트 이름입니다. 닫힌 엘리먼트를 반환합니다.

start(*tag*, *attrs*)

새로운 엘리먼트를 엽니다. *tag*는 엘리먼트 이름입니다. *attrs*는 엘리먼트 어트리뷰트를 포함하는 딕셔너리입니다. 열린 엘리먼트를 반환합니다.

comment(*text*)

주어진 *text*로 주석을 만듭니다. `insert_comments`가 참이면, 트리에 추가합니다.

Added in version 3.8.

pi(*target*, *text*)

주어진 *target* 이름과 *text*로 처리 명령을 만듭니다. `insert_pis`가 참이면, 트리에 추가합니다.

Added in version 3.8.

또한, 사용자 정의 `TreeBuilder` 객체는 다음 메서드를 제공할 수 있습니다:

doctype(*name*, *pubid*, *system*)

doctype 선언을 처리합니다. *name*은 doctype 이름입니다. *pubid*는 공개 식별자입니다. *system*은 시스템 식별자입니다. 이 메서드는 기본 `TreeBuilder` 클래스에 없습니다.

Added in version 3.2.

start_ns(*prefix*, *uri*)

구문 분석기가 새 이름 공간 선언을 발견할 때마다 이를 정의하는 여는 엘리먼트에 대한 `start()` 콜백 전에 호출됩니다. *prefix*는 기본 이름 공간의 경우 `''`이고, 그렇지 않으면 선언된 이름 공간 접두사 이름입니다. *uri*는 이름 공간 URI입니다.

Added in version 3.8.

end_ns(*prefix*)

이름 공간 접두사 매핑을 선언한 엘리먼트의 `end()` 콜백 후에, 스코프를 벗어난 *prefix*의 이름으로 호출됩니다.

Added in version 3.8.

class xml.etree.ElementTree.**C14NWriterTarget**(*write*, **, with_comments=False, strip_text=False, rewrite_prefixes=False, qname_aware_tags=None, qname_aware_attrs=None, exclude_attrs=None, exclude_tags=None*)

C14N 2.0 기록기. 인자는 `canonicalize()` 함수와 같습니다. 이 클래스는 트리를 구축하지 않지만, *write* 함수를 사용하여 콜백 이벤트를 직렬화된 형식으로 직접 변환합니다.

Added in version 3.8.

XMLParser 객체

class xml.etree.ElementTree.**XMLParser**(**, target=None, encoding=None*)

이 클래스는 모듈의 저수준 빌딩 블록입니다. 효율적인 이벤트 기반 XML 구문 분석을 위해 `xml.parsers.expat`을 사용합니다. `feed()` 메서드를 사용하여 XML 데이터를 점진적으로 공급할 수 있으며, 구문 분석 이벤트는 *target* 객체에서 콜백을 호출하여 푸시 API로 변환됩니다. *target*을 생략하면, 표준 `TreeBuilder`가 사용됩니다. *encoding*[Page 1417, 1] 이 제공되면, 값이 XML 파일에 지정된 인코딩을 대체합니다.

버전 3.8에서 변경: 매개 변수는 이제 키워드-전용입니다. *html* 인자는 더는 지원되지 않습니다.

close()

구문 분석기로의 데이터 공급을 완료합니다. 생성 중에 전달된 *target*의 `close()` 메서드를 호출한 결과를 반환합니다; 기본적으로, 최상위 문서 엘리먼트입니다.

feed(*data*)

구문 분석기에 데이터를 공급합니다. *data*는 인코딩된 데이터입니다.

flush()

Triggers parsing of any previously fed unparsed data, which can be used to ensure more immediate feedback, in particular with Expat >=2.6.0. The implementation of `flush()` temporarily disables reparse deferral with Expat (if currently enabled) and triggers a reparse. Disabling reparse deferral has security consequences; please see `xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()` for details.

Note that `flush()` has been backported to some prior releases of CPython as a security fix. Check for availability of `flush()` using `hasattr()` if used in code running across a variety of Python versions.

Added in version 3.13.

`XMLParser.feed()`는 각 여는 태그마다 *target*의 `start(tag, attrs_dict)` 메서드를 호출하고, 닫는 태그마다 `end(tag)` 메서드를 호출하며, 데이터는 메서드 `data(data)`로 처리됩니다. 추가로 지원되는 콜백 메서드는, `TreeBuilder` 클래스를 참조하십시오. `XMLParser.close()`는 *target*의

메서드 `close()`를 호출합니다. *XMLParser*는 트리 구조 구축에만 사용할 수 있는 것은 아닙니다. 다음은 XML 파일의 최대 깊이를 계산하는 예입니다:

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # 구문 분석기의 대상 객체
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # 여는 태그마다 호출됩니다.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # 닫는 태그마다 호출됩니다.
...         self.depth -= 1
...     def data(self, data):
...         pass            # 우리는 데이터로 아무것도 할 필요가 없습니다.
...     def close(self):    # 모든 데이터가 구문 분석되었을 때 호출됩니다.
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

XMLPullParser 객체

class xml.etree.ElementTree.**XMLPullParser**(*events=None*)

비 블로킹 응용 프로그램에 적합한 풀 구문 분석기. 입력 측 API는 *XMLParser*의 것과 유사하지만, 콜백 대상으로 호출을 푸시하는 대신, *XMLPullParser*는 내부 구문 분석 이벤트 리스트를 수집하여 사용자가 여기서 읽을 수 있도록 합니다. *events*는 보고할 이벤트의 시퀀스입니다. 지원되는 이벤트는 문자열 `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` 및 `"end-ns"` 입니다 (“ns” 이벤트는 자세한 이름 공간 정보를 얻는 데 사용됩니다). *events*를 생략하면, `"end"` 이벤트만 보고됩니다.

feed(*data*)

주어진 바이트열 데이터를 구문 분석기에 공급합니다.

flush()

Triggers parsing of any previously fed unparsed data, which can be used to ensure more immediate feedback, in particular with Expat >=2.6.0. The implementation of *flush()* temporarily disables reparse deferral with Expat (if currently enabled) and triggers a reparse. Disabling reparse deferral has security consequences; please see *xml.parsers.expat.xmlparser.SetReparseDeferralEnabled()* for details.

Note that *flush()* has been backported to some prior releases of CPython as a security fix. Check for availability of *flush()* using *hasattr()* if used in code running across a variety of Python versions.

Added in version 3.13.

close()

구문 분석기에 데이터 스트림이 종료되었음을 알립니다. *XMLParser.close()*와 달리, 이 메서드는 항상 *None*을 반환합니다. 구문 분석기가 닫힐 때 아직 꺼내지 않은 이벤트는 *read_events()*로 계속 읽을 수 있습니다.

read_events()

구문 분석기에 공급된 데이터에서 발생한 이벤트에 대한 이터레이터를 반환합니다. 이터레이터는 (event, elem) 쌍을 산출합니다, 여기서 *event*는 이벤트 유형을 나타내는 문자열(예를 들어 "end")이고 *elem*은 발견된 *Element* 객체나, 다음과 같은 기타 문맥 값입니다.

- start, end: 현재 엘리먼트.
- comment, pi: 현재 주석 / 처리 명령
- start-ns: 선언된 이름 공간 매핑을 명명하는 튜플 (prefix, uri)
- end-ns: *None* (향후 버전에서 변경될 수 있습니다)

*read_events()*에 대한 이전 호출에서 제공된 이벤트는 다시 산출되지 않습니다. 이벤트는 이터레이터에서 꺼낼 때만 내부 큐에서 소비되므로, *read_events()*에서 얻은 이터레이터에 대해 병렬로 이터레이션 하는 여러 판독기는 예측할 수 없는 결과를 얻게 됩니다.

> **참고**
>
> *XMLPullParser*는 "start" 이벤트를 방출할 때 시작 태그의 ">" 문자를 보았다는 것만 보장해서, 어트리뷰트는 정의되지만, 텍스트의 내용과 테일(tail) 어트리뷰트는 그 시점에 정의되지 않습니다. 자식 엘리먼트에도 마찬가지로 적용됩니다; 그들은 존재할 수도 그렇지 않을 수도 있습니다.
>
> 완전히 채워진 엘리먼트가 필요하면, 대신 "end" 이벤트를 찾으십시오.

Added in version 3.4.

버전 3.8에서 변경: comment와 pi 이벤트가 추가되었습니다.

예외

class xml.etree.ElementTree.**ParseError**

XML 구문 분석 에러, 이 모듈의 다양한 구문 분석 메서드가 구문 분석이 실패할 때 발생시킵니다. 이 예외 인스턴스의 문자열 표현에는 사용자 친화적인 에러 메시지가 포함됩니다. 또한, 다음과 같은 어트리뷰트를 사용할 수 있습니다:

code

expat 구문 분석기의 숫자 에러 코드. 에러 코드와 의미의 목록은 *xml.parsers.expat* 설명서를 참조하십시오.

position

에러가 발생한 위치를 지정하는, *line*, *column* 숫자의 튜플.

21.6 xml.dom — 문서 객체 모델 API

소스 코드: Lib/xml/dom/__init__.py

문서 객체 모델(Document Object Model), 또는 "DOM"은 XML 문서를 액세스하고 수정하기 위한 W3C(World Wide Web Consortium)의 교차 언어 API입니다. DOM 구현은 XML 문서를 트리 구조로 나타내거나, 클라이언트 코드가 이러한 구조를 처음부터 구축 할 수 있도록 합니다. 그런 다음 잘 알려진 인터페이스를 제공하는 객체 집합을 통해 구조에 액세스 할 수 있습니다.

DOM은 무작위 액세스 응용 프로그램에 매우 유용합니다. SAX에서는 한 번에 한 조각의 문서만 볼 수 있습니다. 하나의 SAX 요소를 보고 있는 동안, 다른 SAX 요소에 액세스할 수 없습니다. 텍스트 노드를 보고 있으면, 이것을 포함하는 요소에 액세스할 수 없습니다. SAX 응용 프로그램을 작성할 때는, 문서에서의

프로그램의 위치를 자신의 코드 어딘가에서 추적해야 합니다. SAX는 여러분을 위해 대신해주지 않습니다. 또한, XML 문서를 미리 보아야 한다면, 운이 다했다고 보아야 합니다.

트리에 액세스할 수 없는 이벤트 중심 모델에서는 일부 응용 프로그램이 불가능합니다. 물론 SAX 이벤트에서 트리를 직접 만들 수는 있지만, DOM을 사용하면 그런 코드를 작성하지 않아도 됩니다. DOM은 XML 데이터의 표준 트리 표현입니다.

문서 객체 모델은 W3C에 의해 단계적으로 또는 그들의 용어로는 "수준" 으로 정의됩니다. API의 파이썬 매핑은 실질적으로 DOM 수준 2 권장 사항을 기반으로 합니다.

DOM 응용 프로그램은 일반적으로 일부 XML을 DOM으로 구문 분석하는 것으로 시작합니다. 이것을 달성하는 방법은 DOM 수준 1은 전혀 다루지 않으며, 수준 2는 제한된 개선만을 제공합니다: `Document` 생성 메서드에 대한 액세스를 제공하는 `DOMImplementation` 객체 클래스가 있습니다만, 구현에 독립적인 방법으로 XML 판독기(reader)/기록기(writer)/Document 구축기(builder)를 액세스하는 방법이 없습니다. 기존 `Document` 객체 없이 이러한 메서드에 액세스할 수 있는 잘 정의된 방법도 없습니다. 파이썬에서, 각 DOM 구현은 *getDOMImplementation()* 함수를 제공합니다. DOM 수준 3은 판독기(reader)에 대한 인터페이스를 정의하는 로드/저장 명세를 추가하지만, 아직 파이썬 표준 라이브러리에서는 사용할 수 없습니다.

일단 DOM 문서 객체가 있으면, 프로퍼티와 메서드를 통해 XML 문서의 일부에 액세스 할 수 있습니다. 이러한 프로퍼티는 DOM 명세에 정의되어 있습니다; 레퍼런스 설명서의 이 부분은 파이썬이 명세를 해석하는 방법을 설명합니다.

W3C에서 제공하는 명세는 Java, ECMAScript 및 OMG IDL 용 DOM API를 정의합니다. 여기에 정의된 파이썬 매핑은 대부분 명세의 IDL 버전을 기반으로 하지만, 엄격한 준수는 필요하지 않습니다 (구현은 IDL의 엄격한 매핑을 자유롭게 지원할 수 있습니다). 매핑 요구 사항에 대한 자세한 내용은 규격 준수 절을 참조하십시오.

더 보기

Document Object Model (DOM) Level 2 Specification
파이썬 DOM API의 기반이 되는 W3C 권장 사항.

Document Object Model (DOM) Level 1 Specification
*xml.dom.minidom*이 지원하는 DOM에 대한 W3C 권장 사항.

Python Language Mapping Specification
OMG IDL에서 파이썬으로의 매핑을 지정합니다.

21.6.1 모듈 내용

*xml.dom*에는 다음 함수가 포함되어 있습니다:

`xml.dom.`**`registerDOMImplementation`**(*name*, *factory*)

factory 함수를 *name*이라는 이름으로 등록합니다. 팩토리(factory) 함수는 `DOMImplementation` 인터페이스를 구현하는 객체를 반환해야 합니다. 팩토리 함수는 호출 때마다 같은 객체를 반환하거나 특정 구현에 적합하다면 새 객체를 반환할 수 있습니다 (예를 들어, 해당 구현이 일부 사용자 정의를 지원하는 경우).

`xml.dom.`**`getDOMImplementation`**(*name=None*, *features=()*)

적절한 DOM 구현을 반환합니다. *name*은 잘 알려진 DOM 구현의 모듈 이름이거나, `None`입니다. `None`이 아니면, 해당 모듈을 임포트하고 성공하면 `DOMImplementation` 객체를 반환합니다. 이름이 지정되지 않고, 환경 변수 `PYTHON_DOM`이 설정되면, 이 변수를 구현을 찾는 데 사용합니다.

이름을 지정하지 않으면, 사용 가능한 구현을 검사하여 필요한 기능 집합이 있는 구현을 찾습니다. 구현을 찾을 수 없으면, *ImportError*를 발생시킵니다. 기능 목록은 사용 가능한 `DOMImplementation` 객체의 `hasFeature()` 메서드로 전달되는 `(feature, version)` 쌍의 시퀀스여야 합니다.

몇 가지 편의 상수도 제공됩니다:

`xml.dom.`**`EMPTY_NAMESPACE`**

이름 공간이 DOM의 노드와 연결되어 있지 않음을 나타내는 데 사용되는 값. 이것은 일반적으로

노드의 namespaceURI에서 발견되거나, 이름 공간별 메서드에 대한 *namespaceURI* 매개 변수로 사용됩니다.

xml.dom.**XML_NAMESPACE**

예약된 접두어 xml과 연관된 이름 공간 URI, Namespaces in XML에서 정의됩니다 (4 절).

xml.dom.**XMLNS_NAMESPACE**

이름 공간 선언의 이름 공간 URI, Document Object Model (DOM) Level 2 Core Specification에서 정의됩니다 (1.1.8 절).

xml.dom.**XHTML_NAMESPACE**

XHTML 이름 공간의 URI, XHTML 1.0: The Extensible HyperText Markup Language에서 정의됩니다 (3.1.1 절).

또한, *xml.dom*에는 베이스 Node 클래스와 DOM 예외 클래스가 포함됩니다. 이 모듈에서 제공하는 Node 클래스는 DOM 명세에 정의된 메서드나 어트리뷰트를 구현하지 않습니다; 구상(concrete) DOM 구현이 이를 제공해야 합니다. 이 모듈의 일부로 제공되는 Node 클래스는 구상 Node 객체의 nodeType 어트리뷰트에 사용되는 상수를 제공합니다; 그것들은 DOM 명세를 준수하기 위해 모듈 수준이 아니라 클래스 내에 위치합니다.

21.6.2 DOM의 객체

DOM에 대한 결정적인 문서는 W3C의 DOM 명세입니다.

DOM 어트리뷰트는 간단한 문자열 대신 노드로 조작될 수도 있음에 유의하십시오. 하지만 그렇게 해야만 하는 경우는 매우 드물어서, 이 사용법은 아직 문서화되지 않았습니다.

인터페이스	절	목적
DOMImplementation	*DOMImplementation 객체*	하부 구현에 대한 인터페이스.
Node	*Node 객체*	문서에 있는 대부분 객체에 대한 베이스 인터페이스.
NodeList	*NodeList 객체*	노드의 시퀀스를 위한 인터페이스.
DocumentType	*DocumentType 객체*	문서를 처리하는 데 필요한 선언에 대한 정보.
Document	*Document 객체*	전체 문서를 나타내는 객체.
Element	*Element 객체*	문서 계층의 엘리먼트 노드.
Attr	*Attr 객체*	엘리먼트 노드의 어트리뷰트 값 노드
Comment	*Comment 객체*	소스 문서에 있는 주석의 표현.
Text	*Text와 CDATASection 객체*	문서의 텍스트 내용을 포함하는 노드.
ProcessingInstructior	*ProcessingInstruction 객체*	처리 명령어 표현.

추가 절에서 파이썬에서 DOM 작업을 위해 정의된 예외에 관해 설명합니다.

DOMImplementation 객체

DOMImplementation 인터페이스는 응용 프로그램이 사용 중인 DOM에서 특정 기능의 가용성을 판별할 방법을 제공합니다. DOM 수준 2는 DOMImplementation을 사용하여 새로운 Document와 DocumentType 객체를 만드는 기능을 추가했습니다.

DOMImplementation.**hasFeature**(*feature*, *version*)

문자열 *feature*와 *version* 쌍으로 식별되는 기능이 구현되었으면 True를 반환합니다.

DOMImplementation.**createDocument**(*namespaceUri*, *qualifiedName*, *doctype*)

지정된 *namespaceUri*와 *qualifiedName*을 가진 자식 Element 객체를 포함하는 새 Document 객체(DOM의 루트)를 반환합니다. *doctype*은 *createDocumentType()*으로 만든 DocumentType 객체거나 None이어야 합니다. 파이썬 DOM API에서, Element 자식이 만들어지지 않음을 표시하기 위해 처음 두 인자는 None일 수 있습니다.

DOMImplementation.**createDocumentType**(*qualifiedName*, *publicId*, *systemId*)

XML 문서 형 선언에 포함된 정보를 나타내는, 지정된 *qualifiedName*, *publicId* 및 *systemId* 문자열을 캡슐화하는 새 DocumentType 객체를 반환합니다.

Node 객체

XML 문서의 모든 구성 요소는 Node의 서브 클래스입니다.

Node.**nodeType**

노드 형을 나타내는 정수. 형의 기호 상수는 Node 객체에 있습니다: ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE, CDATA_SECTION_NODE, ENTITY_NODE, PROCESSING_INSTRUCTION_NODE, COMMENT_NODE, DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE. 이것은 읽기 전용 어트리뷰트입니다.

Node.**parentNode**

현재 노드의 부모, 또는 문서 노드의 경우 None. 값은 항상 Node 객체나 None입니다. Element 노드의 경우, 이것은 부모 엘리먼트가 되는데, 루트 엘리먼트는 예외로, 이때는 Document 객체가 됩니다. Attr 노드의 경우, 항상 None입니다. 이것은 읽기 전용 어트리뷰트입니다.

Node.**attributes**

어트리뷰트 객체의 NamedNodeMap. 엘리먼트에만 실제 값이 있습니다; 다른 것은 이 어트리뷰트에서 None을 제공합니다. 이것은 읽기 전용 어트리뷰트입니다.

Node.**previousSibling**

같은 부모를 갖고 이 노드 바로 앞에 있는 노드. 예를 들어 *self* 엘리먼트의 시작 태그 바로 앞에 오는 종료 태그의 엘리먼트. 물론, XML 문서는 단순히 엘리먼트만으로 구성되지 않기 때문에, 이전 형제 (previous sibling)는 텍스트, 주석 또는 뭔가 다른 것이 될 수 있습니다. 이 노드가 부모의 첫 번째 자식이면, 이 어트리뷰트는 None입니다. 이것은 읽기 전용 어트리뷰트입니다.

Node.**nextSibling**

같은 부모를 갖고 이 노드 바로 뒤에 나오는 노드. *previousSibling*도 참조하십시오. 이것이 부모의 마지막 자식이면, 이 어트리뷰트는 None입니다. 이것은 읽기 전용 어트리뷰트입니다.

Node.**childNodes**

이 노드에 포함된 노드의 리스트. 이것은 읽기 전용 어트리뷰트입니다.

Node.**firstChild**

노드의 첫 번째 자식 (있다면), 또는 None. 이것은 읽기 전용 어트리뷰트입니다.

Node.**lastChild**

노드의 마지막 자식 (있다면), 또는 None. 이것은 읽기 전용 어트리뷰트입니다.

Node.**localName**

콜론이 있으면 그 뒤에 오는 tagName의 부분, 그렇지 않으면 전체 tagName. 값은 문자열입니다.

Node.**prefix**

콜론이 있으면 그 앞에 오는 tagName의 부분, 그렇지 않으면 빈 문자열. 값은 문자열이나 None입니다.

Node.**namespaceURI**

엘리먼트 이름과 연관된 이름 공간. 이것은 문자열이나 None입니다. 이것은 읽기 전용 어트리뷰트입니다.

Node.**nodeName**

이는 각 노드 형마다 다른 의미입니다; 자세한 내용은 DOM 명세를 참조하십시오. 여기서 얻는 정보를 항상 다른 프로퍼티에서 얻을 수 있습니다, 가령 엘리먼트의 tagName 프로퍼티나 어트리뷰트의 name 프로퍼티. 모든 노드 형에서, 이 어트리뷰트의 값은 문자열이나 None입니다. 이것은 읽기 전용 어트리뷰트입니다.

Node.**nodeValue**

이는 각 노드 형마다 다른 의미입니다; 자세한 내용은 DOM 명세를 참조하십시오. 상황은 *nodeName*과 유사합니다. 값은 문자열이나 None입니다.

Node.**hasAttributes**()

노드에 어트리뷰트가 있으면 True를 반환합니다.

Node.**hasChildNodes**()

노드에 자식 노드가 있으면 True를 반환합니다.

Node.**isSameNode**(*other*)

*other*가 이 노드와 같은 노드를 가리키면 True를 반환합니다. 이것은 모든 종류의 프락시 구조를 사용하는 DOM 구현에서 특히 유용합니다 (여러 객체가 같은 노드를 가리킬 수 있기 때문입니다).

> **참고**
>
> 이것은 여전히 "작업 초안" 단계에 있는 제안된 DOM 수준 3 API를 기반으로 하지만, 이 특정 인터페이스는 논란의 여지가 없는 것으로 보입니다. W3C의 변경 사항이 파이썬 DOM 인터페이스에서 이 메서드에 반드시 영향을 미치는 것은 아닙니다 (이를 위한 새 W3C API도 지원되기는 하겠지만).

Node.**appendChild**(*newChild*)

자식 리스트의 끝에 이 노드의 새 자식 노드를 추가하고, *newChild*를 반환합니다. 노드가 이미 트리에 있으면, 먼저 제거됩니다.

Node.**insertBefore**(*newChild*, *refChild*)

기존 자식 앞에 새 자식 노드를 삽입합니다. *refChild*가 이 노드의 자식이어야 합니다; 그렇지 않으면 *ValueError*가 발생합니다. *newChild*가 반환됩니다. *refChild*가 None이면, 자식 리스트의 끝에 *newChild*를 삽입합니다.

Node.**removeChild**(*oldChild*)

자식 노드를 제거합니다. *oldChild*는 이 노드의 자식이어야 합니다; 그렇지 않으면, *ValueError*가 발생합니다. 성공하면 *oldChild*가 반환됩니다. *oldChild*가 더는 사용되지 않으면, 그것의 unlink() 메서드를 호출해야 합니다.

Node.**replaceChild**(*newChild*, *oldChild*)

기존 노드를 새 노드로 교체합니다. *oldChild*는 이 노드의 자식이어야 합니다. 그렇지 않으면, *ValueError*가 발생합니다.

Node.**normalize**()

모든 텍스트 스트레치(stretch)가 단일 Text 인스턴스로 저장되도록, 인접한 텍스트 노드를 결합합니다. 이는 많은 응용 프로그램에서 DOM 트리의 텍스트 처리를 단순화합니다.

Node.**cloneNode**(*deep*)

이 노드를 복제합니다. *deep*을 설정하면 모든 자식 노드도 복제됩니다. 복제를 반환합니다.

NodeList 객체

NodeList는 노드의 시퀀스를 나타냅니다. 이러한 객체는 DOM Core 권장 사항에서 두 가지 방식으로 사용됩니다: Element 객체는 자식 노드의 리스트를 제공하고, Node의 getElementsByTagName() 과 getElementsByTagNameNS() 메서드는 이 인터페이스를 사용하여 조회 결과를 나타냅니다.

DOM 수준 2 권장 사항은 이러한 객체에 대해 하나의 메서드와 하나의 어트리뷰트를 정의합니다:

NodeList.**item**(*i*)

시퀀스의 *i* 번째 항목(있다면)이나 None을 반환합니다. 인덱스 *i*는 0보다 작거나 시퀀스의 길이보다 크거나 같을 수 없습니다.

NodeList.**length**

시퀀스의 노드 수.

또한, 파이썬 DOM 인터페이스는 NodeList 객체를 파이썬 시퀀스로 사용할 수 있도록 몇 가지 추가 지원을 요구합니다. 모든 NodeList 구현에는 __len__() 과 __getitem__() 에 대한 지원이 포함되어야 합니다; 이를 통해 for 문에서 NodeList를 이터레이트할 수 있고, *len()* 내장 함수를 적절히 지원할 수 있습니다.

DOM 구현이 문서 수정을 지원하면, NodeList 구현은 __setitem__() 과 __delitem__() 메서드도 지원해야 합니다.

DocumentType 객체

문서에 의해 선언된 표기법과 엔티티에 대한 정보(구문 분석기가 사용하고 정보를 제공할 수 있으면 외부 부분 집합(external subset)을 포함하는)는 DocumentType 객체에서 사용 가능합니다. 문서의 DocumentType 은 Document 객체의 doctype 어트리뷰트에서 사용 가능합니다; 문서에 DOCTYPE 선언이 없으면, 문서의 doctype 어트리뷰트는 이 인터페이스의 인스턴스 대신 None으로 설정됩니다.

DocumentType은 Node의 특수화(specialization)이고 다음 어트리뷰트를 추가합니다:

DocumentType.**publicId**

문서 형 정의의 외부 부분 집합(external subset)에 대한 공용 식별자. 문자열이나 None 입니다.

DocumentType.**systemId**

문서 형 정의의 외부 부분 집합(external subset)에 대한 시스템 식별자. 문자열로 표현된 URI이거나 None 입니다.

DocumentType.**internalSubset**

문서에서 완전한 내부 부분 집합(internal subset)을 제공하는 문자열. 부분 집합을 묶는 대괄호는 포함되지 않습니다. 문서에 내부 부분 집합이 없으면 None이어야 합니다.

DocumentType.**name**

DOCTYPE 선언에 제공된 루트 엘리먼트의 이름 (있다면).

DocumentType.**entities**

외부 엔티티의 정의를 제공하는 NamedNodeMap 입니다. 엔티티 이름이 두 번 이상 정의되면, 첫 번째 정의 만 제공됩니다 (XML 권장 사항에 따라 다른 정의는 무시됩니다). 구문 분석기가 정보를 제공하지 않거나 정의된 엔티티가 없으면 None 일 수 있습니다.

DocumentType.**notations**

표기법의 정의를 제공하는 NamedNodeMap 입니다. 표기법 이름이 두 번 이상 정의되면, 첫 번째 정의 만 제공됩니다 (XML 권장 사항에 따라 다른 정의는 무시됩니다). 구문 분석기가 정보를 제공하지 않거나 정의된 표기법이 없으면 None 일 수 있습니다.

Document 객체

Document는 구성 엘리먼트, 어트리뷰트, 처리 명령어, 주석 등을 포함한 전체 XML 문서를 나타냅니다. Node의 프로퍼티를 상속한다는 점을 기억하십시오.

Document.**documentElement**

문서의 유일한 루트 엘리먼트.

Document.**createElement**(*tagName*)

새 엘리먼트 노드를 만들고 반환합니다. 엘리먼트는 만들어질 때 문서에 삽입되지 않습니다. insertBefore() 나 appendChild() 와 같은 다른 메서드 중 하나를 사용하여 명시적으로 삽입해야 합니다.

Document.**createElementNS**(*namespaceURI, tagName*)

이름 공간을 사용하여 새 엘리먼트를 만들고 반환합니다. *tagName*은 접두사를 가질 수 있습니다. 엘리먼트는 만들어질 때 문서에 삽입되지 않습니다. insertBefore() 나 appendChild() 와 같은 다른 메서드 중 하나를 사용하여 명시적으로 삽입해야 합니다.

Document.**createTextNode**(*data*)

매개 변수로 전달된 data를 포함하는 텍스트 노드를 만들고 반환합니다. 다른 생성 메서드와 마찬가지로 이 메서드는 노드를 트리에 삽입하지 않습니다.

Document.**createComment**(*data*)

매개 변수로 전달된 data를 포함하는 주석 노드를 만들고 반환합니다. 다른 생성 메서드와 마찬가지로 이 메서드는 노드를 트리에 삽입하지 않습니다.

Document.**createProcessingInstruction**(*target*, *data*)

매개 변수로 전달된 *target*과 *data*를 포함하는 처리 명령어 노드를 만들고 반환합니다. 다른 생성 메서드와 마찬가지로 이 메서드는 노드를 트리에 삽입하지 않습니다.

Document.**createAttribute**(*name*)

어트리뷰트 노드를 만들고 반환합니다. 이 메서드는 어트리뷰틀 노드를 특정 엘리먼트와 연관시키지 않습니다. 새로 만들어진 어트리뷰트 인스턴스를 사용하려면 적절한 `Element` 객체에서 `setAttributeNode()`를 사용해야 합니다.

Document.**createAttributeNS**(*namespaceURI*, *qualifiedName*)

이름 공간을 사용하여 어트리뷰트 노드를 만들고 반환합니다. *tagName*은 접두사를 가질 수 있습니다. 이 메서드는 어트리뷰트 노드를 특정 엘리먼트와 연관시키지 않습니다. 새로 만들어진 어트리뷰트 인스턴스를 사용하려면 적절한 `Element` 객체에서 `setAttributeNode()`를 사용해야합니다.

Document.**getElementsByTagName**(*tagName*)

특정 엘리먼트 형 이름을 가진 모든 자손(직계 자식, 자식의 자식 등)을 검색합니다.

Document.**getElementsByTagNameNS**(*namespaceURI*, *localName*)

특정 이름 공간 URI와 지역 이름(localname)을 사용하여 모든 자손(직계 자식, 자식의 자식 등)을 검색합니다. 지역 이름은 접두사 다음에 나오는 이름 공간의 일부입니다.

Element 객체

`Element`는 `Node`의 서브 클래스이므로, 모든 어트리뷰트를 상속합니다.

Element.**tagName**

엘리먼트 형 이름. 이름 공간을 사용하는 문서에서는 콜론을 포함할 수 있습니다. 값은 문자열입니다.

Element.**getElementsByTagName**(*tagName*)

`Document` 클래스의 동등한 메서드와 같습니다.

Element.**getElementsByTagNameNS**(*namespaceURI*, *localName*)

`Document` 클래스의 동등한 메서드와 같습니다.

Element.**hasAttribute**(*name*)

엘리먼트에 *name*이라는 이름의 어트리뷰트가 있으면 `True`를 반환합니다.

Element.**hasAttributeNS**(*namespaceURI*, *localName*)

엘리먼트에 *namespaceURI*와 *localName*으로 이름이 지정된 어트리뷰트가 있으면 `True`를 반환합니다.

Element.**getAttribute**(*name*)

*name*이라는 이름의 어트리뷰트의 값을 문자열로 반환합니다. 그러한 어트리뷰트가 없으면, 어트리뷰트에 값이 없는 것처럼 빈 문자열이 반환됩니다.

Element.**getAttributeNode**(*attrname*)

*attrname*이라는 이름의 어트리뷰트에 대한 `Attr` 노드를 반환합니다.

Element.**getAttributeNS**(*namespaceURI*, *localName*)

*namespaceURI*와 *localName*으로 이름이 지정된 어트리뷰트의 값을 문자열로 반환합니다. 그러한 어트리뷰트가 없으면, 어트리뷰트에 값이 없는 것처럼 빈 문자열이 반환됩니다.

Element.**getAttributeNodeNS**(*namespaceURI*, *localName*)

*namespaceURI*와 *localName*으로 주어진 어트리뷰트의 값을 노드로 반환합니다.

Element.**removeAttribute**(*name*)

이름(name)으로 어트리뷰트를 제거합니다. 일치하는 어트리뷰트가 없으면 `NotFoundErr`가 발생합니다.

Element.**removeAttributeNode**(*oldAttr*)

존재하면, 어트리뷰트 목록에서 *oldAttr*를 제거하고 반환합니다. *oldAttr*가 없으면 `NotFoundErr`가 발생합니다.

Element.**removeAttributeNS**(*namespaceURI*, *localName*)

이름으로 어트리뷰트를 제거합니다. qname이 아닌 localName을 사용함에 유의하십시오. 일치하는 어트리뷰트가 없어도 예외가 발생하지 않습니다.

Element.**setAttribute**(*name*, *value*)

문자열로 어트리뷰트 값을 설정합니다.

Element.**setAttributeNode**(*newAttr*)

엘리먼트에 새 어트리뷰트 노드를 추가합니다. name 어트리뷰트가 일치할 때 필요하면 기존 어트리뷰트를 대체합니다. 대체가 발생하면, 이전 어트리뷰트 노드가 반환됩니다. *newAttr*가 이미 사용 중이면 InuseAttributeErr가 발생합니다.

Element.**setAttributeNodeNS**(*newAttr*)

엘리먼트에 새 어트리뷰트 노드를 추가합니다. namespaceURI와 localName 어트리뷰트가 일치할 때 필요하면 기존 어트리뷰트를 대체합니다. 대체가 발생하면 이전 어트리뷰트 노드가 반환됩니다. *newAttr*가 이미 사용 중이면 InuseAttributeErr가 발생합니다.

Element.**setAttributeNS**(*namespaceURI*, *qname*, *value*)

문자열로 *namespaceURI*와 *qname*으로 지정된 어트리뷰트 값을 설정합니다. qname은 전체 어트리뷰트 이름임에 유의하십시오. 이것은 위와 다릅니다.

Attr 객체

Attr은 Node를 상속하므로, 모든 어트리뷰트를 상속합니다.

Attr.**name**

어트리뷰트 이름. 이름 공간을 사용하는 문서에서는 콜론을 포함할 수 있습니다.

Attr.**localName**

콜론이 있으면 그 뒤에 오는 이름의 일부, 그렇지 않으면 전체 이름. 이것은 읽기 전용 어트리뷰트입니다.

Attr.**prefix**

콜론이 있으면 그 앞에 오는 이름의 일부, 그렇지 않으면 빈 문자열.

Attr.**value**

어트리뷰트의 텍스트 값. 이것은 nodeValue 어트리뷰트의 동의어입니다.

NamedNodeMap 객체

NamedNodeMap은 Node를 상속하지 않습니다.

NamedNodeMap.**length**

어트리뷰트 목록의 길이입니다.

NamedNodeMap.**item**(*index*)

특정 인덱스에 있는 어트리뷰트를 반환합니다. 어트리뷰트를 얻는 순서는 임의적이지만 DOM 수명 동안 일관적입니다. 각 항목은 어트리뷰트 노드입니다. value 어트리뷰트로 값을 얻으십시오.

이 클래스에 더 많은 매핑 동작을 제공하는 실험적인 메서드도 있습니다. 이를 사용하거나 Element 객체에서 표준화된 getAttribute*() 메서드 집합을 사용할 수 있습니다.

Comment 객체

Comment는 XML 문서의 주석을 나타냅니다. Node의 서브 클래스이지만, 자식 노드를 가질 수 없습니다.

Comment.**data**

주석의 내용을 제공하는 문자열. 이 어트리뷰트는 선행 <!--와 후행 --> 사이의 모든 문자를 포함하지만, 이들을 포함하지는 않습니다.

Text와 CDATASection 객체

`Text` 인터페이스는 XML 문서의 텍스트를 나타냅니다. 구문 분석기와 DOM 구현이 DOM의 XML 확장을 지원하면, CDATA로 표시된 섹션으로 묶인 텍스트 부분은 `CDATASection` 객체에 저장됩니다. 이 두 인터페이스는 같지만, `nodeType` 어트리뷰트에서 다른 값을 제공합니다.

이 인터페이스는 `Node` 인터페이스를 확장합니다. 자식 노드를 가질 수 없습니다.

`Text.`**`data`**

문자열로 표현된 텍스트 노드의 내용.

> **참고**
>
> `CDATASection` 노드의 사용이 그 노드가 완전한 CDATA 표시 섹션을 표현한다고 나타내는 것은 아닙니다, 단지 노드의 내용이 CDATA 섹션의 일부임을 나타낼 뿐입니다. 단일 CDATA 섹션은 문서 트리에서 둘 이상의 노드로 표현될 수 있습니다. 인접한 두 개의 `CDATASection` 노드가 다른 CDATA 표시 섹션을 나타내는지를 확인할 방법은 없습니다.

ProcessingInstruction 객체

XML 문서의 처리 명령어를 나타냅니다; 이것은 `Node` 인터페이스를 상속하며 자식 노드를 가질 수 없습니다.

`ProcessingInstruction.`**`target`**

첫 번째 공백 문자까지의 처리 명령어의 내용. 이것은 읽기 전용 어트리뷰트입니다.

`ProcessingInstruction.`**`data`**

첫 번째 공백 문자 다음에 오는 처리 명령어의 내용.

예외

DOM 수준 2 권장 사항은 단일 예외 *`DOMException`*과 응용 프로그램이 어떤 종류의 에러가 발생했는지 판별하도록 하는 여러 상수를 정의합니다. *`DOMException`* 인스턴스에는 구체적인 예외에 적절한 값을 제공하는 *`code`* 어트리뷰트가 있습니다.

파이썬 DOM 인터페이스는 상수를 제공하지만, 동시에 예외 집합을 확장하여 DOM에 의해 정의된 각 예외 코드마다 구체적인 예외가 존재하도록 합니다. 구현 시 적절한 구체적인 예외를 발생시켜야 하며, 각 예외는 *`code`* 속성으로 적절한 값을 제공합니다.

`exception` `xml.dom.`**`DOMException`**

모든 구체적인 DOM 예외에 사용되는 베이스 예외 클래스. 이 예외 클래스는 직접 인스턴스 화할 수 없습니다.

`exception` `xml.dom.`**`DomstringSizeErr`**

지정된 텍스트 범위가 문자열에 맞지 않을 때 발생합니다. 이것은 파이썬 DOM 구현에서 사용되는 것으로 알려지지 않았지만, 파이썬으로 작성되지 않은 DOM 구현에서 수신될 수 있습니다.

`exception` `xml.dom.`**`HierarchyRequestErr`**

노드 형이 허용하지 않는 곳에 노드를 삽입하려고 할 때 발생합니다.

`exception` `xml.dom.`**`IndexSizeErr`**

메서드의 인덱스(index)나 크기(size) 매개 변수가 음수이거나 허용된 값을 초과할 때 발생합니다.

`exception` `xml.dom.`**`InuseAttributeErr`**

문서의 다른 곳에 이미 존재하는 `Attr` 노드를 삽입하려고 할 때 발생합니다.

`exception` `xml.dom.`**`InvalidAccessErr`**

하부 객체에서 매개 변수나 연산이 지원되지 않으면 발생합니다.

exception xml.dom.**InvalidCharacterErr**

이 예외는 문자열 매개 변수에 XML 1.0 권장 사항에서 사용 중인 컨텍스트에서 허용되지 않는 문자가 포함될 때 발생합니다. 예를 들어, 엘리먼트 형 이름에 스페이스가 있는 `Element` 노드를 만들려고 하면 이 에러가 발생합니다.

exception xml.dom.**InvalidModificationErr**

노드 형을 수정하려고 할 때 발생합니다.

exception xml.dom.**InvalidStateErr**

정의되지 않았거나 더는 사용할 수 없는 객체를 사용하려고 할 때 발생합니다.

exception xml.dom.**NamespaceErr**

Namespaces in XML 권장 사항에서 허용되지 않는 방식으로 객체를 변경하려고 시도하면 이 예외가 발생합니다.

exception xml.dom.**NotFoundErr**

참조된 컨텍스트에 노드가 존재하지 않을 때 발생하는 예외. 예를 들어, `NamedNodeMap.removeNamedItem()` 은 전달된 노드가 맵에 존재하지 않으면 이것을 발생시킵니다.

exception xml.dom.**NotSupportedErr**

구현이 요청된 형의 객체나 연산을 지원하지 않을 때 발생합니다.

exception xml.dom.**NoDataAllowedErr**

데이터를 지원하지 않는 노드에 데이터가 지정되면 발생합니다.

exception xml.dom.**NoModificationAllowedErr**

수정이 허용되지 않는 객체(가령 읽기 전용 노드)를 수정하려고 하면 발생합니다.

exception xml.dom.**SyntaxErr**

유효하지 않거나 잘못된 문자열이 지정될 때 발생합니다.

exception xml.dom.**WrongDocumentErr**

노드가 현재 속한 것과 다른 문서에 삽입되고 구현이 한 문서에서 다른 문서로 노드 이전을 지원하지 않을 때 발생합니다.

DOM 권장 사항에 정의된 예외 코드는 이 테이블에 따라 위에서 설명한 예외에 매핑됩니다:

상수	예외
DOMSTRING_SIZE_ERR	*DomstringSizeErr*
HIERARCHY_REQUEST_ERR	*HierarchyRequestErr*
INDEX_SIZE_ERR	*IndexSizeErr*
INUSE_ATTRIBUTE_ERR	*InuseAttributeErr*
INVALID_ACCESS_ERR	*InvalidAccessErr*
INVALID_CHARACTER_ERR	*InvalidCharacterErr*
INVALID_MODIFICATION_ERR	*InvalidModificationErr*
INVALID_STATE_ERR	*InvalidStateErr*
NAMESPACE_ERR	*NamespaceErr*
NOT_FOUND_ERR	*NotFoundErr*
NOT_SUPPORTED_ERR	*NotSupportedErr*
NO_DATA_ALLOWED_ERR	*NoDataAllowedErr*
NO_MODIFICATION_ALLOWED_ERR	*NoModificationAllowedErr*
SYNTAX_ERR	*SyntaxErr*
WRONG_DOCUMENT_ERR	*WrongDocumentErr*

21.6.3 규격 준수

이 섹션에서는 규격 준수 요구 사항과 파이썬 DOM API, W3C DOM 권장 사항 및 파이썬의 OMG IDL 매핑 간의 관계에 관해 설명합니다.

형 매핑

DOM 명세에 사용된 IDL 형은 다음 표에 따라 파이썬 형에 매핑됩니다.

IDL 형	파이썬 형
boolean	bool 또는 int
int	int
long int	int
unsigned int	int
DOMString	str 또는 bytes
null	None

접근자 메서드

OMG IDL에서 파이썬으로의 매핑은 Java 매핑과 거의 같은 방식으로 IDL `attribute` 선언에 대한 접근자 함수를 정의합니다. 다음과 같은 IDL 선언 매핑은:

```
readonly attribute string someValue;
         attribute string anotherValue;
```

세 개의 접근자 함수를 산출합니다: `someValue`를 위한 "get" 메서드(`_get_someValue()`)와 `anotherValue`를 위한 "get"과 "set" 메서드(`_get_anotherValue()`와 `_set_anotherValue()`). 특히, 매핑은 IDL 어트리뷰트가 일반 파이썬 어트리뷰트로 액세스할 수 있도록 요구하지 않습니다: `object.someValue`는 작동할 필요 없으며, *AttributeError*를 발생시킬 수 있습니다.

그러나, 파이썬 DOM API는 일반 어트리뷰트 액세스가 동작하도록 요구합니다. 이것은 파이썬 IDL 컴파일러에 의해 생성된 일반적인 서로게이트가 작동하지 않을 수 있으며, DOM 객체가 CORBA를 통해 액세스되는 경우 래퍼 객체가 클라이언트에 필요할 수 있음을 의미합니다. CORBA DOM 클라이언트에 대해 추가적인 고려가 필요하지만, 파이썬에서 CORBA를 통해 DOM을 사용한 경험이 있는 구현자들은 이것을 문제라고 보지 않습니다. `readonly`로 선언된 어트리뷰트는 모든 DOM 구현에서 쓰기 액세스를 제한하지 않을 수 있습니다.

파이썬 DOM API에서는, 접근자 함수가 필요하지 않습니다. 제공되면, 파이썬 IDL 매핑으로 정의된 형식을 취해야 하지만, 어트리뷰트를 파이썬에서 직접 액세스할 수 있어서 이러한 메서드들은 불필요한 것으로 간주합니다. `readonly` 어트리뷰트에 "set" 접근자를 제공해서는 안 됩니다.

IDL 정의는 W3C DOM API의 요구 사항을 완전히 담지 않습니다. 가령 `getElementsByTagName()`의 반환 값과 같은 특정 객체가 "살아있다(live)"는 개념과 같은 것을 표현하지 못합니다. 파이썬 DOM API는 구현이 이러한 요구 사항을 강제하도록 요구하지 않습니다.

21.7 `xml.dom.minidom` — 최소 DOM 구현

소스 코드: Lib/xml/dom/minidom.py

---

*xml.dom.minidom*은 다른 언어와 유사한 API를 갖는 문서 객체 모델 인터페이스의 최소 구현입니다. 전체(full) DOM보다 단순하고 훨씬 작고자 합니다. DOM에 아직 능숙하지 않은 사용자는 XML 처리에 대신 *xml.etree.ElementTree* 모듈을 사용하는 것을 고려해야 합니다.

> **참고**
>
> If you need to parse untrusted or unauthenticated data, see *XML security*.

DOM 응용 프로그램은 일반적으로 일부 XML을 DOM으로 구문 분석하는 것으로 시작합니다. *xml.dom.minidom*에서는, 구문 분석 함수를 통해 수행됩니다:

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml')  # 파일명으로 XML 파일을 구문 분석합니다

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # 열린 파일을 구문 분석합니다

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

parse() 함수는 파일명이나 열린 파일 객체를 취할 수 있습니다.

xml.dom.minidom.**parse**(*filename_or_file*, *parser=None*, *bufsize=None*)

주어진 입력에서 Document를 반환합니다. *filename_or_file*은 파일명이나 파일류 객체일 수 있습니다. 제공되면, *parser*는 SAX2 구문 분석기 객체여야 합니다. 이 함수는 구문 분석기의 문서 처리기를 변경하고 이름 공간 지원을 활성화합니다; 다른 구문 분석기 구성(엔티티 해석기 설정과 같은)은 미리 수행되어 있어야 합니다.

문자열로 XML을 갖고 있다면, *parseString()* 함수를 대신 사용할 수 있습니다:

xml.dom.minidom.**parseString**(*string*, *parser=None*)

*string*을 표현하는 Document를 반환합니다. 이 메서드는 문자열에 대한 *io.StringIO* 객체를 만들고 이를 *parse()*에 전달합니다.

두 함수 모두 문서의 내용을 표현하는 Document 객체를 반환합니다.

*parse()*와 *parseString()* 함수가 하는 일은 임의의 SAX 구문 분석기에서 구문 분석 이벤트를 받아들일 수 있고 이를 DOM 트리로 변환하는 "DOM 구축기(builder)"를 XML 구문 분석기와 연결하는 것입니다. 함수의 이름은 오해의 소지가 있지만, 인터페이스를 배울 때 이해하기 쉽습니다. 이 함수가 반환되기 전에 문서 구문 분석이 완료됩니다; 단지 이 함수들이 구문 분석기 구현 자체를 제공하지는 않을 뿐입니다.

"DOM 구현" 객체의 메서드를 호출하여 Document를 만들 수도 있습니다. *xml.dom* 패키지나 *xml.dom.minidom* 모듈에 있는 getDOMImplementation() 함수를 호출하여 이 객체를 얻을 수 있습니다. 일단 Document를 얻으면, 자식 노드를 추가하여 DOM을 채울 수 있습니다:

```
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

일단 DOM 문서 객체를 얻으면, 프로퍼티와 메서드를 통해 XML 문서의 일부에 액세스 할 수 있습니다. 이러한 프로퍼티들은 DOM 명세에 정의되어 있습니다. 문서 객체의 주 프로퍼티는 documentElement 프로퍼티입니다. XML 문서의 메인 엘리먼트를 제공합니다: 모든 다른 것들을 담는 것. 예제 프로그램은 다음과 같습니다:

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

DOM 트리로의 작업이 끝나면, unlink() 메서드를 선택적으로 호출하여 필요 없는 객체를 조기에 정리할 수 있습니다. unlink()는 DOM API에 대한 *xml.dom.minidom*만의 확장이며 그 노드와 자손들을 실질적으로 쓸모없게 만듭니다. 그렇지 않으면, 파이썬의 가비지 수집기가 결국 트리의 객체를 처리하게 될 것입니다.

더 보기

Document Object Model (DOM) Level 1 Specification
*xml.dom.minidom*이 지원하는 DOM에 대한 W3C 권장 사항.

21.7.1 DOM 객체

파이썬 용 DOM API의 정의는 `xml.dom` 모듈 설명서의 일부로 제공됩니다. 이 절은 그 API와 `xml.dom.minidom`의 차이점을 나열합니다.

`Node.unlink()`

순환 GC가 없는 파이썬 버전에서 가비지 수집되도록 DOM 내의 내부 참조를 끊습니다. 순환 GC를 사용할 수 있더라도, 이를 사용하면 대량의 메모리를 더 빨리 사용할 수 있도록 하므로, 더 필요 없게 되는 즉시 DOM 객체에 대해 이를 호출하는 것이 좋습니다. `Document` 객체에서만 호출하면 되지만, 해당 노드의 자식을 삭제하기 위해 자식 노드에서 호출할 수 있습니다.

`with` 문을 사용하면 이 메서드를 명시적으로 호출하지 않아도 됩니다. 다음 코드는 `with` 블록이 종료될 때 *dom*을 자동으로 unlink 합니다:

```
with xml.dom.minidom.parse(datasource) as dom:
    ... # dom으로 작업합니다.
```

`Node.writexml(writer, indent=", addindent=", newl=", encoding=None, standalone=None)`

기록기(writer) 객체에 XML을 씁니다. 기록기는 입력으로 텍스트를 받지만 바이트열은 받지 않습니다, 파일 객체 인터페이스와 일치하는 `write()` 메서드를 가져야 합니다. *indent* 매개 변수는 현재 노드의 들여쓰기입니다. *addindent* 매개 변수는 현재 노드의 서브 노드에 사용할 증분(incremental) 들여쓰기입니다. *newl* 매개 변수는 개행을 끝내는 데 사용할 문자열을 지정합니다.

`Document` 노드의 경우, 추가 키워드 인자 *encoding*을 사용하여 XML 헤더의 인코딩 필드를 지정할 수 있습니다.

유사하게, *standalone* 인자를 명시적으로 지정하면 standalone 문서 선언이 XML 문서의 프롤로그에 추가됩니다. 값이 `True`로 설정되면 `standalone="yes"`가 추가되고, 그렇지 않으면 `"no"`로 설정됩니다. 인자를 명시하지 않으면 문서에서 선언을 생략합니다.

버전 3.8에서 변경: `writexml()` 메서드는 이제 사용자가 지정한 어트리뷰트 순서를 유지합니다.

버전 3.9에서 변경: *standalone* 매개 변수를 추가했습니다.

`Node.toxml(encoding=None, standalone=None)`

DOM 노드가 나타내는 XML이 포함된 문자열이나 바이트열을 반환합니다.

명시적인 *encoding*[1] 인자를 사용하면, 결과는 지정된 인코딩의 바이트열 입니다. *encoding* 인자가 없으면, 결과는 유니코드 문자열이며, 결과 문자열의 XML 선언은 인코딩을 지정하지 않습니다. UTF-8이 XML의 기본 인코딩이기 때문에, UTF-8 이외의 인코딩으로 이 문자열을 인코딩하는 것은 올바르지 않습니다.

standalone 인자는 `writexml()`에서와 동일하게 동작합니다.

버전 3.8에서 변경: `toxml()` 메서드는 이제 사용자가 지정한 어트리뷰트 순서를 유지합니다.

버전 3.9에서 변경: *standalone* 매개 변수를 추가했습니다.

`Node.toprettyxml(indent='\t', newl='\n', encoding=None, standalone=None)`

문서의 예쁘게 인쇄된 버전을 반환합니다. *indent*는 들여쓰기 문자열을 지정하고 기본값은 탭입니다; *newl*은 각 줄의 끝에서 방출되는 문자열을 지정하고 기본값은 `\n`입니다.

encoding 인자는 `toxml()`의 해당 인자처럼 동작합니다.

standalone 인자는 `writexml()`에서와 동일하게 동작합니다.

버전 3.8에서 변경: `toprettyxml()` 메서드는 이제 사용자가 지정한 어트리뷰트 순서를 유지합니다.

버전 3.9에서 변경: *standalone* 매개 변수를 추가했습니다.

[1] XML 출력에 포함된 인코딩 이름은 적절한 표준을 준수해야 합니다. 예를 들어, XML 문서의 선언에서 "UTF-8"은 유효하지만, "UTF8"은 파이썬이 이를 인코딩 이름으로 받아들이더라도 유효하지 않습니다. https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl 과 https://www.iana.org/assignments/character-sets/character-sets.xhtml 을 참조하십시오.

21.7.2 DOM 예제

이 예제 프로그램은 간단한 프로그램의 상당히 현실적인 예입니다. 이 특별한 경우에, 우리는 DOM의 유연성을 크게 활용하지 않습니다.

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print("<html>")
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print("</html>")

def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print(f"<title>{getText(title.childNodes)}</title>")

def handleSlideTitle(title):
    print(f"<h2>{getText(title.childNodes)}</h2>")

def handlePoints(points):
    print("<ul>")
    for point in points:
        handlePoint(point)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    print("</ul>")

def handlePoint(point):
    print(f"<li>{getText(point.childNodes)}</li>")

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print(f"<p>{getText(title.childNodes)}</p>")

handleSlideshow(dom)
```

21.7.3 minidom과 DOM 표준

xml.dom.minidom 모듈은 본질적으로 일부 DOM 2 기능(주로 이름 공간 기능)이 있는 DOM 1.0 호환 DOM 입니다.

파이썬에서 DOM 인터페이스의 사용법은 간단합니다. 다음과 같은 매핑 규칙이 적용됩니다:

- 인터페이스는 인스턴스 객체를 통해 액세스 됩니다. 응용 프로그램은 클래스를 직접 인스턴스로 만들어서는 안 됩니다; `Document` 객체에서 제공되는 생성자 함수를 사용해야 합니다. 파생 인터페이스는 베이스 인터페이스의 모든 연산(및 어트리뷰트)과 새로운 연산을 지원합니다.
- 연산은 메서드로 사용됩니다. DOM은 `in` 매개 변수만 사용하므로, 인자는 정상적인 순서(왼쪽에서 오른쪽으로)로 전달됩니다. 선택적 인자가 없습니다. `void` 연산은 `None`을 반환합니다.
- IDL 어트리뷰트는 인스턴스 어트리뷰트에 매핑됩니다. 파이썬 용 OMG IDL 언어 매핑과의 호환성을 위해, 접근자 메서드 `_get_foo()`와 `_set_foo()`를 통해 어트리뷰트 `foo`에 액세스할 수도 있습니다. `readonly` 어트리뷰트는 변경하지 않아야 합니다; 실행 시간에 강제되지는 않습니다.
- `short int`, `unsigned int`, `unsigned long long` 및 `boolean` 형은 모두 파이썬 정수 객체에 매핑됩니다.
- `DOMString` 형은 파이썬 문자열에 매핑됩니다. *xml.dom.minidom*은 바이트열이나 문자열을 지원하지만, 일반적으로 문자열을 생성합니다. `DOMString` 형의 값은 W3C의 DOM 명세에 의해 IDL `null` 값을 가질 수 있을 때 `None`일 수도 있습니다.
- `const` 선언은 해당 스코프에 있는 변수에 매핑됩니다 (예를 들어 `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`); 변경되지 않아야 합니다.
- `DOMException`은 현재 *xml.dom.minidom*에서 지원되지 않습니다. 대신, *xml.dom.minidom*은 *TypeError*와 *AttributeError*와 같은 표준 파이썬 예외를 사용합니다.
- `NodeList` 객체는 파이썬의 내장 리스트 형을 사용하여 구현됩니다. 이러한 객체는 DOM 명세에 정의된 인터페이스를 제공하지만, 이전 버전의 파이썬에서는 공식 API를 지원하지 않습니다. 그러나 W3C 권장 사항에 정의된 인터페이스보다 훨씬 "파이썬답습니다".

다음 인터페이스는 *xml.dom.minidom*에서 구현되지 않습니다:

- `DOMTimeStamp`
- `EntityReference`

이들 대부분은 대부분의 DOM 사용자에게 일반적인 쓸모를 제공하지 않는 XML 문서의 정보를 반영합니다.

21.8 `xml.dom.pulldom` — 부분 DOM 트리 구축 지원

소스 코드: Lib/xml/dom/pulldom.py

xml.dom.pulldom 모듈은 필요할 때 문서의 DOM 액세스 가능한 조각을 생성하도록 요청할 수 있는 "풀 구문 분석기 (pull parser)"를 제공합니다. 기본 개념은 들어오는 XML 스트림에서 "이벤트"를 끌어당겨서 (pull) 처리하는 것입니다. 콜백을 통한 이벤트 구동 처리 모델(event-driven processing model)을 사용하는 SAX와 달리 풀 구문 분석기 사용자는 스트림에서 이벤트를 명시적으로 가져와서 처리가 완료되거나 에러 조건이 발생할 때까지 그 이벤트들을 루핑해야 합니다.

> **참고**
>
> If you need to parse untrusted or unauthenticated data, see *XML security*.

버전 3.7.1에서 변경: SAX 구문 분석기는 보안을 강화하기 위해 더는 일반 외부 엔티티를 처리하지 않습니다. 외부 엔티티를 처리를 활성화하려면, 사용자 정의 구문 분석기 인스턴스를 전달하십시오:

```
from xml.dom.pulldom import parse
from xml.sax import make_parser
from xml.sax.handler import feature_external_ges

parser = make_parser()
parser.setFeature(feature_external_ges, True)
parse(filename, parser=parser)
```

예:

```
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'item':
        if int(node.getAttribute('price')) > 50:
            doc.expandNode(node)
            print(node.toxml())
```

`event`는 상수이며 다음 중 하나일 수 있습니다:

- `START_ELEMENT`
- `END_ELEMENT`
- `COMMENT`
- `START_DOCUMENT`
- `END_DOCUMENT`
- `CHARACTERS`
- `PROCESSING_INSTRUCTION`
- `IGNORABLE_WHITESPACE`

`node`는 `xml.dom.minidom.Document`, `xml.dom.minidom.Element` 또는 `xml.dom.minidom.Text` 형의 객체입니다.

문서는 "평평한(flat)" 이벤트 스트림으로 취급되므로, 문서 "트리"는 묵시적으로 탐색 되며 트리에서의 깊이와 관계없이 원하는 요소를 찾습니다. 다시 말해, 문서 노드의 재귀적 검색과 같은 계층적 문제를 고려할 필요는 없습니다. 하지만, 엘리먼트의 문맥이 중요하다면, 문맥과 관련된 상태를 유지하거나 (즉, 주어진 지점에서 문서의 어느 위치에 있는지 기억함으로써), *DOMEventStream.expandNode()* 메서드를 사용하고 DOM 관련 처리로 전환해야 합니다.

class `xml.dom.pulldom.`**`PullDom`**(*documentFactory=None*)

*xml.sax.handler.ContentHandler*의 서브 클래스.

class xml.dom.pulldom.**SAX2DOM**(*documentFactory=None*)

*xml.sax.handler.ContentHandler*의 서브 클래스.

xml.dom.pulldom.**parse**(*stream_or_string*, *parser=None*, *bufsize=None*)

주어진 입력으로부터 *DOMEventStream*을 반환합니다. *stream_or_string*은 파일 이름이거나 파일류 객체일 수 있습니다. 주어질 때, *parser*는 *XMLReader* 객체여야 합니다. 이 함수는 구문 분석기의 문서 처리기를 변경하고 이름 공간 지원을 활성화합니다; 다른 구문 분석기 구성(엔티티 해석기 설정과 같은)은 미리 수행되어 있어야 합니다.

문자열로 XML을 갖고 있다면, *parseString()* 함수를 대신 사용할 수 있습니다:

xml.dom.pulldom.**parseString**(*string*, *parser=None*)

(유니코드) *string*을 표현하는 *DOMEventStream*을 반환합니다.

xml.dom.pulldom.**default_bufsize**

*parse()*의 *bufsize* 매개 변수의 기본값.

이 변수의 값은 *parse()*를 호출하기 전에 변경될 수 있으며 새 값이 적용됩니다.

21.8.1 DOMEventStream 객체

class xml.dom.pulldom.**DOMEventStream**(*stream*, *parser*, *bufsize*)

버전 3.11에서 변경: `__getitem__()` 메서드 지원은 제거했습니다.

getEvent()

*event*와 현재 *node*를 포함하는 튜플을 반환합니다. 노드는 이벤트가 `START_DOCUMENT`와 같으면 `xml.dom.minidom.Document`, 이벤트가 `START_ELEMENT`나 `END_ELEMENT`와 같으면 `xml.dom.minidom.Element`, 이벤트가 `CHARACTERS`와 같으면 `xml.dom.minidom.Text` 입니다. *expandNode()*가 호출되지 않는 한 현재 노드에는 자식에 대한 정보가 없습니다.

expandNode(*node*)

*node*의 모든 자식을 *node*로 확장합니다. 예:

```
from xml.dom import pulldom

xml = '<html><title>Foo</title> <p>Some text <div>and more</div></p> </
↪html>'
doc = pulldom.parseString(xml)
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'p':
        # 다음 문장은 '<p/>' 만 인쇄합니다
        print(node.toxml())
        doc.expandNode(node)
        # 다음 문장은 노드와 모든 자식을 인쇄합니다 '<p>Some text <div>and␣
↪more</div></p>'
        print(node.toxml())
```

reset()

21.9 xml.sax — SAX2 구문 분석기 지원

소스 코드: Lib/xml/sax/__init__.py

---

xml.sax 패키지는 파이썬용 SAX(Simple API for XML) 인터페이스를 구현하는 여러 모듈을 제공합니다. 패키지 자체는 대부분 SAX API의 사용자가 사용하게 될 SAX 예외와 편리 함수를 제공합니다.

> **참고**
>
> If you need to parse untrusted or unauthenticated data, see *XML security*.

버전 3.7.1에서 변경: SAX 구문 분석기는 보안을 강화하기 위해 더는 일반 외부 엔티티를 처리하지 않습니다. 이전에는, 구문 분석기가 DTD와 엔티티에 대해 네트워크 연결을 만들어 원격 파일을 가져오거나 파일 시스템에서 로컬 파일을 로드했습니다. 이 기능은 구문 분석기 객체에 대해 인자 `feature_external_ges`로 메서드 `setFeature()`를 사용하여 다시 활성화할 수 있습니다.

편리 함수는 다음과 같습니다:

`xml.sax.`**`make_parser`**(*parser_list=[]*)

SAX `XMLReader` 객체를 만들고 반환합니다. 발견된 첫 번째 구문 분석기가 사용됩니다. *parser_list*가 제공되면, `create_parser()` 라는 함수가 있는 모듈의 이름을 나타내는 문자열의 이터러블이어야 합니다. *parser_list*에 나열된 모듈은 기본 구문 분석기 목록에 있는 모듈보다 먼저 사용됩니다.

버전 3.8에서 변경: *parser_list* 인자는 리스트뿐만 아니라 임의의 이터러블일 수 있습니다.

`xml.sax.`**`parse`**(*filename_or_stream, handler, error_handler=handler.ErrorHandler()*)

SAX 구문 분석기를 만들어 문서 구문 분석에 사용합니다. *filename_or_stream*으로 전달된 문서는 파일명이나 파일 객체일 수 있습니다. *handler* 매개 변수는 SAX `ContentHandler` 인스턴스여야 합니다. *error_handler*가 주어지면, SAX `ErrorHandler` 인스턴스여야 합니다; 생략하면, 모든 에러에서 `SAXParseException`이 발생합니다. 반환 값은 없습니다; 모든 작업은 전달된 *handler*가 처리해야 합니다.

`xml.sax.`**`parseString`**(*string, handler, error_handler=handler.ErrorHandler()*)

`parse()`와 비슷하지만, 매개 변수로 받은 버퍼 *string*에서 구문 분석합니다. *string*은 `str` 인스턴스나 바이트열류 객체여야 합니다.

버전 3.5에서 변경: `str` 인스턴스에 대한 지원이 추가되었습니다.

전형적인 SAX 응용 프로그램은 입력기(reader), 처리기(handler) 및 입력 소스(input source)의 세 가지 종류의 객체를 사용합니다: 이 문맥에서 "입력기(Reader)"는 구문 분석기에 대한 또 다른 용어입니다, 즉, 입력 소스에서 바이트나 문자를 읽고, 이벤트 시퀀스를 생성하는 코드 조각입니다. 그런 다음 이벤트는 처리기 객체로 배포됩니다, 즉 입력기가 처리기의 메서드를 호출합니다. 따라서 SAX 응용 프로그램은 입력기 객체를 얻고, 입력 소스를 만들거나 열고, 처리기를 만들고, 이 객체들을 모두 연결해야 합니다. 준비의 마지막 단계로, 입력기가 입력을 구문 분석하도록 호출됩니다. 구문 분석하는 동안, 처리기 객체의 메서드는 입력 데이터로부터 온 구조적 및 구문적 이벤트를 기반으로 호출됩니다.

이러한 객체들은, 인터페이스만 중요합니다; 그들은 일반적으로 응용 프로그램 자체에 의해 인스턴스로 만들어지지 않습니다. 파이썬에는 인터페이스에 대한 명시적인 개념이 없으므로, 이것들이 형식적으로는 클래스로 소개되지만, 응용 프로그램은 제공된 클래스를 상속하지 않는 구현을 사용할 수 있습니다. `InputSource`, `Locator`, `Attributes`, `AttributesNS` 및 `XMLReader` 인터페이스는 모듈 `xml.sax.xmlreader`에 정의되어 있습니다. 처리기 인터페이스는 `xml.sax.handler`에 정의되어 있습니다. 편의상, `InputSource`(종종 직접 인스턴스를 만듭니다)와 처리기 클래스들은 `xml.sax`에서도 사용할 수 있습니다. 이러한 인터페이스는 아래에 설명되어 있습니다.

이러한 클래스 외에도, `xml.sax`는 다음과 같은 예외 클래스를 제공합니다.

exception `xml.sax.`**`SAXException`**(*msg, exception=None*)

XML 에러나 경고를 캡슐화합니다. 이 클래스에는 XML 구문 분석기나 응용 프로그램의 기본 에러나 경고 정보가 포함될 수 있습니다: 추가 기능을 제공하거나 지역화를 추가하기 위해 서브 클래싱 될 수 있습니다. `ErrorHandler` 인터페이스에 정의된 처리기가 이 예외의 인스턴스를 수신하지만, 예외를 실제로 발생시킬 필요는 없음에 유의하십시오 — 정보를 담는 컨테이너로도 유용합니다.

인스턴스로 만들어질 때, *msg*는 사람이 읽을 수 있는 에러에 관한 설명이어야 합니다. 선택적 *exception* 매개 변수를 주면, `None`이거나 구문 분석 코드에 의해 잡힌 예외여야 하고, 정보로 함께 전달됩니다.

이것은 다른 SAX 예외 클래스의 베이스 클래스입니다.

exception `xml.sax.`**`SAXParseException`**(*msg, exception, locator*)

구문 분석 에러 시 발생하는 `SAXException`의 서브 클래스. 이 클래스의 인스턴스는 SAX

ErrorHandler 인터페이스의 메서드에 전달되어 구문 분석 에러에 대한 정보를 제공합니다. 이 클래스는 *SAXException* 인터페이스뿐만 아니라 SAX *Locator* 인터페이스를 지원합니다.

exception xml.sax.**SAXNotRecognizedException**(*msg*, *exception=None*)

SAX *XMLReader*가 인식할 수 없는 기능이나 속성을 만날 때 발생하는 *SAXException*의 서브 클래스. SAX 응용 프로그램과 확장은 유사한 목적으로 이 클래스를 사용할 수 있습니다.

exception xml.sax.**SAXNotSupportedException**(*msg*, *exception=None*)

SAX *XMLReader*가 지원되지 않는 기능을 활성화하거나 구현에서 지원하지 않는 값으로 속성을 설정하도록 요청될 때 발생하는 *SAXException*의 서브 클래스. SAX 응용 프로그램과 확장은 유사한 목적으로 이 클래스를 사용할 수 있습니다.

더 보기

SAX: The Simple API for XML
이 사이트는 SAX API의 정의가 집중되는 곳입니다. Java 구현과 온라인 설명서를 제공합니다. 구현과 역사적 정보에 대한 링크도 있습니다.

모듈 *xml.sax.handler*
응용 프로그램이 제공하는 객체에 대한 인터페이스의 정의.

모듈 *xml.sax.saxutils*
SAX 응용 프로그램에서 사용하기 위한 편리 함수.

모듈 *xml.sax.xmlreader*
구문 분석기가 제공하는 객체에 대한 인터페이스의 정의.

21.9.1 SAXException 객체

SAXException 예외 클래스는 다음 메서드를 지원합니다:

SAXException.**getMessage**()

에러 상태를 설명하는 사람이 읽을 수 있는 메시지를 반환합니다.

SAXException.**getException**()

캡슐화된 예외 객체나 None을 반환합니다.

21.10 xml.sax.handler — SAX 처리기의 베이스 클래스

소스 코드: Lib/xml/sax/handler.py

SAX API는 다섯 가지 처리기를 정의합니다: 콘텐츠 처리기, DTD 처리기, 에러 처리기, 엔티티 해석기 및 어휘 처리기. 응용 프로그램은 일반적으로 이벤트에 관심이 있는 인터페이스 만 구현하면 됩니다; 단일 객체나 여러 객체에서 인터페이스를 구현할 수 있습니다. 처리기 구현은 모든 메서드가 기본 구현을 갖도록 *xml.sax.handler* 모듈에서 제공된 베이스 클래스에서 상속해야 합니다.

class xml.sax.handler.**ContentHandler**

이것은 SAX의 주 콜백 인터페이스이며, 응용 프로그램에 가장 중요한 인터페이스입니다. 이 인터페이스의 이벤트 순서는 문서에서의 정보 순서를 반영합니다.

class xml.sax.handler.**DTDHandler**

DTD 이벤트를 처리합니다.

이 인터페이스는 기본 구문 분석에 필요한 DTD 이벤트만 지정합니다 (구문 분석되지 않은 엔티티와 어트리뷰트).

class xml.sax.handler.**EntityResolver**

엔티티 해석을 위한 기본 인터페이스. 이 인터페이스를 구현하는 객체를 만든 후 구문 분석기에 객체를 등록하면, 구문 분석기는 객체의 메서드를 호출하여 모든 외부 엔티티를 해석합니다.

class xml.sax.handler.**ErrorHandler**

응용 프로그램에 에러와 경고 메시지를 표시하기 위해 구문 분석기에서 사용하는 인터페이스. 이 객체의 메서드는 에러가 즉시 예외로 변환되는지 또는 다른 방식으로 처리되는지를 제어합니다.

class xml.sax.handler.**LexicalHandler**

Interface used by the parser to represent low frequency events which may not be of interest to many applications.

이러한 클래스 외에도, *xml.sax.handler*는 기능과 속성 이름에 대한 기호 상수를 제공합니다.

xml.sax.handler.**feature_namespaces**

값: "http://xml.org/sax/features/namespaces"
참: 이름 공간 처리를 수행합니다.
거짓: 선택적으로 이름 공간 처리를 수행하지 않습니다 (namespace-prefixes를 암시합니다; 기본값).
액세스: (구문 분석) 읽기 전용; (구문 분석하지 않음) 읽기/쓰기

xml.sax.handler.**feature_namespace_prefixes**

값: "http://xml.org/sax/features/namespace-prefixes"
참: 이름 공간 선언에 사용된 원래 접두사 이름과 어트리뷰트를 보고합니다.
거짓: 이름 공간 선언에 사용된 어트리뷰트를 보고하지 않고, 선택적으로 원래 접두사 이름을 보고하지 않습니다 (기본값).
액세스: (구문 분석) 읽기 전용; (구문 분석하지 않음) 읽기/쓰기

xml.sax.handler.**feature_string_interning**

값: "http://xml.org/sax/features/string-interning"
참: 모든 엘리먼트 이름, 접두사, 어트리뷰트 이름, 이름 공간 URI 및 지역 이름은 내장 intern 함수를 사용하여 인턴 됩니다.
거짓: 그럴 수 있지만, 이름이 반드시 인턴 될 필요는 없습니다 (기본값).
액세스: (구문 분석) 읽기 전용; (구문 분석하지 않음) 읽기/쓰기

xml.sax.handler.**feature_validation**

값: "http://xml.org/sax/features/validation"
참: 모든 유효성 검사 에러를 보고합니다 (external-general-entities와 external-parameter-entities를 암시합니다).
거짓: 유효성 검사 에러를 보고하지 않습니다.
액세스: (구문 분석) 읽기 전용; (구문 분석하지 않음) 읽기/쓰기

xml.sax.handler.**feature_external_ges**

값: "http://xml.org/sax/features/external-general-entities"
참: 모든 외부 일반 (텍스트) 엔티티를 포함합니다.
거짓: 외부 일반 엔티티를 포함하지 않습니다.
액세스: (구문 분석) 읽기 전용; (구문 분석하지 않음) 읽기/쓰기

xml.sax.handler.**feature_external_pes**

값: "http://xml.org/sax/features/external-parameter-entities"
참: 외부 DTD 서브 세트를 포함하여, 모든 외부 파라미터 엔티티를 포함합니다.
거짓: 외부 DTD 서브 세트를 포함하여, 어떤 외부 파라미터 엔티티도 포함하지 않습니다.
액세스: (구문 분석) 읽기 전용; (구문 분석하지 않음) 읽기/쓰기

xml.sax.handler.**all_features**

모든 기능 리스트.

xml.sax.handler.**property_lexical_handler**

값: "http://xml.org/sax/properties/lexical-handler"
데이터형: xml.sax.handler.LexicalHandler (파이썬 2에서는 지원되지 않습니다)

설명: 주석과 같은 어휘 이벤트에 대한 선택적 확장 처리기.
액세스: 읽기/쓰기

xml.sax.handler.**property_declaration_handler**

값: "http://xml.org/sax/properties/declaration-handler"
데이터형: xml.sax.sax2lib.DeclHandler (파이썬 2에서는 지원되지 않습니다)
설명: 표기법과 구문 분석되지 않은 엔티티 이외의 DTD 관련 이벤트에 대한 선택적 확장 처리기.
액세스: 읽기/쓰기

xml.sax.handler.**property_dom_node**

값: "http://xml.org/sax/properties/dom-node"
데이터형: org.w3c.dom.Node (파이썬 2에서는 지원되지 않습니다)
설명: 구문 분석할 때, 이것이 DOM 이터레이터이면 방문 중인 현재 DOM 노드; 구문 분석하지 않을 때, 이터레이션을 위한 루트 DOM 노드.
액세스: (구문 분석) 읽기 전용; (구문 분석하지 않음) 읽기/쓰기

xml.sax.handler.**property_xml_string**

값: "http://xml.org/sax/properties/xml-string"
데이터형: 바이트열
설명: 현재 이벤트의 소스인 리터럴 문자열.
액세스: 읽기 전용

xml.sax.handler.**all_properties**
알려진 모든 속성 이름 리스트

21.10.1 ContentHandler 객체

사용자는 자신의 응용 프로그램을 지원하기 위해 *ContentHandler*를 서브 클래싱 할 것으로 기대됩니다. 입력 문서에서 적절한 이벤트에 대해 구문 분석기가 다음 메서드를 호출합니다:

ContentHandler.**setDocumentLocator**(*locator*)
응용 프로그램에 문서 이벤트의 출처를 찾기 위한 로케이터(locator)를 제공하기 위해 구문 분석기가 호출합니다.

SAX 구문 분석기가 (절대적으로 필요한 것은 아니지만) 로케이터를 제공할 것을 강력히 권장합니다: 그렇게 한다면, DocumentHandler 인터페이스의 다른 메서드를 호출하기 전에 이 메서드를 호출하여 로케이터를 응용 프로그램에 제공해야 합니다.

로케이터를 사용하면 구문 분석기가 에러를 보고하지 않더라도 응용 프로그램이 문서 관련 이벤트의 종료 위치를 판별할 수 있습니다. 일반적으로, 응용 프로그램은 이 정보를 사용하여 자체 에러(가령 응용 프로그램의 비즈니스 규칙과 일치하지 않는 문자 내용)를 보고합니다. 로케이터가 반환한 정보는 아마도 검색 엔진에 사용하기에는 충분하지 않을 것입니다.

로케이터는 이 인터페이스에서 이벤트를 호출하는 동안에만 올바른 정보를 반환함에 유의하십시오. 응용 프로그램은 다른 시간에 사용하려고 시도해서는 안 됩니다.

ContentHandler.**startDocument**()
문서 시작 알림을 받습니다.

SAX 구문 분석기는 이 인터페이스나 DTDHandler의 다른 모든 메서드(*setDocumentLocator()* 제외) 전에 이 메서드를 한 번만 호출합니다.

ContentHandler.**endDocument**()
문서 끝 알림을 받습니다.

SAX 구문 분석기는 이 메서드를 한 번만 호출하며, 그것이 구문 분석 중에 마지막으로 호출된 메서드가 됩니다. 구문 분석기는 (복구할 수 없는 에러로 인해) 구문 분석을 포기했거나 입력 끝에 도달할 때까지 이 메서드를 호출하지 않아야 합니다.

ContentHandler.**startPrefixMapping**(*prefix*, *uri*)

접두사 URI 이름 공간 매핑의 스코프를 시작합니다.

이 이벤트의 정보는 일반적인 이름 공간 처리에 필요하지 않습니다: SAX XML 판독기는 feature_namespaces 기능이 활성화되면 (기본값) 엘리먼트와 어트리뷰트 이름의 접두사를 자동으로 대체합니다.

그러나, 응용 프로그램이 문자 데이터나 어트리뷰트 값에 접두사를 사용해야 할 때 자동으로 안전하게 확장할 수 없는 경우가 있습니다; startPrefixMapping() 과 endPrefixMapping() 이벤트는 필요한 경우 해당 컨텍스트에서 스스로 접두사를 확장하도록 정보를 응용 프로그램에 제공합니다.

startPrefixMapping() 과 endPrefixMapping() 이벤트는 서로에 대해 올바르게 중첩된다고 보장되지 않음에 유의하십시오: 모든 startPrefixMapping() 이벤트는 해당 startElement() 이벤트 전에 발생하고, 모든 endPrefixMapping() 이벤트는 해당 endElement() 이벤트 후에 발생하지만, 그들 간의 순서는 보장되지 않습니다.

ContentHandler.**endPrefixMapping**(*prefix*)

접두사 URI 매핑 스코프를 끝냅니다.

자세한 내용은 startPrefixMapping() 을 참조하십시오. 이 이벤트는 항상 해당 endElement() 이벤트 이후에 발생하지만, endPrefixMapping() 이벤트의 순서는 이 이상으로는 보장되지 않습니다.

ContentHandler.**startElement**(*name*, *attrs*)

이름 공간이 아닌 모드에서 엘리먼트의 시작을 알립니다.

The *name* parameter contains the raw XML 1.0 name of the element type as a string and the *attrs* parameter holds an object of the *Attributes* interface containing the attributes of the element. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the copy() method of the *attrs* object.

ContentHandler.**endElement**(*name*)

이름 공간이 아닌 모드에서 엘리먼트의 끝을 알립니다.

name 매개 변수는 startElement() 이벤트와 마찬가지로 엘리먼트 유형의 이름을 포함합니다.

ContentHandler.**startElementNS**(*name*, *qname*, *attrs*)

이름 공간 모드에서 엘리먼트의 시작을 알립니다.

The *name* parameter contains the name of the element type as a (uri, localname) tuple, the *qname* parameter contains the raw XML 1.0 name used in the source document, and the *attrs* parameter holds an instance of the *AttributesNS* interface containing the attributes of the element. If no namespace is associated with the element, the *uri* component of *name* will be None. The object passed as *attrs* may be re-used by the parser; holding on to a reference to it is not a reliable way to keep a copy of the attributes. To keep a copy of the attributes, use the copy() method of the *attrs* object.

feature_namespace_prefixes 기능이 활성화되지 않는 한, 구문 분석기는 *qname* 매개 변수를 None 으로 설정할 수 있습니다.

ContentHandler.**endElementNS**(*name*, *qname*)

이름 공간 모드에서 엘리먼트의 끝을 알립니다.

name 매개 변수는 startElementNS() 메서드와 마찬가지로 *qname* 매개 변수처럼 엘리먼트 유형의 이름을 포함합니다.

ContentHandler.**characters**(*content*)

문자 데이터의 알림을 받습니다.

구문 분석기는 이 메서드를 호출하여 각 문자 데이터 청크를 보고합니다. SAX 구문 분석기는 연속된 모든 문자 데이터를 단일 청크로 반환하거나 여러 청크로 분할할 수 있습니다; 그러나, 로케이터가 유용한 정보를 제공할 수 있도록, 단일 이벤트의 모든 문자는 같은 외부 엔티티에서 온 것이어야 합니다.

*content*는 문자열이나 바이트열 인스턴스일 수 있습니다; expat 판독기 모듈은 항상 문자열을 생성합니다.

> **참고**
>
> Python XML Special Interest Group에서 제공된 이전 SAX 1 인터페이스는 이 메서드에 더 Java와 유사한 인터페이스를 사용했습니다. 파이썬에서 사용되는 대부분의 구문 분석기가 구식 인터페이스의 장점을 취하지 않았기 때문에, 더 간단한 서명으로 변경했습니다. 이전 코드를 새 인터페이스로 변환하려면, 이전 *offset*과 *length* 매개 변수로 콘텐츠를 슬라이싱하는 대신 *content*를 사용하십시오.

`ContentHandler.`**`ignorableWhitespace`**(*whitespace*)

엘리먼트 내용에서 무시할 수 있는 공백에 대한 알림을 받습니다.

유효성 검사 구문 분석기는 무시할 수 있는 공백의 각 청크를 보고하기 위해 이 메서드를 사용해야 합니다 (W3C XML 1.0 권장 사항, 섹션 2.10을 참조하십시오).

SAX 구문 분석기는 모든 연속적인 공백을 단일 청크로 반환하거나 여러 청크로 나눌 수 있습니다; 그러나 로케이터가 유용한 정보를 제공할 수 있도록, 단일 이벤트의 모든 문자는 같은 외부 엔티티에서 온 것이어야 합니다.

`ContentHandler.`**`processingInstruction`**(*target*, *data*)

처리 명령의 알림을 받습니다.

구문 분석기는 발견된 각 처리 명령에 대해 이 메서드를 한 번 호출합니다: 처리 명령은 메인 문서 엘리먼트 전후에 발생할 수 있음에 유의하십시오.

SAX 구문 분석기는 이 메서드를 사용하여 XML 선언(XML 1.0, 섹션 2.8)이나 텍스트 선언(XML 1.0, 섹션 4.3.1)을 보고해서는 안 됩니다.

`ContentHandler.`**`skippedEntity`**(*name*)

건너뛴 엔티티의 알림을 받습니다.

구문 분석기는 각 엔티티를 건너뛸 때마다 이 메서드를 한 번 호출합니다. 유효성을 검사하지 않는 프로세서는 선언을 보지 않으면 엔티티를 건너뛸 수 있습니다 (예를 들어 엔티티가 외부 DTD 서브세트에서 선언되었기 때문에). `feature_external_ges`와 `feature_external_pes` 속성의 값에 따라, 모든 프로세서가 외부 엔티티를 건너뛸 수 있습니다.

21.10.2 DTDHandler 객체

`DTDHandler` 인스턴스는 다음 메서드를 제공합니다:

`DTDHandler.`**`notationDecl`**(*name*, *publicId*, *systemId*)

표기법 선언 이벤트를 처리합니다.

`DTDHandler.`**`unparsedEntityDecl`**(*name*, *publicId*, *systemId*, *ndata*)

구문 분석되지 않은 엔티티 선언 이벤트를 처리합니다.

21.10.3 EntityResolver 객체

`EntityResolver.`**`resolveEntity`**(*publicId*, *systemId*)

엔티티의 시스템 식별자를 해석하고 읽을 시스템 식별자를 문자열로 반환하거나, 읽을 InputSource를 반환합니다. 기본 구현은 *systemId*를 반환합니다.

21.10.4 ErrorHandler 객체

이 인터페이스를 갖는 객체는 `XMLReader`에서 에러와 경고 정보를 받는 데 사용됩니다. 이 인터페이스를 구현하는 객체를 만든 다음 `XMLReader`에 객체를 등록하면, 구문 분석기는 객체의 메서드를 호출하여 모든 경고와 에러를 보고합니다. 세 가지 수준의 에러가 있습니다: 경고, (어쩌면) 복구 가능한 에러 및 복구 불가능한 에러. 모든 메서드는 `SAXParseException`를 유일한 매개 변수로 취합니다. 전달된 예외 객체를 발생 시켜 에러와 경고를 예외로 변환할 수 있습니다.

ErrorHandler.**error**(*exception*)

구문 분석기가 복구 가능한 에러를 만나면 호출됩니다. 이 메서드가 예외를 발생시키지 않으면, 구문 분석이 계속될 수 있지만, 응용 프로그램에서 추가 문서 정보를 기대하지 않아야 합니다. 구문 분석기가 계속되도록 허용하면 입력 문서에서 추가 에러가 발견될 수 있습니다.

ErrorHandler.**fatalError**(*exception*)

구문 분석기가 복구 불가능한 에러를 만나면 호출됩니다; 이 메서드가 반환될 때 구문 분석이 종료될 것으로 기대합니다.

ErrorHandler.**warning**(*exception*)

구문 분석기가 응용 프로그램에 경미한 경고 정보를 제공할 때 호출됩니다. 이 메서드가 반환될 때 구문 분석이 계속될 것으로 기대되며, 문서 정보는 계속 응용 프로그램에 전달됩니다. 이 메서드에서 예외를 발생시키면 구문 분석이 종료됩니다.

21.10.5 LexicalHandler 객체

어휘 이벤트에 대한 선택적 SAX2 처리기.

This handler is used to obtain lexical information about an XML document. Lexical information includes information describing the document encoding used and XML comments embedded in the document, as well as section boundaries for the DTD and for any CDATA sections. The lexical handlers are used in the same manner as content handlers.

Set the LexicalHandler of an XMLReader by using the setProperty method with the property identifier `'http://xml.org/sax/properties/lexical-handler'`.

LexicalHandler.**comment**(*content*)

Reports a comment anywhere in the document (including the DTD and outside the document element).

LexicalHandler.**startDTD**(*name*, *public_id*, *system_id*)

Reports the start of the DTD declarations if the document has an associated DTD.

LexicalHandler.**endDTD**()

Reports the end of DTD declaration.

LexicalHandler.**startCDATA**()

Reports the start of a CDATA marked section.

The contents of the CDATA marked section will be reported through the characters handler.

LexicalHandler.**endCDATA**()

Reports the end of a CDATA marked section.

21.11 xml.sax.saxutils — SAX 유틸리티

소스 코드: Lib/xml/sax/saxutils.py

---

xml.sax.saxutils 모듈은 SAX 응용 프로그램을 만들 때 직접 사용하거나 베이스 클래스로 사용하는 데 모두 유용한 많은 클래스와 함수를 포함합니다.

xml.sax.saxutils.**escape**(*data*, *entities*={})

data 문자열에 있는 `'&'`, `'<'` 및 `'>'`를 이스케이프 합니다.

딕셔너리를 선택적 *entities* 매개 변수로 전달하여 data의 다른 문자열을 이스케이프 할 수 있습니다. 키와 값은 모두 문자열이어야 합니다; 각 키는 해당 값으로 치환되게 됩니다. 문자 `'&'`, `'<'` 및 `'>'`는 *entities*가 제공되더라도 항상 이스케이프 됩니다.

> **참고**
>
> This function should only be used to escape characters that can' t be used directly in XML. Do not use this function as a general string translation function.

`xml.sax.saxutils.`**`unescape`**(*data*, *entities*={})

data 문자열에 있는 `'&'`, `'<'` 및 `'>'`를 역 이스케이프 합니다.

딕셔너리를 선택적 *entities* 매개 변수로 전달하여 data의 다른 문자열을 역 이스케이프 할 수 있습니다. 키와 값은 모두 문자열이어야 합니다; 각 키는 해당 값으로 치환되게 됩니다. `'&'`, `'<'` 및 `'>'`는 *entities*가 제공되더라도 항상 역 이스케이프 됩니다.

`xml.sax.saxutils.`**`quoteattr`**(*data*, *entities*={})

`escape()`와 비슷하지만, *data*가 어트리뷰트 값으로 사용되도록 준비합니다. 반환 값은 추가로 필요한 치환이 적용된 *data*의 따옴표 붙은 버전입니다. `quoteattr()`는 *data*의 내용에 따라 인용 부호 문자를 선택하여 가능하면 문자열의 인용 부호 문자를 인코딩하지 않습니다. 작은따옴표와 큰따옴표가 모두 *data*에 이미 있으면, 큰따옴표 문자가 인코딩되고, *data*는 큰따옴표로 묶입니다. 결과 문자열은 어트리뷰트 값으로 직접 사용할 수 있습니다:

```
>>> print("<element attr=%s>" % quoteattr("ab ' cd \" ef"))
<element attr="ab ' cd &quot; ef">
```

이 함수는 참조 구상 문법(reference concrete syntax)을 사용하여 HTML이나 모든 SGML을 위한 어트리뷰트 값을 생성할 때 유용합니다.

class `xml.sax.saxutils.`**`XMLGenerator`**(*out=None*, *encoding='iso-8859-1'*, *short_empty_elements=False*)

이 클래스는 SAX 이벤트를 다시 XML 문서에 쓰는 방식으로 `ContentHandler` 인터페이스를 구현합니다. 다시 말해, `XMLGenerator`를 내용 처리기로 사용하면 구문 분석 중인 원본 문서를 재생산합니다. *out*은 파일류 객체 여야하고, 기본값은 *sys.stdout*입니다. *encoding*은 출력 스트림의 인코딩이고, 기본값은 `'iso-8859-1'`입니다. *short_empty_elements*는 내용이 없는 엘리먼트의 형식을 제어합니다: `False`(기본값)는 시작/끝 태그 쌍으로 출력하고, `True`로 설정하면 하나의 스스로 닫힌 태그를 출력합니다.

버전 3.2에서 변경: *short_empty_elements* 매개 변수를 추가했습니다.

class `xml.sax.saxutils.`**`XMLFilterBase`**(*base*)

이 클래스는 `XMLReader`와 클라이언트 응용 프로그램의 이벤트 처리기 사이에 위치하도록 설계되었습니다. 기본적으로, 이것은 요청을 입력기에 전달하고 이벤트를 변경 없이 처리기에 전달할 뿐 아무것도 하지 않지만, 서브 클래스는 특정 메서드를 재정의하여 이벤트 스트림이나 구성 요청이 지나갈 때 수정할 수 있습니다.

`xml.sax.saxutils.`**`prepare_input_source`**(*source*, *base=''*)

이 함수는 입력 소스와 선택적인 베이스 URL을 받아들이고 완전히 결정되고 읽을 준비가 된 `InputSource` 객체를 반환합니다. 입력 소스는 문자열, 파일류 객체 또는 `InputSource` 객체로 지정할 수 있습니다; 구문 분석기는 `parse()` 메서드에 대한 다형적인 *source* 인자를 구현하는 데 이 함수를 사용할 수 있습니다.

21.12 `xml.sax.xmlreader` — XML 구문 분석기 인터페이스

소스 코드: Lib/xml/sax/xmlreader.py

SAX 구문 분석기는 `XMLReader` 인터페이스를 구현합니다. 이들은 함수 `create_parser()`를 제공해야 하는 파이썬 모듈로 구현됩니다. 이 함수는 새로운 구문 분석기 객체를 만들기 위해 인자 없이 `xml.sax.make_parser()`에 의해 호출됩니다.

class `xml.sax.xmlreader.`**`XMLReader`**

SAX 구문 분석기가 상속할 수 있는 베이스 클래스.

class xml.sax.xmlreader.**IncrementalParser**

때에 따라 입력 소스를 한 번에 구문 분석하지 않고 문서를 사용 가능할 때마다 청크로 공급하는 것이 바람직합니다. 입력기(reader)는 일반적으로 전체 파일을 읽지 않고 덩어리로 읽습니다; 여전히 전체 문서가 처리될 때까지 `parse()` 는 반환하지 않습니다. 따라서 `parse()` 의 블로킹 동작이 바람직하지 않을 때 이 인터페이스를 사용해야 합니다.

구문 분석기가 인스턴스화 되면 즉시 feed 메서드에서 데이터를 받아들일 수 있습니다. 구문 분석이 close 호출로 완료된 후, 구문 분석기가 feed나 parse 메서드를 사용하여 새 데이터를 받아들일 준비가 되도록 하려면 reset 메서드를 호출해야 합니다.

이러한 메서드들은 구문 분석 중, 즉 parse가 호출된 후 반환하기 전에 호출하지 않아야 합니다.

기본적으로, 이 클래스는 SAX 2.0 드라이버 작성자의 편의를 위해 IncrementalParser 인터페이스의 feed, close 및 reset 메서드를 사용하여 XMLReader 인터페이스의 parse 메서드를 구현합니다.

class xml.sax.xmlreader.**Locator**

SAX 이벤트를 문서 위치에 관련시키기 위한 인터페이스. 로케이터 객체는 DocumentHandler 메서드들을 호출하는 동안에만 유효한 결과를 반환합니다; 다른 때에는 결과를 예측할 수 없습니다. 정보가 없으면, 메서드는 `None`을 반환할 수 있습니다.

class xml.sax.xmlreader.**InputSource**(*system_id=None*)

엔티티를 읽기 위해 *`XMLReader`*에 필요한 정보의 캡슐화.

이 클래스는 공개 식별자, 시스템 식별자, 바이트 스트림 (문자 인코딩 정보도 포함할 수 있습니다) 및/또는 엔티티의 문자 스트림에 대한 정보를 포함할 수 있습니다.

응용 프로그램은 *`XMLReader.parse()`* 메서드에서 사용하고 EntityResolver.resolveEntity에서 반환하기 위해 이 클래스의 객체를 만듭니다.

*`InputSource`*는 응용 프로그램에 속하며, *`XMLReader`*는 응용 프로그램에서 전달된 *`InputSource`* 객체를 수정할 수는 없지만, 복사하여 수정할 수는 있습니다.

class xml.sax.xmlreader.**AttributesImpl**(*attrs*)

이것은 `Attributes` 인터페이스의 구현입니다 (*Attributes 인터페이스* 절을 참조하십시오). 이것은 `startElement()` 호출에서 요소 어트리뷰트를 나타내는 딕셔너리류 객체입니다. 가장 유용한 딕셔너리 연산 외에도, 인터페이스에서 설명하는 여러 가지 메서드를 지원합니다. 이 클래스의 객체는 입력기(reader)가 인스턴스화 해야 합니다; *attrs*는 어트리뷰트 이름에서 어트리뷰트 값으로의 매핑을 포함하는 딕셔너리류 객체여야 합니다.

class xml.sax.xmlreader.**AttributesNSImpl**(*attrs*, *qnames*)

`startElementNS()` 에 전달될 *`AttributesImpl`*의 이름 공간(namespace) 인식 변형입니다. *`AttributesImpl`*에서 파생되었지만, *namespaceURI*와 *localname*의 2-튜플로 구성된 어트리뷰트 이름을 이해합니다. 또한, 원본 문서에 나타나는 정규화 된 이름을 기대하는 여러 가지 메서드를 제공합니다. 이 클래스는 `AttributesNS` 인터페이스를 구현합니다 (*AttributesNS 인터페이스* 절을 참조하십시오).

21.12.1 XMLReader 객체

`XMLReader` 인터페이스는 다음과 같은 메서드를 지원합니다:

XMLReader.**parse**(*source*)

SAX 이벤트를 생성하면서 입력 소스를 처리합니다. *source* 객체는 시스템 식별자 (입력 소스를 식별하는 문자열 – 보통 파일 이름이나 URL), *`pathlib.Path`*, 경로류 객체 또는 *`InputSource`* 객체일 수 있습니다. *`parse()`*가 반환되면, 입력이 완전히 처리된 것이고 구문 분석기 객체를 파기하거나 재설정(reset)할 수 있습니다.

버전 3.5에서 변경: 문자 스트림 지원이 추가되었습니다.

버전 3.8에서 변경: 경로류 객체에 대한 지원이 추가되었습니다.

XMLReader.**getContentHandler**()

현재 *`ContentHandler`*를 반환합니다.

XMLReader.**setContentHandler**(*handler*)

현재 *ContentHandler*를 설정합니다. *ContentHandler*가 설정되지 않으면, 내용 이벤트가 버려집니다.

XMLReader.**getDTDHandler**()

현재 *DTDHandler*를 반환합니다.

XMLReader.**setDTDHandler**(*handler*)

현재 *DTDHandler*를 설정합니다. *DTDHandler*가 설정되지 않으면, DTD 이벤트가 버려집니다.

XMLReader.**getEntityResolver**()

현재 *EntityResolver*를 반환합니다.

XMLReader.**setEntityResolver**(*handler*)

현재 *EntityResolver*를 설정합니다. *EntityResolver*가 설정되지 않으면, 외부 엔티티를 결정하려고 할 때 엔티티에 대한 시스템 식별자가 열리게 되고, 사용할 수 없으면 실패합니다.

XMLReader.**getErrorHandler**()

현재 *ErrorHandler*를 반환합니다.

XMLReader.**setErrorHandler**(*handler*)

현재 에러 처리기를 설정합니다. *ErrorHandler*가 설정되지 않으면, 에러는 예외를 발생시키고, 경고는 인쇄됩니다.

XMLReader.**setLocale**(*locale*)

응용 프로그램이 에러와 경고에 대한 로케일을 설정하도록 합니다.

SAX 구문 분석기는 에러와 경고에 대한 지역화를 제공하지 않아도 됩니다; 그러나, 요청된 로케일을 지원할 수 없으면 SAX 예외를 발생시켜야 합니다. 응용 프로그램은 구문 분석 중에 로케일 변경을 요청할 수 있습니다.

XMLReader.**getFeature**(*featurename*)

기능 *featurename*의 현재 설정을 반환합니다. 기능이 인식되지 않으면, `SAXNotRecognizedException`이 발생합니다. 잘 알려진 기능 이름(featurename)은 모듈 *xml.sax.handler*에 나열되어 있습니다.

XMLReader.**setFeature**(*featurename*, *value*)

*featurename*을 *value*로 설정합니다. 기능이 인식되지 않으면, `SAXNotRecognizedException`가 발생합니다. 구문 분석기가 기능이나 해당 설정을 지원하지 않으면 *SAXNotSupportedException*이 발생합니다.

XMLReader.**getProperty**(*propertyname*)

속성 *propertyname*의 현재 설정을 반환합니다. 속성이 인식되지 않으면 `SAXNotRecognizedException`이 발생합니다. 잘 알려진 속성 이름은 모듈 *xml.sax.handler*에 나열되어 있습니다.

XMLReader.**setProperty**(*propertyname*, *value*)

*propertyname*을 *value*로 설정합니다. 속성이 인식되지 않으면 `SAXNotRecognizedException`이 발생합니다. 구문 분석기가 속성이나 해당 설정을 지원하지 않으면 *SAXNotSupportedException*이 발생합니다.

21.12.2 IncrementalParser 객체

IncrementalParser 인스턴스는 다음과 같은 추가 메서드를 제공합니다:

IncrementalParser.**feed**(*data*)

data 청크를 처리합니다.

IncrementalParser.**close**()

문서의 끝을 가정합니다. 끝에서만 확인할 수 있는 올바른 구성(well-formedness) 조건을 확인하고, 처리기를 호출하며 구문 분석 중에 할당된 자원을 정리할 수 있습니다.

IncrementalParser.**reset**()

이 메서드는 close가 호출된 후에 호출되어 새 문서를 구문 분석할 수 있도록 구문 분석기를 재설정합니다. reset을 호출하지 않고, close 후에 parse나 feed를 호출한 결과는 정의되지 않습니다.

21.12.3 Locator 객체

Locator 인스턴스는 다음 메서드를 제공합니다:

Locator.**getColumnNumber**()

현재 이벤트가 시작되는 열 번호를 반환합니다.

Locator.**getLineNumber**()

현재 이벤트가 시작되는 줄 번호를 반환합니다.

Locator.**getPublicId**()

현재 이벤트의 공개 식별자를 반환합니다.

Locator.**getSystemId**()

현재 이벤트의 시스템 식별자를 반환합니다.

21.12.4 InputSource 객체

InputSource.**setPublicId**(*id*)

이 *InputSource*의 공개 식별자를 설정합니다.

InputSource.**getPublicId**()

이 *InputSource*의 공개 식별자를 반환합니다.

InputSource.**setSystemId**(*id*)

이 *InputSource*의 시스템 식별자를 설정합니다.

InputSource.**getSystemId**()

이 *InputSource*의 시스템 식별자를 반환합니다.

InputSource.**setEncoding**(*encoding*)

이 *InputSource*의 문자 인코딩을 설정합니다.

인코딩은 XML 인코딩 선언에 허용되는 문자열이어야 합니다 (XML 권장 사항의 4.3.3 절을 참조하십시오).

*InputSource*에 문자 스트림도 포함되어 있으면 *InputSource*의 인코딩 어트리뷰트는 무시됩니다.

InputSource.**getEncoding**()

이 InputSource의 문자 인코딩을 가져옵니다.

InputSource.**setByteStream**(*bytefile*)

이 입력 소스의 바이트 스트림(바이너리 파일)을 설정합니다.

문자 스트림도 지정되어 있으면 SAX 구문 분석기는 이것을 무시하지만, URI 연결 자체를 여는 것보다 바이트 스트림을 먼저 사용합니다.

응용 프로그램이 바이트 스트림의 문자 인코딩을 알고 있으면, setEncoding 메서드로 설정해야 합니다.

InputSource.**getByteStream**()

이 입력 소스의 바이트 스트림을 가져옵니다.

getEncoding 메서드는 이 바이트 스트림의 문자 인코딩을 반환하거나, 알 수 없으면 None을 반환합니다.

`InputSource.setCharacterStream(charfile)`

이 입력 소스에 대한 문자 스트림(텍스트 파일)을 설정합니다.

문자 스트림이 지정되면 SAX 구문 분석기는 모든 바이트 스트림을 무시하고 시스템 식별자에 대한 URI 연결을 열려고 시도하지 않습니다.

`InputSource.getCharacterStream()`

이 입력 소스의 문자 스트림을 가져옵니다.

21.12.5 Attributes 인터페이스

`Attributes` 객체는 메서드 `copy()`, `get()`, `__contains__()`, `items()`, `keys()` 및 `values()`를 포함하는 매핑 프로토콜의 일부를 구현합니다. 다음과 같은 메서드도 제공됩니다:

`Attributes.getLength()`

어트리뷰트 수를 반환합니다.

`Attributes.getNames()`

어트리뷰트의 이름을 반환합니다.

`Attributes.getType(name)`

어트리뷰트 *name*의 유형을 반환합니다. 일반적으로 `'CDATA'`입니다.

`Attributes.getValue(name)`

어트리뷰트 *name*의 값을 반환합니다.

21.12.6 AttributesNS 인터페이스

이 인터페이스는 `Attributes` 인터페이스의 서브 형입니다 (*Attributes 인터페이스* 절을 참조하십시오). 그 인터페이스에서 지원하는 모든 메서드는 `AttributesNS` 객체에서도 사용 가능합니다.

다음과 같은 메서드도 제공됩니다:

`AttributesNS.getValueByQName(name)`

정규화된 이름(qualified name)의 값을 반환합니다.

`AttributesNS.getNameByQName(name)`

정규화된 *name*에 대한 `(namespace, localname)` 쌍을 반환합니다.

`AttributesNS.getQNameByName(name)`

`(namespace, localname)` 쌍에 대한 정규화된 이름을 반환합니다.

`AttributesNS.getQNames()`

모든 어트리뷰트의 정규화된 이름을 반환합니다.

21.13 xml.parsers.expat — Expat을 사용한 빠른 XML 구문 분석

> **참고**
>
> If you need to parse untrusted or unauthenticated data, see *XML security*.

`xml.parsers.expat` 모듈은 Expat 유효성 검사 없는(non-validating) XML 구문 분석기에 대한 파이썬 인터페이스입니다. 이 모듈은 XML 구문 분석기의 현재 상태를 나타내는 단일 확장형 `xmlparser`를 제공합니다. `xmlparser` 객체가 만들어진 후, 객체의 다양한 어트리뷰트를 처리기 함수로 설정할 수 있습니다. 그런 다음 XML 문서가 구문 분석기에 공급되면, XML 문서에 있는 문자 데이터와 마크업에 대해 처리기 함수가 호출됩니다.

이 모듈은 pyexpat 모듈을 사용하여 Expat 구문 분석기에 대한 액세스를 제공합니다. pyexpat 모듈의 직접적인 사용은 폐지되었습니다.

이 모듈은 하나의 예외와 하나의 형 객체를 제공합니다:

exception xml.parsers.expat.**ExpatError**

Expat이 에러를 보고할 때 발생하는 예외. Expat 에러 해석에 대한 자세한 정보는 섹션 *ExpatError 예외*를 참조하십시오.

exception xml.parsers.expat.**error**

*ExpatError*의 별칭.

xml.parsers.expat.**XMLParserType**

ParserCreate() 함수의 반환 값의 형.

xml.parsers.expat 모듈에는 두 가지 함수가 있습니다:

xml.parsers.expat.**ErrorString**(*errno*)

주어진 에러 번호 *errno*에 대한 설명 문자열을 반환합니다.

xml.parsers.expat.**ParserCreate**(*encoding=None*, *namespace_separator=None*)

새로운 xmlparser 객체를 만들고 반환합니다. 지정되면, *encoding*은 XML 데이터가 사용하는 인코딩의 이름을 지정하는 문자열이어야 합니다. Expat은 파이썬만큼 많은 인코딩을 지원하지 않으며, 인코딩 레퍼토리를 확장할 수 없습니다; UTF-8, UTF-16, ISO-8859-1 (Latin1) 및 ASCII를 지원합니다. *encoding*[1] 이 제공되면 문서의 묵시적이나 명시적 인코딩을 대체합니다.

Expat은 XML 이름 공간 처리를 선택적으로 수행할 수 있는데, *namespace_separator*에 값을 제공하는 것으로 활성화됩니다. 값은 한 문자 문자열이어야 합니다; 문자열의 길이가 잘못되면 *ValueError*가 발생합니다 (None은 생략과 같은 것으로 간주합니다). 이름 공간 처리가 활성화되면, 이름 공간에 속하는 엘리먼트 형 이름과 어트리뷰트 이름이 확장됩니다. 엘리먼트 처리기 StartElementHandler와 EndElementHandler 에 전달된 엘리먼트 이름은 이름 공간 URI, 이름 공간 구분 문자 및 이름의 로컬 부분의 이어붙이기입니다. 이름 공간 구분자가 0바이트(chr(0))이면 이름 공간 URI와 로컬 부분이 구분자 없이 연결됩니다.

예를 들어, *namespace_separator*가 스페이스 문자(' ')로 설정되고 다음 문서가 구문 분석되면:

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

StartElementHandler는 각 엘리먼트에 대해 다음 문자열을 수신합니다:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

pyexpat에서 사용하는 Expat 라이브러리의 제한으로 인해, 반환된 xmlparser 인스턴스는 단일 XML 문서를 구문 분석하는 데만 사용할 수 있습니다. 고유한 구문 분석기 인스턴스를 제공하려면 문서마다 ParserCreate를 호출하십시오.

더 보기

Expat XML 구문 분석기
Expat 프로젝트의 홈페이지.

[1] XML 출력에 포함된 인코딩 문자열은 적절한 표준을 준수해야 합니다. 예를 들어, "UTF-8"은 유효하지만, "UTF8"은 유효하지 않습니다. https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl 과 https://www.iana.org/assignments/character-sets/character-sets.xhtml 을 참조하십시오.

21.13.1 XMLParser 객체

xmlparser 객체에는 다음과 같은 메서드가 있습니다:

xmlparser.**Parse**(*data*[, *isfinal*])

문자열 *data*의 내용을 구문 분석하고, 구문 분석된 데이터를 처리하기 위해 적절한 처리기 함수를 호출합니다. 이 메서드에 대한 최종 호출에서 *isfinal*은 참이어야 합니다; 여러 파일을 제출하는 것이 아니라, 단일 파일을 여러 조각으로 나누어 구문 분석 할 수 있도록 합니다. *data*는 언제든지 빈 문자열이 될 수 있습니다.

xmlparser.**ParseFile**(*file*)

file 객체에서 읽은 XML 데이터를 구문 분석합니다. *file*은 더는 데이터가 없을 때 빈 문자열을 반환하는 read(nbytes) 메서드만 제공하면 됩니다.

xmlparser.**SetBase**(*base*)

선언에서 시스템 식별자에 있는 상대 URI를 결정하는 데 사용할 베이스를 설정합니다. 상대 식별자 결정은 응용 프로그램에 맡겨집니다: 이 값은 *base* 인자로 *ExternalEntityRefHandler()*, *NotationDeclHandler()* 및 *UnparsedEntityDeclHandler()* 함수에 전달됩니다.

xmlparser.**GetBase**()

*SetBase()*에 대한 이전 호출에 의해 설정된 베이스를 포함하는 문자열을 반환하거나, *SetBase()*가 호출되지 않았으면 None을 반환합니다.

xmlparser.**GetInputContext**()

현재 이벤트를 생성한 입력 데이터를 문자열로 반환합니다. 데이터는 텍스트를 포함하는 엔티티의 인코딩을 따릅니다. 이벤트 처리기가 활성화되지 않은 상태에서 호출될 때, 반환 값은 None입니다.

xmlparser.**ExternalEntityParserCreate**(*context*[, *encoding*])

부모 구문 분석기가 구문 분석한 내용이 참조하는 외부 구문 분석된 엔티티를 구문 분석하는 데 사용할 수 있는 "자식" 구문 분석기를 만듭니다. *context* 매개 변수는 아래 설명된 *ExternalEntityRefHandler()* 처리기 함수에 전달된 문자열이어야 합니다. 자식 구문 분석기는 *ordered_attributes*와 *specified_attributes*가 이 구문 분석기의 값으로 설정된 상태로 만들어집니다.

xmlparser.**SetParamEntityParsing**(*flag*)

매개 변수 엔티티(외부 DTD 부분 집합을 포함합니다)의 구문 분석을 제어합니다. 가능한 *flag* 값은 XML_PARAM_ENTITY_PARSING_NEVER, XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE 및 XML_PARAM_ENTITY_PARSING_ALWAYS입니다. 플래그 설정에 성공하면 참을 반환합니다.

xmlparser.**UseForeignDTD**([*flag*])

*flag*에 대해 참값(기본값)으로 이를 호출하면 Expat이 대체 DTD를 로드할 수 있도록 모든 인자를 *None*으로 *ExternalEntityRefHandler*를 호출하도록 합니다. 문서에 문서 형 선언이 포함되어 있지 않으면, *ExternalEntityRefHandler*는 여전히 호출되지만, *StartDoctypeDeclHandler*와 *EndDoctypeDeclHandler*는 호출되지 않습니다.

*flag*에 대해 거짓 값을 전달하면 참값을 전달한 이전 호출이 취소되지만, 그렇지 않으면 효과가 없습니다.

이 메서드는 *Parse()*나 *ParseFile()* 메서드가 호출되기 전에만 호출 할 수 있습니다; 이들 중 어느 것이라도 호출된 후에 호출하면 *code* 어트리뷰트가 errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING]로 설정된 *ExpatError*가 발생합니다.

xmlparser.**SetReparseDeferralEnabled**(*enabled*)

> **경고**
>
> Calling SetReparseDeferralEnabled(False) has security implications, as detailed below; please make sure to understand these consequences prior to using the SetReparseDeferralEnabled method.

Expat 2.6.0 introduced a security mechanism called "reparse deferral" where instead of causing denial of service through quadratic runtime from reparsing large tokens, reparsing of unfinished tokens is now delayed

by default until a sufficient amount of input is reached. Due to this delay, registered handlers may — depending of the sizing of input chunks pushed to Expat — no longer be called right after pushing new input to the parser. Where immediate feedback and taking over responsibility of protecting against denial of service from large tokens are both wanted, calling `SetReparseDeferralEnabled(False)` disables reparse deferral for the current Expat parser instance, temporarily or altogether. Calling `SetReparseDeferralEnabled(True)` allows re-enabling reparse deferral.

Note that *`SetReparseDeferralEnabled()`* has been backported to some prior releases of CPython as a security fix. Check for availability of *`SetReparseDeferralEnabled()`* using *`hasattr()`* if used in code running across a variety of Python versions.

Added in version 3.13.

`xmlparser.`**`GetReparseDeferralEnabled`**`()`

Returns whether reparse deferral is currently enabled for the given Expat parser instance.

Added in version 3.13.

`xmlparser` 객체에는 다음과 같은 어트리뷰트가 있습니다:

`xmlparser.`**`buffer_size`**

*`buffer_text`*가 참일 때 사용되는 버퍼의 크기. 이 어트리뷰트에 새로운 정숫값을 대입하여 새로운 버퍼 크기를 설정할 수 있습니다. 크기가 변경되면 버퍼가 플러시 됩니다.

`xmlparser.`**`buffer_text`**

이 값을 참으로 설정하면 `xmlparser` 객체가 Expat에서 반환한 텍스트 내용을 버퍼링하여 가능할 때마다 *`CharacterDataHandler()`* 콜백을 여러 번 호출하지 않도록 합니다. Expat은 일반적으로 모든 줄 끝에서 문자 데이터를 덩어리로 나누기 때문에 성능을 크게 향상할 수 있습니다. 이 어트리뷰트는 기본적으로 거짓이며, 언제든지 변경될 수 있습니다. 거짓일 때, 줄 넘김이 없는 데이터도 덩어리노 나뉠 수 있음에 유의하십시오.

`xmlparser.`**`buffer_used`**

*`buffer_text`*가 활성화되면, 버퍼에 저장된 바이트 수. 이 바이트열은 UTF-8로 인코딩된 텍스트를 나타냅니다. *`buffer_text`*가 거짓이면 이 어트리뷰트는 의미가 없습니다.

`xmlparser.`**`ordered_attributes`**

이 어트리뷰트를 0이 아닌 정수로 설정하면 어트리뷰트가 딕셔너리가 아닌 리스트로 보고됩니다. 어트리뷰트는 문서 텍스트에서 발견된 순서대로 표시됩니다. 각 어트리뷰트에 대해, 두 가지 리스트 항목이 표시됩니다: 어트리뷰트 이름과 어트리뷰트 값. (이 모듈의 이전 버전도 이 형식을 사용했습니다.) 기본적으로, 이 어트리뷰트는 거짓입니다; 언제든지 변경될 수 있습니다.

`xmlparser.`**`specified_attributes`**

0이 아닌 정수로 설정되면, 구문 분석기는 문서 인스턴스에 지정된 어트리뷰트만 보고하고 어트리뷰트 선언에서 파생된 어트리뷰트는 보고하지 않습니다. 이를 설정한 응용 프로그램은 XML 프로세서의 동작에 대한 표준을 준수하기 위해 선언에서 사용 가능한 추가 정보를 사용할 때 특히 주의해야 합니다. 기본적으로, 이 어트리뷰트는 거짓입니다; 언제든지 변경될 수 있습니다.

다음 어트리뷰트는 `xmlparser` 객체가 만난 가장 최근 에러와 관련된 값을 포함하며, 일단 `Parse()` 나 `ParseFile()` 에 대한 호출이 *`xml.parsers.expat.ExpatError`* 예외를 발생시켰을 때만 올바른 값을 갖습니다.

`xmlparser.`**`ErrorByteIndex`**

에러가 발생한 바이트 인덱스.

`xmlparser.`**`ErrorCode`**

문제를 지정하는 숫자 코드. 이 값은 *`ErrorString()`* 함수에 전달되거나, `errors` 객체에 정의된 상수 중 하나와 비교될 수 있습니다.

`xmlparser.`**`ErrorColumnNumber`**

에러가 발생한 열 번호.

`xmlparser.`**`ErrorLineNumber`**

에러가 발생한 줄 번호.

다음 어트리뷰트는 xmlparser 객체의 현재 구문 분석 위치와 관련된 값을 포함합니다. 구문 분석 이벤트를 보고하는 콜백 중에, 이들은 이벤트를 생성한 첫 번째 문자 시퀀스의 위치를 나타냅니다. 콜백 외부에서 호출되면, 표시된 위치는 마지막 구문 분석 이벤트 바로 다음입니다 (관련 콜백이 있는지에 관계없이).

xmlparser.**CurrentByteIndex**

구문 분석기 입력의 현재 바이트 인덱스.

xmlparser.**CurrentColumnNumber**

구문 분석기 입력의 현재 열 번호.

xmlparser.**CurrentLineNumber**

구문 분석기 입력의 현재 줄 번호.

설정할 수 있는 처리기 목록은 다음과 같습니다. xmlparser 객체 *o*에 처리기를 설정하려면, o.handlername = func를 사용하십시오. *handlername*은 다음 목록에서 취해야 하며, *func*는 올바른 수의 인자를 받아들이는 콜러블 객체여야 합니다. 달리 명시되지 않는 한, 인자는 모두 문자열입니다.

xmlparser.**XmlDeclHandler**(*version*, *encoding*, *standalone*)

XML 선언이 구문 분석될 때 호출됩니다. XML 선언은 XML 권장 사항의 해당 버전에 대한 (선택적) 선언, 문서 텍스트의 인코딩 및 선택적 "독립형 (standalone)" 선언입니다. *version*과 *encoding*은 문자열이며, *standalone*은 문서가 독립형으로 선언되면 1이 되고, 독립형이 아닌 것으로 선언되면 0이 됩니다, 또는 독립형 절이 생략되면 -1입니다. Expat 버전 1.95.0 이상에서만 사용할 수 있습니다.

xmlparser.**StartDoctypeDeclHandler**(*doctypeName*, *systemId*, *publicId*, *has_internal_subset*)

Expat이 문서 형 선언(<!DOCTYPE ...) 구문 분석을 시작할 때 호출됩니다. *doctypeName*은 제시된 대로 정확하게 제공됩니다. *systemId*와 *publicId* 매개 변수는 지정되면 시스템 (system)과 공용 (public) 식별자를, 생략되면 None을 제공합니다. 문서에 내부 문서 선언 부분집합이 포함되어 있으면 *has_internal_subset*은 참입니다. Expat 버전 1.2 이상이 필요합니다.

xmlparser.**EndDoctypeDeclHandler**()

Expat이 문서 형 선언 구문 분석을 완료할 때 호출됩니다. Expat 버전 1.2 이상이 필요합니다.

xmlparser.**ElementDeclHandler**(*name*, *model*)

각 엘리먼트 형 선언마다 한 번씩 호출됩니다. *name*은 엘리먼트 형의 이름이고, *model*은 콘텐츠 모델의 표현입니다.

xmlparser.**AttlistDeclHandler**(*elname*, *attname*, *type*, *default*, *required*)

엘리먼트 형에 대해 선언된 어트리뷰트마다 호출됩니다. 어트리뷰트 리스트 선언이 세 개의 어트리뷰트를 선언하면 이 처리기는 어트리뷰트마다 한 번씩 세 번 호출됩니다. *elname*은 선언이 적용되는 엘리먼트의 이름이고 *attname*은 선언된 어트리뷰트의 이름입니다. 어트리뷰트 형은 *type*으로 전달된 문자열입니다; 가능한 값은 'CDATA', 'ID', 'IDREF', … 입니다. *default*는 어트리뷰트가 문서 인스턴스에 의해 지정되지 않을 때 사용되는 어트리뷰트의 기본값을 제공하거나, 기본값이 없으면 None 입니다 (#IMPLIED 값). 문서 인스턴스에서 어트리뷰트가 필수면, *required*는 참입니다. Expat 버전 1.95.0 이상이 필요합니다.

xmlparser.**StartElementHandler**(*name*, *attributes*)

모든 엘리먼트의 시작에서 호출됩니다. *name*은 엘리먼트 이름을 포함하는 문자열이고, *attributes*는 엘리먼트 어트리뷰트입니다. *ordered_attributes*가 참이면, 이것은 리스트입니다 (자세한 설명은 *ordered_attributes*를 참조하십시오). 그렇지 않으면 이름을 값으로 매핑하는 딕셔너리입니다.

xmlparser.**EndElementHandler**(*name*)

모든 엘리먼트의 끝에서 호출됩니다.

xmlparser.**ProcessingInstructionHandler**(*target*, *data*)

모든 처리 명령어 (processing instruction)에서 호출됩니다.

xmlparser.**CharacterDataHandler**(*data*)

문자 데이터에 대해 호출됩니다. 일반 문자 데이터, CDATA 표시 내용 및 무시할 수 있는 공백에 대해 호출됩니다. 이들을 구별해야 하는 응용 프로그램은 *StartCdataSectionHandler*, *EndCdataSectionHandler* 및 *ElementDeclHandler* 콜백을 사용하여 필요한 정보를 수집할 수 있습니다. 문자 데이터는 짧더라도 덩어리로 묶일 수 있어서 *CharacterDataHandler()*에 대한 호출을

여러 번 받을 수 있음에 유의하십시오. 이를 피하려면 `buffer_text` 인스턴스 어프리뷰트를 `True`로 설정하십시오.

`xmlparser.`**`UnparsedEntityDeclHandler`**(*entityName*, *base*, *systemId*, *publicId*, *notationName*)

구문 분석되지 않은 (NDATA) 엔티티 선언에 대해 호출됩니다. 이것은 Expat 라이브러리 1.2 버전에만 존재합니다; 최신 버전의 경우 `EntityDeclHandler`를 대신 사용하십시오. (Expat 라이브러리의 하부 함수는 더는 사용되지 않는 것으로 선언되었습니다.)

`xmlparser.`**`EntityDeclHandler`**(*entityName*, *is_parameter_entity*, *value*, *base*, *systemId*, *publicId*, *notationName*)

모든 엔티티 선언에 대해 호출됩니다. 파라미터와 내부 엔티티의 경우, *value*는 엔티티의 선언된 내용을 제공하는 문자열입니다; 외부 엔티티의 경우 `None`이 됩니다. *notationName* 매개 변수는 구문 분석된 엔티티의 경우 `None`이고, 구문 분석되지 않은 엔티티의 경우는 표기법 이름입니다. 엔티티가 파라미터 엔티티인 경우 *is_parameter_entity*는 참이고, 일반 엔티티의 경우 거짓입니다 (대부분 응용 프로그램은 일반 엔티티만 고려하면 됩니다). Expat 라이브러리 버전 1.95.0부터 사용 가능합니다.

`xmlparser.`**`NotationDeclHandler`**(*notationName*, *base*, *systemId*, *publicId*)

표기법 선언에 대해 호출됩니다. *notationName*, *base*, *systemId* 및 *publicId*는 주어지면 문자열입니다. 공용 식별자를 생략하면, *publicId*는 `None`이 됩니다.

`xmlparser.`**`StartNamespaceDeclHandler`**(*prefix*, *uri*)

엘리먼트에 이름 공간 선언이 포함되어 있으면 호출됩니다. 이름 공간 선언은 선언이 배치된 엘리먼트에 대한 `StartElementHandler` 가 호출되기 전에 처리됩니다.

`xmlparser.`**`EndNamespaceDeclHandler`**(*prefix*)

이름 공간 선언이 포함된 엘리먼트에 대해 닫기 태그에 도달하면 호출됩니다. 이는 각 이름 공간 선언 스코프의 시작을 나타내기 위해 `StartNamespaceDeclHandler` 가 호출된 순서와 반대로 엘리먼트의 각 이름 공간 선언에 대해 한 번씩 호출됩니다. 이 처리기에 대한 호출은 엘리먼트의 끝을 위한 해당 `EndElementHandler` 이후에 수행됩니다.

`xmlparser.`**`CommentHandler`**(*data*)

주석에 대해 호출됩니다. *data*는 주석의 텍스트이며, 선행 `'<!--'`와 후행 `'-->'`를 제외합니다.

`xmlparser.`**`StartCdataSectionHandler`**()

CDATA 섹션의 시작에서 호출됩니다. CDATA 섹션의 구문적 시작과 종료를 식별할 수 있으려면 이것과 `EndCdataSectionHandler` 가 필요합니다.

`xmlparser.`**`EndCdataSectionHandler`**()

CDATA 섹션의 끝에서 호출됩니다.

`xmlparser.`**`DefaultHandler`**(*data*)

적용 가능한 처리기가 지정되지 않은 XML 문서의 모든 문자에 대해 호출됩니다. 이것은 보고될 수 있지만, 처리기가 제공되지 않은 구성의 일부인 문자를 의미합니다.

`xmlparser.`**`DefaultHandlerExpand`**(*data*)

이것은 `DefaultHandler()`와 같지만, 내부 엔티티의 확장을 막지 않습니다. 엔티티 참조는 기본 처리기로 전달되지 않습니다.

`xmlparser.`**`NotStandaloneHandler`**()

XML 문서가 독립형 문서로 선언되지 않은 경우 호출됩니다. 외부 부분 집합이나 파라미터 엔티티에 대한 참조가 있지만, XML 선언에서 XML 선언이 standalone을 `yes`로 설정하지 않은 경우에 발생합니다. 이 처리기가 `0`을 반환하면, 구문 분석기는 `XML_ERROR_NOT_STANDALONE` 에러를 발생시킵니다. 이 처리기가 설정되지 않으면, 이 조건에 대해 구문 분석기가 예외를 발생시키지 않습니다.

`xmlparser.`**`ExternalEntityRefHandler`**(*context*, *base*, *systemId*, *publicId*)

외부 엔티티에 대한 참조에 대해 호출됩니다. *base*는 `SetBase()`에 대한 이전 호출에서 설정한 현재 베이스입니다. 공용과 시스템 식별자, *systemId* 및 *publicId*는 지정되면 문자열입니다; 공용 식별자가 제공되지 않으면 *publicId*는 `None`이 됩니다. *context* 값은 불투명하며 아래 설명된 대로만 사용해야 합니다.

외부 엔티티를 구문 분석하려면, 이 처리기를 구현해야 합니다. `ExternalEntityParserCreate(context)`를 사용하여 서브 구문 분석기를 만들고, 적절한

콜백으로 초기화하고, 엔티티를 구문 분석해야 합니다. 이 처리기는 정수를 반환해야 합니다; 0을 반환하면, 구문 분석기는 XML_ERROR_EXTERNAL_ENTITY_HANDLING 에러를 발생시키고, 그렇지 않으면 구문 분석이 계속됩니다.

이 처리기가 제공되지 않으면, 외부 엔티티는 제공된다면 *DefaultHandler* 콜백에 의해 보고됩니다.

21.13.2 ExpatError 예외

ExpatError 예외에는 여러 가지 흥미로운 어트리뷰트가 있습니다:

ExpatError.**code**

특정 에러에 대한 Expat의 내부 에러 번호. *errors.messages* 딕셔너리는 이러한 에러 번호를 Expat의 에러 메시지에 매핑합니다. 예를 들면:

```
from xml.parsers.expat import ParserCreate, ExpatError, errors

p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatError as err:
    print("Error:", errors.messages[err.code])
```

errors 모듈은 에러 메시지 상수와 이러한 메시지를 에러 코드로 역 매핑하는 딕셔너리 *codes*도 제공합니다, 아래를 참조하십시오.

ExpatError.**lineno**

에러가 감지된 줄 번호. 첫 번째 줄은 1로 번호가 매겨집니다.

ExpatError.**offset**

에러가 발생한 줄에서의 문자 오프셋. 첫 번째 열의 번호는 0입니다.

21.13.3 예

다음 프로그램은 인자를 출력하기만 하는 세 개의 처리기를 정의합니다.

```
import xml.parsers.expat

# 3개의 처리기 함수
def start_element(name, attrs):
    print('Start element:', name, attrs)
def end_element(name):
    print('End element:', name)
def char_data(data):
    print('Character data:', repr(data))

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

이 프로그램의 출력은 다음과 같습니다:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

21.13.4 콘텐츠 모델 기술

콘텐츠 모델은 중첩된 튜플을 사용하여 기술됩니다. 각 튜플에는 네 가지 값이 있습니다: 형, 수량 지정자 (quantifier), 이름 및 자식 튜플. 자식은 단순히 추가의 콘텐츠 모델 기술입니다.

처음 두 필드의 값은 *xml.parsers.expat.model* 모듈에 정의된 상수입니다. 이 상수는 두 그룹으로 모을 수 있습니다: 모델 형 그룹과 수량 지정자 그룹.

모델 형 그룹의 상수는 다음과 같습니다:

xml.parsers.expat.model.**XML_CTYPE_ANY**

모델 이름으로 명명된 엘리먼트가 콘텐츠 모델 `ANY`를 갖도록 선언되었습니다.

xml.parsers.expat.model.**XML_CTYPE_CHOICE**

명명된 엘리먼트가 여러 옵션 중에서 선택할 수 있도록 합니다; 이것은 `(A | B | C)`와 같은 콘텐츠 모델에 사용됩니다.

xml.parsers.expat.model.**XML_CTYPE_EMPTY**

`EMPTY`로 선언된 엘리먼트는 이 모델 형을 갖습니다.

xml.parsers.expat.model.**XML_CTYPE_MIXED**

xml.parsers.expat.model.**XML_CTYPE_NAME**

xml.parsers.expat.model.**XML_CTYPE_SEQ**

일련의 모델이 차례로 나타나는 모델은 이 모델 형으로 표시됩니다. `(A, B, C)`와 같은 모델에 사용됩니다.

수량 지정자 그룹의 상수는 다음과 같습니다:

xml.parsers.expat.model.**XML_CQUANT_NONE**

수정자가 제공되지 않아서, `A`와 같이, 정확히 한 번만 나타날 수 있습니다.

xml.parsers.expat.model.**XML_CQUANT_OPT**

모델은 선택적입니다: `A?` 와 같이, 한 번만 나타나거나 전혀 나타나지 않을 수 있습니다.

xml.parsers.expat.model.**XML_CQUANT_PLUS**

모델은 한 번 이상 발생해야 합니다 (`A+`처럼).

xml.parsers.expat.model.**XML_CQUANT_REP**

`A*`와 같이, 모델은 0번 이상 나타나야 합니다.

21.13.5 Expat 에러 상수

xml.parsers.expat.errors 모듈에는 다음과 같은 상수가 제공됩니다. 이 상수는 에러가 발생했을 때 발생하는 `ExpatError` 예외 객체의 일부 어트리뷰트를 해석하는 데 유용합니다. 이전 버전과의 호환성을 위해, 상숫값은 숫자 에러 *code*가 아니라 에러 *message*이므로, *code* 어트리뷰트를 `errors.codes[errors.XML_ERROR_CONSTANT_NAME]`와 비교합니다.

`errors` 모듈에는 다음과 같은 어트리뷰트가 있습니다:

xml.parsers.expat.errors.**codes**
문자열 설명을 에러 코드에 매핑하는 딕셔너리.

Added in version 3.2.

xml.parsers.expat.errors.**messages**
숫자 에러 코드를 문자열 설명에 매핑하는 딕셔너리.

Added in version 3.2.

xml.parsers.expat.errors.**XML_ERROR_ASYNC_ENTITY**

xml.parsers.expat.errors.**XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF**
어트리뷰트 값의 엔티티 참조가 내부 엔티티 대신 외부 엔티티를 참조합니다.

xml.parsers.expat.errors.**XML_ERROR_BAD_CHAR_REF**
문자 참조가 XML에서 잘못된 문자를 가리킵니다 (예를 들어, 문자 0 또는 '&#0;').

xml.parsers.expat.errors.**XML_ERROR_BINARY_ENTITY_REF**
엔티티 참조가 표기법으로 선언된 엔티티를 참조해서, 구문 분석할 수 없습니다.

xml.parsers.expat.errors.**XML_ERROR_DUPLICATE_ATTRIBUTE**
시작 태그에서 어트리뷰트가 두 번 이상 사용되었습니다.

xml.parsers.expat.errors.**XML_ERROR_INCORRECT_ENCODING**

xml.parsers.expat.errors.**XML_ERROR_INVALID_TOKEN**
입력 바이트를 문자에 올바르게 대입할 수 없을 때 발생합니다; 예를 들어, UTF-8 입력 스트림의 NUL 바이트(값 0).

xml.parsers.expat.errors.**XML_ERROR_JUNK_AFTER_DOC_ELEMENT**
문서 엘리먼트 다음에 공백 이외의 것이 나타났습니다.

xml.parsers.expat.errors.**XML_ERROR_MISPLACED_XML_PI**
입력 데이터의 시작 이외의 곳에서 XML 선언이 발견되었습니다.

xml.parsers.expat.errors.**XML_ERROR_NO_ELEMENTS**
문서에 엘리먼트가 없습니다 (XML은 모든 문서에 정확히 하나의 최상위 엘리먼트가 포함되어야 함을 요구합니다).

xml.parsers.expat.errors.**XML_ERROR_NO_MEMORY**
Expat이 내부적으로 메모리를 할당하지 못했습니다.

xml.parsers.expat.errors.**XML_ERROR_PARAM_ENTITY_REF**
파라미터 엔티티 참조가 허용되지 않는 위치에서 발견되었습니다.

xml.parsers.expat.errors.**XML_ERROR_PARTIAL_CHAR**
입력에서 불완전한 문자가 발견되었습니다.

xml.parsers.expat.errors.**XML_ERROR_RECURSIVE_ENTITY_REF**
엔티티 참조가 같은 엔티티에 대한 다른 참조를 포함합니다; 어쩌면 다른 이름을 통해서, 그리고 어쩌면 간접적으로.

xml.parsers.expat.errors.**XML_ERROR_SYNTAX**
지정되지 않은 구문 에러를 만났습니다.

xml.parsers.expat.errors.**XML_ERROR_TAG_MISMATCH**
종료 태그가 가장 안쪽에 있는 시작 태그와 일치하지 않습니다.

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_TOKEN**
어떤 토큰이 (가령 시작 태그) 스트림이 끝나기 전에 닫히지 않았거나 다음 토큰을 만났습니다.

xml.parsers.expat.errors.**XML_ERROR_UNDEFINED_ENTITY**
정의되지 않은 엔티티를 참조했습니다.

xml.parsers.expat.errors.**XML_ERROR_UNKNOWN_ENCODING**
문서 인코딩을 Expat에서 지원하지 않습니다.

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_CDATA_SECTION**
CDATA 표시 섹션이 닫히지 않았습니다.

xml.parsers.expat.errors.**XML_ERROR_EXTERNAL_ENTITY_HANDLING**

xml.parsers.expat.errors.**XML_ERROR_NOT_STANDALONE**
문서가 XML 선언에서 자신을 "독립형"으로 선언했지만 구문 분석기가 독립형이 아니라고 결정했으며 `NotStandaloneHandler` 가 설정되고 `0`을 반환했습니다.

xml.parsers.expat.errors.**XML_ERROR_UNEXPECTED_STATE**

xml.parsers.expat.errors.**XML_ERROR_ENTITY_DECLARED_IN_PE**

xml.parsers.expat.errors.**XML_ERROR_FEATURE_REQUIRES_XML_DTD**
DTD 지원이 컴파일되어 있어야 하는 작업이 요청되었지만, Expat이 DTD 지원 없이 구성되었습니다. *xml.parsers.expat* 모듈의 표준 빌드에서 이것이 보고되어서는 안 됩니다.

xml.parsers.expat.errors.**XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING**
구문 분석이 시작된 후에 구문 분석이 시작되기 전에만 변경할 수 있는 동작 변경이 요청되었습니다. 이것은 (현재) `UseForeignDTD()` 에 의해서만 발생합니다.

xml.parsers.expat.errors.**XML_ERROR_UNBOUND_PREFIX**
이름 공간 처리가 활성화되었을 때 선언되지 않은 접두사가 발견되었습니다.

xml.parsers.expat.errors.**XML_ERROR_UNDECLARING_PREFIX**
문서가 접두사와 연관된 이름 공간 선언을 제거하려고 했습니다.

xml.parsers.expat.errors.**XML_ERROR_INCOMPLETE_PE**
파라미터 엔티티에 불완전한 마크업이 포함되어 있습니다.

xml.parsers.expat.errors.**XML_ERROR_XML_DECL**
문서에 문서 엘리먼트가 전혀 없습니다.

xml.parsers.expat.errors.**XML_ERROR_TEXT_DECL**
외부 엔티티에 있는 텍스트 선언을 구문 분석하는 중 에러가 발생했습니다.

xml.parsers.expat.errors.**XML_ERROR_PUBLICID**
허용되지 않는 문자가 공용 id에서 발견되었습니다.

xml.parsers.expat.errors.**XML_ERROR_SUSPENDED**
일시 중지된 구문 분석기에 작업이 요청되었지만, 허용되지 않습니다. 여기에는 추가 입력을 제공하거나 구문 분석기를 중지하려는 시도가 포함됩니다.

xml.parsers.expat.errors.**XML_ERROR_NOT_SUSPENDED**
구문 분석기가 일시 중지되지 않았을 때 구문 분석기를 재개하려고 했습니다.

xml.parsers.expat.errors.**XML_ERROR_ABORTED**
이것은 파이썬 응용 프로그램에 보고되어서는 안 됩니다.

xml.parsers.expat.errors.**XML_ERROR_FINISHED**
입력을 구문 분석하는 것을 종료한 구문 분석기에 작업을 요청했지만, 허용되지 않습니다. 여기에는 추가 입력을 제공하거나 구문 분석기를 중지하려는 시도가 포함됩니다.

xml.parsers.expat.errors.**XML_ERROR_SUSPEND_PE**

xml.parsers.expat.errors.**XML_ERROR_RESERVED_PREFIX_XML**
An attempt was made to undeclare reserved namespace prefix `xml` or to bind it to another namespace URI.

xml.parsers.expat.errors.**XML_ERROR_RESERVED_PREFIX_XMLNS**
An attempt was made to declare or undeclare reserved namespace prefix `xmlns`.

`xml.parsers.expat.errors.`**`XML_ERROR_RESERVED_NAMESPACE_URI`**

An attempt was made to bind the URI of one the reserved namespace prefixes `xml` and `xmlns` to another namespace prefix.

`xml.parsers.expat.errors.`**`XML_ERROR_INVALID_ARGUMENT`**

이것은 파이썬 응용 프로그램에 보고되어서는 안 됩니다.

`xml.parsers.expat.errors.`**`XML_ERROR_NO_BUFFER`**

이것은 파이썬 응용 프로그램에 보고되어서는 안 됩니다.

`xml.parsers.expat.errors.`**`XML_ERROR_AMPLIFICATION_LIMIT_BREACH`**

The limit on input amplification factor (from DTD and entities) has been breached.

`xml.parsers.expat.errors.`**`XML_ERROR_NOT_STARTED`**

The parser was tried to be stopped or suspended before it started.

Added in version 3.14.

CHAPTER 22

인터넷 프로토콜과 지원

이 장에서 설명하는 모듈은 인터넷 프로토콜을 구현하고 관련 기술을 지원합니다. 모두 파이썬으로 구현됩니다. 대부분 모듈은 시스템 의존적인 모듈 `socket`을 요구하는데, 현재 대부분의 대중적인 플랫폼에서 지원됩니다. 다음은 개요입니다:

22.1 webbrowser — 편리한 웹 브라우저 제어기

소스 코드: Lib/webbrowser.py

`webbrowser` 모듈은 웹 기반 문서를 사용자에게 표시할 수 있는 고수준 인터페이스를 제공합니다. 대부분은, 이 모듈의 `open()` 함수를 호출하면 올바른 작업이 수행됩니다.

유닉스에서, X11에서는 그래픽 브라우저가 선호되지만, 그래픽 브라우저를 사용할 수 없거나 X11 디스플레이를 사용할 수 없으면 텍스트 모드 브라우저가 사용됩니다. 텍스트 모드 브라우저가 사용되면, 사용자가 브라우저를 종료할 때까지 호출하는 프로세스가 블록 됩니다.

If the environment variable `BROWSER` exists, it is interpreted as the `os.pathsep`-separated list of browsers to try ahead of the platform defaults. When the value of a list part contains the string `%s`, then it is interpreted as a literal browser command line to be used with the argument URL substituted for `%s`; if the value is a single word that refers to one of the already registered browsers this browser is added to the front of the search list; if the part does not contain `%s`, it is simply interpreted as the name of the browser to launch.[1]

버전 3.14에서 변경: The `BROWSER` variable can now also be used to reorder the list of platform defaults. This is particularly useful on macOS where the platform defaults do not refer to command-line tools on `PATH`.

유닉스가 아닌 플랫폼에서, 또는 유닉스에서 원격 브라우저를 사용할 수 있을 때, 제어하는 프로세스는 사용자가 브라우저를 완료할 때까지 기다리지 않고, 원격 브라우저가 디스플레이에 자체 창을 유지하도록 허용합니다. 유닉스에서 원격 브라우저를 사용할 수 없으면, 제어하는 프로세스가 새 브라우저를 시작하고 기다립니다.

On iOS, the `BROWSER` environment variable, as well as any arguments controlling autoraise, browser preference, and new tab/window creation will be ignored. Web pages will *always* be opened in the user' s preferred browser, in a new tab, with the browser being brought to the foreground. The use of the `webbrowser` module on iOS requires the `ctypes` module. If `ctypes` isn' t available, calls to `open()` will fail.

스크립트 **webbrowser**는 모듈의 명령 줄 인터페이스로 사용될 수 있습니다. 인자로 URL을 받아들입니다. 다음과 같은 선택적 매개 변수를 받아들입니다:

[1] 전체 경로 없이 여기에서 명명된 실행 파일은 `PATH` 환경 변수에 지정된 디렉터리에서 검색됩니다.

-n, --new-window

가능하다면 새 브라우저 창에서 URL을 엽니다.

-t, --new-tab

새 브라우저 탭에서 URL을 엽니다.

당연히, 이 옵션들은 상호 배타적입니다. 사용 예:

```
python -m webbrowser -t "https://www.python.org"
```

가용성: not WASI, not Android.

다음 예외가 정의됩니다:

exception webbrowser.**Error**

브라우저 제어 에러가 일어날 때 발생하는 예외.

다음 함수가 정의됩니다:

webbrowser.**open**(*url*, *new=0*, *autoraise=True*)

기본 브라우저를 사용하여 *url*을 표시합니다. *new*가 0이면, 가능하다면 같은 브라우저 창에서 *url*이 열립니다. *new*가 1이면, 가능하다면 새 브라우저 창이 열립니다. *new*가 2이면, 가능하다면 새 브라우저 페이지("탭")가 열립니다. *autoraise*가 `True`이면, 가능하다면 창을 올립니다(raise) (이것은 많은 창 관리자에서 이 변수의 설정과 관계없이 일어납니다).

Returns `True` if a browser was successfully launched, `False` otherwise.

일부 플랫폼에서, 이 함수를 사용하여 파일명을 여는 것은 동작하고 운영 체제의 연결된 프로그램이 시작될 수 있습니다. 하지만, 이것은 지원되지도 이식성 있지도 않습니다.

인자 `url`로 감사 이벤트(*auditing event*) `webbrowser.open`을 발생시킵니다.

webbrowser.**open_new**(*url*)

가능하다면, 기본 브라우저의 새 창에서 *url*을 엽니다, 그렇지 않으면, 유일한 브라우저 창에서 *url*을 엽니다.

Returns `True` if a browser was successfully launched, `False` otherwise.

webbrowser.**open_new_tab**(*url*)

가능하다면, 기본 브라우저의 새 페이지("탭")에서 *url*을 엽니다, 그렇지 않으면 `open_new()`와 동등합니다.

Returns `True` if a browser was successfully launched, `False` otherwise.

webbrowser.**get**(*using=None*)

브라우저 유형 *using*에 대한 제어기 객체를 반환합니다. *using*이 `None`이면, 호출자의 환경에 적합한 기본 브라우저을 위한 제어기 객체를 반환합니다.

webbrowser.**register**(*name*, *constructor*, *instance=None*, **, preferred=False*)

브라우저 유형 *name*을 등록합니다. 일단 브라우저 유형이 등록되면, `get()` 함수는 해당 브라우저 유형에 대한 제어기를 반환할 수 있습니다. *instance*가 제공되지 않거나, `None`이면, 필요할 때 *constructor*가 매개 변수 없이 호출되어 인스턴스를 만듭니다. *instance*가 제공되면, *constructor*는 절대로 호출되지 않으며, `None`일 수 있습니다.

*preferred*를 `True`로 설정하면 이 브라우저를 인자 없이 `get()`을 호출할 때 선호되는 결과가 되도록 합니다. 그렇지 않으면, 이 엔트리 포인트는 `BROWSER` 변수를 설정하거나 선언한 처리기의 이름과 일치하는 비어 있지 않은 인자로 `get()`을 호출하려는 경우에만 유용합니다.

버전 3.7에서 변경: *preferred* 키워드 전용 매개 변수가 추가되었습니다.

여러 가지 브라우저 유형이 미리 정의되어 있습니다. 이 표는 `get()` 함수로 전달될 수 있는 유형 이름과 제어기 클래스의 해당 인스턴스 화를 제공합니다, 모두 이 모듈에서 정의됩니다.

유형 이름	클래스 이름	노트
'mozilla'	Mozilla('mozilla')	
'firefox'	Mozilla('mozilla')	
'epiphany'	Epiphany('epiphany')	
'kfmclient'	Konqueror()	(1)
'konqueror'	Konqueror()	(1)
'kfm'	Konqueror()	(1)
'opera'	Opera()	
'links'	GenericBrowser('links')	
'elinks'	Elinks('elinks')	
'lynx'	GenericBrowser('lynx')	
'w3m'	GenericBrowser('w3m')	
'windows-default'	WindowsDefault	(2)
'macosx'	MacOSXOSAScript('default')	(3)
'safari'	MacOSXOSAScript('safari')	(3)
'google-chrome'	Chrome('google-chrome')	
'chrome'	Chrome('chrome')	
'chromium'	Chromium('chromium')	
'chromium-browser'	Chromium('chromium-browser')	
'iosbrowser'	IOSBrowser	(4)

노트:

(1) "Konqueror"는 유닉스용 KDE 데스크톱 환경의 파일 관리자이며, KDE가 실행 중일 때만 의미가 있습니다. 신뢰성 있게 KDE를 탐지하는 방법이 있다면 좋을 것입니다; `KDEDIR` 변수로는 충분하지 않습니다. KDE 2에서 **konqueror** 명령을 사용할 때조차도 "kfm" 이라는 이름이 사용됨에 유의하십시오 — 구현이 Konqueror를 실행하기 위한 최상의 전략을 선택합니다.

(2) 윈도우 플랫폼에서만.

(3) 맥 OS 에서만.

(4) iOS 에서만.

Added in version 3.2: A new `MacOSXOSAScript` class has been added and is used on Mac instead of the previous `MacOSX` class. This adds support for opening browsers not currently set as the OS default.

Added in version 3.3: Chrome/Chromium에 대한 지원이 추가되었습니다.

버전 3.12에서 변경: Support for several obsolete browsers has been removed. Removed browsers include Grail, Mosaic, Netscape, Galeon, Skipstone, Iceape, and Firefox versions 35 and below.

버전 3.13에서 변경: iOS에 대한 지원이 추가되었습니다.

여기 몇 가지 간단한 예제가 있습니다:

```
url = 'https://docs.python.org/'

# 브라우저 창이 이미 열려 있으면, 새 탭에 URL을 엽니다.
webbrowser.open_new_tab(url)

# 새 창에 URL을 엽니다, 가능하다면 창을 올립니다.
webbrowser.open_new(url)
```

22.1.1 브라우저 제어기 객체

Browser controllers provide the *name* attribute, and the following three methods which parallel module-level convenience functions:

`controller.`**`name`**

System-dependent name for the browser.

controller.**open**(*url*, *new=0*, *autoraise=True*)

이 제어기가 처리하는 브라우저를 사용하여 *url*을 표시합니다. *new*가 1이면, 가능하다면 새 브라우저 창이 열립니다. *new*가 2이면, 가능하다면 새 브라우저 페이지("탭")가 열립니다.

controller.**open_new**(*url*)

가능하다면, 이 제어기가 처리하는 브라우저의 새 창에 *url*을 엽니다, 그렇지 않으면, 유일한 브라우저 창에 *url*을 엽니다. 별칭 `open_new()`.

controller.**open_new_tab**(*url*)

가능하다면, 이 제어기가 처리하는 브라우저의 새 페이지("탭")에 *url*을 엽니다, 그렇지 않으면 `open_new()`와 동등합니다.

22.2 wsgiref — WSGI 유틸리티와 참조 구현

소스 코드: Lib/wsgiref

WSGI(Web Server Gateway Interface)는 웹 서버 소프트웨어와 파이썬으로 작성된 웹 응용 프로그램 간의 표준 인터페이스입니다. 표준 인터페이스는 여러 웹 서버에서 WSGI를 지원하는 응용 프로그램을 쉽게 사용할 수 있도록 합니다.

웹 서버와 프로그래밍 프레임워크의 작성자만 WSGI 설계의 모든 세부 사항과 코너 케이스를 알 필요가 있습니다. 단지 WSGI 응용 프로그램을 설치하거나 기존 프레임워크를 사용하여 웹 응용 프로그램을 작성하기 위해, WSGI의 모든 세부 사항을 이해할 필요는 없습니다.

`wsgiref`는 웹 서버나 프레임워크에 WSGI 지원을 추가하는 데 사용할 수 있는, WSGI 명세의 참조 구현입니다. WSGI 환경 변수와 응답 헤더를 조작하는 유틸리티, WSGI 서버 구현을 위한 베이스 클래스, WSGI 응용 프로그램을 서비스하는 데모 HTTP 서버, 정적 형 검사를 위한 형 및 WSGI 서버와 응용 프로그램이 WSGI 명세(**PEP 3333**)에 부합하는지 확인하는 유효성 검사 도구를 제공합니다.

WSGI에 대한 더 자세한 정보 및 자습서와 기타 자원에 대한 링크는 wsgi.readthedocs.io를 참조하십시오.

22.2.1 wsgiref.util – WSGI 환경 유틸리티

이 모듈은 WSGI 환경으로 작업하기 위한 다양한 유틸리티 함수를 제공합니다. WSGI 환경은 **PEP 3333**에서 설명한 대로 HTTP 요청 변수를 포함하는 딕셔너리입니다. *environ* 매개 변수를 취하는 모든 함수는 WSGI 호환 딕셔너리가 제공될 것으로 기대합니다; 자세한 명세는 **PEP 3333**를 형 어노테이션에 사용할 수 있는 형 에일리어스는 `WSGIEnvironment`를 참조하십시오.

wsgiref.util.**guess_scheme**(*environ*)

environ 딕셔너리에서 `HTTPS` 환경 변수를 검사하여 `wsgi.url_scheme`이 "http"와 "https" 중 어느 것인지 추측합니다. 반환 값은 문자열입니다.

이 함수는 CGI 나 FastCGI와 같은 CGI와 유사한 프로토콜을 감싸는 게이트웨이를 만들 때 유용합니다. 일반적으로, 이러한 프로토콜을 제공하는 서버는 SSL을 통해 요청이 수신될 때 "1", "yes" 또는 "on" 값을 갖는 `HTTPS` 변수를 포함합니다. 따라서, 이 함수는 그런 값을 발견하면 "https"를 반환하고, 그렇지 않으면 "http"를 반환합니다.

wsgiref.util.**request_uri**(*environ*, *include_query=True*)

PEP 3333의 "URL 재구성" 절에 있는 알고리즘을 사용하여 전체 요청 URI를 반환합니다. 선택적으로 질의 문자열(query string)을 포함합니다. *include_query*가 거짓이면 질의 문자열은 결과 URI에 포함되지 않습니다.

wsgiref.util.**application_uri**(*environ*)

`PATH_INFO`와 `QUERY_STRING` 변수가 무시된다는 점을 제외하면 `request_uri()`와 유사합니다. 결과는 요청이 가리키는 응용 프로그램 객체의 기본 URI입니다.

wsgiref.util.**shift_path_info**(*environ*)

단일 이름을 PATH_INFO에서 SCRIPT_NAME로 이동하고 이름을 반환합니다. *environ* 딕셔너리는 그 자리에서 수정됩니다; 원본 PATH_INFO 나 SCRIPT_NAME를 그대로 유지해야 하면 사본을 사용하십시오.

PATH_INFO에 남아있는 경로 세그먼트가 없으면, None이 반환됩니다.

일반적으로, 이 루틴은 요청 URI 경로의 각 부분을 처리하는 데 사용됩니다, 예를 들어 경로를 일련의 딕셔너리 키로 취급합니다. 이 루틴은 전달된 환경을 수정하여 대상 URI에 있는 다른 WSGI 응용 프로그램을 호출하는 데 적합하게 만듭니다. 예를 들어, /foo에 WSGI 응용 프로그램이 있고, 요청 URI 경로가 /foo/bar/baz이고, /foo의 WSGI 응용 프로그램이 *shift_path_info()*를 호출하면 "bar" 문자열을 수신하고 전달할 환경이 /foo/bar에 있는 WSGI 응용 프로그램에 전달하기 적합하도록 갱신됩니다. 즉, SCRIPT_NAME는 /foo에서 /foo/bar로 변경되고, PATH_INFO는 /bar/baz에서 /baz로 변경됩니다.

PATH_INFO가 단지 "/" 일 때, 이 루틴은 빈 문자열을 반환하고 SCRIPT_NAME에 후행 슬래시를 추가합니다; 빈 경로 세그먼트는 일반적으로 무시되고, SCRIPT_NAME는 일반적으로 슬래시로 끝나지 않음에도 그렇게 합니다. 의도적인 동작입니다. 이 루틴을 사용하여 객체를 탐색할 때, 응용 프로그램이 /x로 끝나는 URI와 /x/로 끝나는 URI의 차이를 구별할 수 있도록 하기 위함입니다.

wsgiref.util.**setup_testing_defaults**(*environ*)

테스트 목적으로 *environ*를 뺀 한 기본값으로 갱신합니다.

이 루틴은 HTTP_HOST, SERVER_NAME, SERVER_PORT, REQUEST_METHOD, SCRIPT_NAME, PATH_INFO 및 모든 **PEP 3333**에서 정의된 wsgi.* 변수를 포함하여 WSGI에 필요한 다양한 매개 변수를 추가합니다. 기본값만 제공하며, 이 변수들에 대한 기존 설정을 대체하지 않습니다.

이 루틴은 WSGI 서버와 응용 프로그램의 단위 테스트가 더미 환경을 쉽게 설정하도록 하기 위한 것입니다. 데이터가 가짜이므로, 실제 WSGI 서버나 응용 프로그램에서 사용해서는 안 됩니다!

Example usage (see also *demo_app()* for another example):

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# 상대적으로 간단한 WSGI 응용 프로그램. setup_testing_defaults로 갱신된 후에 환경
# 딕셔너리를 인쇄할 것입니다.
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-8')]

    start_response(status, headers)

    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
           for key, value in environ.items()]
    return ret

with make_server('', 8000, simple_app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()
```

위의 환경 함수 외에도, *wsgiref.util* 모듈은 다음과 같은 기타 유틸리티를 제공합니다:

wsgiref.util.**is_hop_by_hop**(*header_name*)

'header_name' 이 **RFC 2616**가 정의하고 있는 HTTP/1.1 "Hop-by-Hop" 헤더면 True를 반환합니다.

class wsgiref.util.**FileWrapper**(*filelike*, *blksize=8192*)

파일류 객체를 이터레이터로 변환하는 데 사용되는 *wsgiref.types.FileWrapper* 프로토콜의 구체적 구현. 결과 객체는 이터러블입니다. 객체가 이터레이트될 때, 선택적인 *blksize* 매개 변수는

산출할 바이트열을 얻기 위해 *filelike* 객체의 `read()` 메서드에 반복적으로 전달됩니다. `read()` 가 빈 바이트열을 반환하면 이터레이션이 종료되고 다시 시작할 수 없습니다.

*filelike*에 `close()` 메서드가 있으면, 반환된 객체에도 `close()` 메서드가 있고, 호출될 때 *filelike* 객체의 `close()` 메서드를 호출합니다.

사용 예:

```
from io import StringIO
from wsgiref.util import FileWrapper

# 파일류 객체로 StringIO 버퍼를 사용하고 있습니다
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print(chunk)
```

버전 3.11에서 변경: `__getitem__()` 메서드 지원은 제거되었습니다.

22.2.2 `wsgiref.headers` – WSGI 응답 헤더 도구

이 모듈은 매핑류 인터페이스를 사용하여 WSGI 응답 헤더를 편리하게 조작할 수 있는 클래스 *Headers* 하나를 제공합니다.

class `wsgiref.headers.`**`Headers`**([*headers*])

PEP 3333에 설명된 것처럼 헤더 이름/값 튜플의 리스트이어야 하는 *headers*를 감싸는 매핑류 객체를 만듭니다. *headers*의 기본값은 빈 리스트입니다.

Headers 객체는 `__getitem__()`, *get()*, `__setitem__()`, *setdefault()*, `__delitem__()` 및 `__contains__()` 를 포함한 일반적인 매핑 연산을 지원합니다. 이러한 각 메서드에 대해, 키는 헤더 이름(대소 문자를 구분하지 않습니다)이며, 값은 해당 헤더 이름과 연관된 첫 번째 값입니다. 헤더를 설정하면 해당 헤더의 기존 값이 삭제된 다음, 감싸진 헤러 리스트의 끝에 새 값이 추가됩니다. 헤더의 기존 순서는 일반적으로 유지되며, 감싸진 리스트의 끝에 새 헤더가 추가됩니다.

딕셔너리와 달리, *Headers* 객체는 감싸진 헤더 리스트에 없는 키를 가져오거나 삭제하려고 하면 에러를 발생시키지 않습니다. 존재하지 않는 헤더를 가져오려고 하면 `None`을 반환하고, 존재하지 않는 헤더를 삭제하면 아무것도 하지 않습니다.

Headers 객체는 `keys()`, `values()` 및 `items()` 메서드도 지원합니다. `keys()` 와 `items()` 에 의해 반환된 리스트에는 다중-값 헤더가 있으면 같은 키가 두 번 이상 포함될 수 있습니다. *Headers* 객체의 `len()` 는 `items()` 의 길이와 같고, 이는 감싸진 헤더 리스트의 길이와 같습니다. 실제로, `items()` 메서드는 단지 감싸진 헤더 리스트의 복사본을 반환합니다.

Headers 객체에서 `bytes()` 를 호출하면 HTTP 응답 헤더로 전송하기에 적합한 포맷된 바이트열이 반환됩니다. 각 헤더는 콜론과 공백으로 구분된 값과 함께 줄에 들어갑니다. 각 줄은 캐리지 리턴과 줄넘김으로 끝나며, 바이트열은 빈 줄로 끝납니다.

Headers 객체는 매핑 인터페이스와 포매팅 기능 외에도, 다중-값 헤더를 조회하거나 추가하고, MIME 파라미터가 있는 헤더를 추가하기 위해 다음과 같은 메서드를 제공합니다:

`get_all`(*name*)

주어진 이름의 헤더에 대한 모든 값의 리스트를 반환합니다.

반환된 리스트는 원래 헤더 리스트에 나타나거나 이 인스턴스에 추가된 순서대로 정렬되고, 중복을 포함할 수 있습니다. 삭제되고 다시 삽입된 필드는 항상 헤더 리스트의 끝에 추가됩니다. 주어진 이름의 필드가 존재하지 않으면, 빈 리스트를 반환합니다.

`add_header`(*name*, *value*, ***_params*)

키워드 인자로 지정되는 선택적 MIME 파라미터와 함께 헤더를 추가합니다 (다중-값 가능).

*name*은 추가할 헤더 필드입니다. 키워드 인자는 헤더 필드에 대한 MIME 파라미터를 설정하는 데 사용될 수 있습니다. 각 파라미터는 문자열이나 `None` 이어야 합니다. 파라미터 이름의 밑줄은 대시로 변환됩니다; 파이썬 식별자에서는 대시가 유효하지 않지만 많은 MIME 파라미터

이름에는 대시가 포함되기 때문입니다. 파라미터값이 문자열이면 `name="value"` 형식으로 헤더 값 파라미터에 추가됩니다. `None`이면 파라미터 이름 만 추가됩니다. (이것은 값이 없는 MIME 파라미터에 사용됩니다.) 사용 예:

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

위의 코드는 다음과 같은 헤더를 추가합니다:

```
Content-Disposition: attachment; filename="bud.gif"
```

버전 3.5에서 변경: *headers* 매개 변수는 선택적입니다.

22.2.3 `wsgiref.simple_server` – 간단한 WSGI HTTP 서버

이 모듈은 WSGI 응용 프로그램을 서빙하는 간단한 HTTP 서버(`http.server` 기반)를 구현합니다. 각 서버 인스턴스는 주어진 호스트와 포트에서 단일 WSGI 응용 프로그램을 서빙합니다. 단일 호스트와 포트에서 여러 응용 프로그램을 서빙하려면, `PATH_INFO`를 구문 분석하여 각 요청에 대해 호출할 응용 프로그램을 선택하는 WSGI 응용 프로그램을 만들어야 합니다. (예를 들어, `wsgiref.util`의 `shift_path_info()` 함수를 사용해서.)

`wsgiref.simple_server.`**`make_server`**(*host, port, app, server_class=WSGIServer, handler_class=WSGIRequestHandler*)

host 와 *port*에서 수신을 기다리고, *app*에 대한 연결을 수락하는 새 WSGI 서버를 만듭니다. 반환 값은 제공된 *server_class*의 인스턴스이며, 지정된 *handler_class*를 사용하여 요청을 처리합니다. *app*는 **PEP 3333**에 정의된 WSGI 응용 프로그램 객체여야 합니다.

사용 예:

```
from wsgiref.simple_server import make_server, demo_app

with make_server('', 8000, demo_app) as httpd:
    print("Serving HTTP on port 8000...")

    # 프로세스를 죽일 때까지 요청에 응답합니다
    httpd.serve_forever()

    # 대안: 하나의 요청을 처리한 다음 종료합니다
    httpd.handle_request()
```

`wsgiref.simple_server.`**`demo_app`**(*environ, start_response*)

이 함수는 작지만 완벽한 WSGI 응용 프로그램인데, "Hello world!" 메시지와 *environ* 매개 변수에 제공된 키/값 쌍 목록이 포함된 텍스트 페이지를 반환합니다. WSGI 서버(가령 `wsgiref.simple_server`)가 간단한 WSGI 응용 프로그램을 올바르게 실행할 수 있는지 확인하는 데 유용합니다.

start_response 콜러블은 `StartResponse` 프로토콜을 따라야 합니다.

class `wsgiref.simple_server.`**`WSGIServer`**(*server_address, RequestHandlerClass*)

`WSGIServer` 인스턴스를 만듭니다. *server_address*는 `(host,port)` 튜플이어야 하며, *RequestHandlerClass*는 `http.server.BaseHTTPRequestHandler`의 서브 클래스여야 하는데, 요청을 처리하는 데 사용됩니다.

`make_server()` 함수가 모든 세부 사항을 처리할 수 있으므로, 일반적으로 이 생성자를 호출할 필요가 없습니다.

`WSGIServer`는 `http.server.HTTPServer`의 서브 클래스이므로, 모든 메서드(가령 `serve_forever()`와 `handle_request()`)를 사용할 수 있습니다. `WSGIServer`는 또한 다음과 같은 WSGI 전용 메서드를 제공합니다:

`set_app`(*application*)

콜러블 *application*을 요청을 수신하는 WSGI 응용 프로그램으로 설정합니다.

get_app()

현재 설정되어있는 응용 프로그램 콜러블을 반환합니다.

그러나 일반적으로, 이러한 추가 메서드를 사용할 필요가 없는데, *set_app()*는 일반적으로 *make_server()*에서 호출되고, *get_app()*은 주로 요청 처리기 인스턴스의 필요를 위해 존재하기 때문입니다

class wsgiref.simple_server.**WSGIRequestHandler**(*request*, *client_address*, *server*)

지정된 *request* (즉, 소켓), *client_address* (`(host,port)` 튜플) 및 *server* (*WSGIServer* 인스턴스)를 위한 HTTP 처리기를 만듭니다.

이 클래스의 인스턴스를 직접 만들 필요는 없습니다; *WSGIServer* 객체가 필요에 따라 자동으로 만듭니다. 그러나, 이 클래스를 서브 클래싱하여 *handler_class*로 *make_server()* 함수에 제공할 수 있습니다. 서브 클래스에서 재정의하는데 적합한 메서드들은 다음과 같습니다:

get_environ()

요청에 대한 *WSGIEnvironment* 딕셔너리를 반환합니다. 기본 구현은 *WSGIServer* 객체의 `base_environ` 딕셔너리 어트리뷰트의 내용을 복사한 다음, HTTP 요청으로부터 온 다양한 헤더를 추가합니다. 이 메서드를 호출할 때마다 **PEP 3333**에 지정된 CGI 환경 변수를 모두 포함하는 새 딕셔너리를 반환해야 합니다.

get_stderr()

`wsgi.errors` 스트림으로 사용해야 하는 객체를 반환합니다. 기본 구현은 단지 `sys.stderr`를 반환합니다.

handle()

HTTP 요청을 처리합니다. 기본 구현은 *wsgiref.handlers* 클래스를 사용하여 실제 WSGI 응용 프로그램 인터페이스를 구현하는 처리기 인스턴스를 만듭니다.

22.2.4 wsgiref.validate — WSGI 적합성 검사기

새로운 WSGI 응용 프로그램 객체, 프레임워크, 서버 또는 미들웨어를 만들 때, *wsgiref.validate*를 사용하여 새 코드의 적합성을 확인하는 것이 유용할 수 있습니다. 이 모듈은 WSGI 서버나 게이트웨이와 WSGI 응용 프로그램 객체 사이의 통신을 검증하는 WSGI 응용 프로그램 객체를 만드는 함수를 제공하여, 양측의 프로토콜 준수 여부를 검사할 수 있도록 합니다.

이 유틸리티는 완전한 **PEP 3333** 적합성을 보장하지는 않습니다; 이 모듈에서 에러가 없다고 해서 에러가 존재하지 않는다는 것을 의미하지는 않습니다. 그러나, 이 모듈에서 오류가 발생하면, 서버나 응용 프로그램이 100% 적합하지 않다는 것은 사실상 확실합니다.

이 모듈은 Ian Bicking의 "Python Paste" 라이브러리의 `paste.lint` 모듈을 기반으로 합니다.

wsgiref.validate.**validator**(*application*)

application 감싸고 새 WSGI 응용 프로그램 객체를 반환합니다. 반환된 응용 프로그램은 모든 요청을 원래 *application*으로 전달하고, *application*과 이를 호출하는 서버가 모두 WSGI 명세와 **RFC 2616**를 준수하는지 확인합니다.

탐지된 모든 부적합 결과는 *AssertionError*를 일으킵니다; 그러나 이러한 에러를 처리하는 방법은 서버에 따라 다릅니다. 예를 들어, *wsgiref.simple_server*와 *wsgiref.handlers*를 기반으로 하는 다른 (에러 처리 메서드를 재정의해서 다른 작업을 수행하지 않는) 서버는 단순히 에러가 발생했다는 메시지를 출력하고 `sys.stderr` 나 기타 에러 스트림으로 트레이스백을 덤프합니다.

이 래퍼는 의심스럽기는 하지만 **PEP 3333**에서 실제로 금지되지 않을 수도 있는 동작을 나타내도록 *warnings* 모듈을 사용하여 출력을 생성할 수도 있습니다. 파이썬 명령 줄 옵션이나 *warnings* API를 사용해서 억제되지 않는 한, 그러한 경고는 `sys.stderr`(같은 객체가 아니라면 `wsgi.errors`가 아닙니다)에 기록됩니다.

사용 예:

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# 의도적으로 표준을 준수하지 않는 콜러블 객체이므로,
# 적합성 검사기가 중단됩니다
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # 이것은 중단하게 되는데, 목록을 반환해야 하기 때문입니다,
    # 적합성 검사기가 알려줄 것입니다
    return b"Hello World"

# 이것은 적합성 검사기로 감싼 응용 프로그램입니다
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000....")
    httpd.serve_forever()
```

22.2.5 wsgiref.handlers – 서버/게이트웨이 베이스 클래스

이 모듈은 WSGI 서버와 게이트웨이를 구현하기 위한 베이스 처리기 클래스를 제공합니다. 이러한 베이스 클래스는 입력, 출력 및 에러 스트림과 함께 CGI와 유사한 환경이 제공되는 한, WSGI 응용 프로그램과 통신하는 대부분 작업을 처리합니다.

class wsgiref.handlers.**CGIHandler**

sys.stdin, sys.stdout, sys.stderr 및 os.environ를 통한 CGI 기반 호출. 이것은 WSGI 응용 프로그램이 있고 CGI 스크립트로 실행하려고 할 때 유용합니다. CGIHandler().run(app) 을 호출하기만 하면 됩니다. 여기서 app은 호출할 WSGI 응용 프로그램 객체입니다.

이 클래스는 wsgi.run_once를 참으로, wsgi.multithread를 거짓으로, wsgi.multiprocess를 참으로 설정하고, 필요한 CGI 스트림과 환경을 얻기 위해 항상 *sys*와 *os*를 사용하는 *BaseCGIHandler*의 서브 클래스입니다.

class wsgiref.handlers.**IISCGIHandler**

config allowPathInfo 옵션 (IIS>=7) 나 metabase allowPathInfoForScriptMappings (IIS<7) 를 설정하지 않고도, Microsoft IIS 웹 서버에 배포할 때 사용할 수 있는 *CGIHandler*의 특수한 대안.

기본적으로, IIS는 앞에 SCRIPT_NAME이 중복된 PATH_INFO를 제공하므로, 라우팅을 구현하려는 WSGI 응용 프로그램에 문제가 발생합니다. 이 처리기는 중복된 경로를 제거합니다.

IIS가 올바른 PATH_INFO를 전달하도록 구성할 수 있지만, PATH_TRANSLATED가 잘못되는 다른 버그가 발생합니다. 다행히도 이 변수는 거의 사용되지 않으며 WSGI에서 보장하지 않습니다. 그러나 IIS<7에서는 설정이 가상 호스트 수준에서만 이루어지므로, 다른 모든 스크립트 매핑에 영향을 미치며, 그중 많은 것들이 PATH_TRANSLATED 버그에 노출되면 망가집니다. 이러한 이유로 IIS<7은 거의 수정해서 배포되지 않습니다 (아직도 UI가 없어서 IIS7 조차도 거의 사용하지 않습니다.).

CGI 코드가 옵션이 설정되었는지를 알 수 있는 방법이 없으므로, 별도의 처리기 클래스가 제공됩니다. *CGIHandler*와 같은 방식으로 사용됩니다. 즉, IISCGIHandler().run(app) 을 호출합니다. 여기서 app은 호출할 WSGI 응용 프로그램 객체입니다.

Added in version 3.2.

class wsgiref.handlers.**BaseCGIHandler**(*stdin, stdout, stderr, environ, multithread=True, multiprocess=False*)

*CGIHandler*와 유사하지만, *sys* 와 *os* 모듈을 사용하는 대신, CGI 환경과 I/O 스트림이 명시적으로 지정됩니다. *multithread* 와 *multiprocess* 값은 처리기 인스턴스가 실행하는 모든 응용 프로그램에 대한 wsgi.multithread 와 wsgi.multiprocess 플래그를 설정하는 데 사용됩니다.

이 클래스는 HTTP "오리진 서버" 이외의 소프트웨어에서 사용하기 위한 *SimpleHandler*의 서브 클래스입니다. HTTP 상태를 보내기 위해 Status: 헤더를 사용하는 게이트웨이 프로토콜 구현(가령

CGI, FastCGI, SCGI 등)을 작성하는 경우 `SimpleHandler` 대신 이것을 서브 클래싱하고 싶을 겁니다.

class wsgiref.handlers.**SimpleHandler**(*stdin, stdout, stderr, environ, multithread=True, multiprocess=False*)

`BaseCGIHandler`와 유사하지만, HTTP 오리진 서버에 사용하도록 설계되었습니다. HTTP 서버 구현을 작성하는 경우 `BaseCGIHandler` 대신 이것을 서브 클래싱하고 싶을 겁니다.

이 클래스는 `BaseHandler`의 서브 클래스입니다. `__init__()`, `get_stdin()`, `get_stderr()`, `add_cgi_vars()`, `_write()` 및 `_flush()` 메서드를 재정의하여 명시적으로 환경과 스트림을 생성자를 통해 설정하는 것을 지원합니다. 제공된 환경과 스트림은 `stdin`, `stdout`, `stderr` 및 `environ` 어트리뷰트에 저장됩니다.

*stdout*의 `write()` 메서드는 `io.BufferedIOBase`처럼 각 덩어리 전체를 기록해야 합니다.

class wsgiref.handlers.**BaseHandler**

이것은 WSGI 응용 프로그램을 실행하기 위한 추상 베이스 클래스입니다. 원칙적으로 여러 요청에 대해 재사용할 수 있는 서브 클래스를 만들 수 있지만, 각 인스턴스는 단일 HTTP 요청을 처리합니다.

`BaseHandler` 인스턴스에는 외부 사용을 위한 하나의 메서드 만 있습니다:

run(*app*)

지정된 WSGI 응용 프로그램인 *app*을 실행합니다.

다른 모든 `BaseHandler` 메서드는 응용 프로그램을 실행하는 과정에서 이 메서드에 의해 호출되므로, 주로 과정을 사용자 정의하기 위해 존재합니다.

다음 메서드는 서브 클래스에서 반드시 재정의되어야 합니다:

_write(*data*)

클라이언트로의 전송을 위해 바이트열 *data*를 버퍼링합니다. 이 메서드가 실제로 데이터를 전송해도 상관없습니다; `BaseHandler`는 쓰기와 플러시 연산을 분리하여 하부 시스템에 실제로 이러한 구분이 있을 때 효율성을 높입니다.

_flush()

버퍼링 된 데이터를 클라이언트로 전송하도록 강제합니다. 이 메서드가 아무 일도 하지 않아도 상관없습니다 (즉, `_write()`가 실제로 데이터를 보낸다면).

get_stdin()

현재 처리 중인 요청의 `wsgi.input`으로 사용하기에 적합한 `InputStream`과 호환되는 객체를 반환합니다.

get_stderr()

현재 처리 중인 요청의 `wsgi.errors`로 사용하기에 적합한 `ErrorStream`과 호환되는 객체를 반환합니다.

add_cgi_vars()

현재 요청에 대한 CGI 변수를 `environ` 어트리뷰트에 삽입합니다.

재정의하고 싶을 수 있는 다른 메서드와 어트리뷰트는 다음과 같습니다. 그러나, 이 목록은 요약에 지나지 않고, 재정의할 수 있는 모든 메서드가 포함되어 있지 않습니다. 사용자 정의된 `BaseHandler` 서브 클래스를 작성하기 전에 독스트링과 소스 코드를 참조하여 추가 정보를 얻어야 합니다.

WSGI 환경을 사용자 정의하기 위한 어트리뷰트와 메서드:

wsgi_multithread

`wsgi.multithread` 환경 변수에 사용될 값. `BaseHandler`에서는 기본값이 참이지만, 다른 서브 클래스는 다른 기본값을 가질 수 있습니다 (또는 생성자에 의해 설정될 수 있습니다).

wsgi_multiprocess

`wsgi.multiprocess` 환경 변수에 사용될 값. `BaseHandler`에서는 기본값이 참이지만, 다른 서브 클래스는 다른 기본값을 가질 수 있습니다 (또는 생성자에 의해 설정될 수 있습니다).

wsgi_run_once

wsgi.run_once 환경 변수에 사용될 값. *BaseHandler*에서는 기본값이 거짓이지만, *CGIHandler*는 기본적으로 참으로 설정합니다.

os_environ

모든 요청의 WSGI 환경에 포함될 기본 환경 변수. 기본적으로, *wsgiref.handlers*를 임포트 한 시점의 os.environ 사본이지만, 서브 클래스는 클래스나 인스턴스 수준에서 자체적으로 만들 수 있습니다. 기본값이 여러 클래스와 인스턴스 간에 공유되므로, 딕셔너리는 읽기 전용으로 간주해야 합니다.

server_software

origin_server 어트리뷰트가 설정된 경우, 이 어트리뷰트의 값은 기본 SERVER_SOFTWARE WSGI 환경 변수를 설정하는 데 사용되고, HTTP 응답의 기본 Server: 헤더를 설정하는 데도 사용됩니다. HTTP 오리진 서버가 아닌 처리기(가령 *BaseCGIHandler* 와 *CGIHandler*)에서는 무시됩니다.

버전 3.3에서 변경: "Python" 이라는 용어는 "CPython", "Jython" 등과 같은 구현 특정 용어로 대체됩니다.

get_scheme()

현재의 요청에 사용되고 있는 URL 스킴을 반환합니다. 기본 구현은 *wsgiref.util*의 guess_scheme() 함수를 사용하여, 현재 요청의 environ 변수를 기반으로, 스킴이 "http" 와 "https" 중 어느 것인지 추측합니다.

setup_environ()

environ 어트리뷰트를 완전히 채워진 WSGI 환경으로 설정합니다. 기본 구현에서는 위의 모든 메서드와 어트리뷰트에 더해 *get_stdin()*, *get_stderr()* 및 *add_cgi_vars()* 메서드와 *wsgi_file_wrapper* 어트리뷰트를 모두 사용합니다. *origin_server* 어트리뷰트가 참이고 *server_software* 어트리뷰트가 설정된 경우 SERVER_SOFTWARE 키가 없으면 삽입합니다.

예외 처리를 사용자 정의하기 위한 메서드와 어트리뷰트:

log_exception(*exc_info*)

exc_info 튜플을 서버 로그에 기록합니다. *exc_info*는 (type, value, traceback) 튜플입니다. 기본 구현은 요청의 wsgi.errors 스트림에 트레이스백을 쓰고 플러시 합니다. 서브 클래스는 이 메서드를 재정의해서, 형식을 변경하거나 출력의 대상을 바꾸거나, 관리자에게 트레이스백을 메일로 보내거나, 적절한 것으로 생각되는 다른 액션을 수행할 수 있습니다.

traceback_limit

기본 *log_exception()* 메서드에 의해 출력되는 트레이스백에 포함하는 프레임의 최대 수. None이면, 모든 프레임이 포함됩니다.

error_output(*environ*, *start_response*)

이 메서드는 사용자를 위한 에러 페이지를 생성하는 WSGI 응용 프로그램입니다. 헤더가 클라이언트에 전송되기 전에 오류가 발생할 때만 호출됩니다.

이 메서드는 sys.exception() 을 사용하여 현재 에러에 액세스할 수 있으며, 호출할 때 해당 정보를 *start_response*로 전달해야 합니다 (**PEP 3333**의 "에러 처리" 절에서 설명하듯이). 특히, *start_response* 콜러블은 *StartResponse* 프로토콜을 따라야 합니다.

기본 구현은 *error_status*, *error_headers* 및 *error_body* 어트리뷰트를 사용하여 출력 페이지를 생성합니다. 서브 클래스는 이것을 재정의하여, 더욱 동적인 에러 출력을 생성할 수 있습니다.

그러나, 보안 관점에서 오래된 사용자에게는 진단을 내보내지 않는 것이 좋습니다; 이상적으로, 진단 출력을 활성화하기 위해서는 특별한 것을 해야 합니다. 이것이 기본 구현이 아무것도 포함하지 않는 이유입니다.

error_status

에러 응답에 사용되는 HTTP 상태. **PEP 3333**에 정의된 상태 문자열이어야 합니다; 기본값은 500 코드와 메시지입니다.

error_headers

에러 응답에 사용되는 HTTP 헤더. 이것은 **PEP 3333**에서 설명하는 WSGI 응답 헤더 (`(name, value)` 튜플)의 리스트여야 합니다. 기본 리스트는 단지 콘텐츠 형식을 `text/plain`으로 설정합니다.

error_body

에러 응답 바디. 이것은 HTTP 응답 바디 바이트열이어야 합니다. 기본적으로 "A server error occurred. Please contact the administrator." 라는 단순 텍스트입니다.

PEP 3333의 "선택적 플랫폼 특정 파일 처리" 기능을 위한 메서드와 어트리뷰트:

wsgi_file_wrapper

wsgiref.types.FileWrapper 와 호환되는 `wsgi.file_wrapper` 팩토리나 `None`. 이 어트리뷰트의 기본값은 *wsgiref.util.FileWrapper* 클래스입니다.

sendfile()

플랫폼 특정 파일 전송을 구현하기 위해 재정의합니다. 이 메서드는 응용 프로그램의 반환 값이 *wsgi_file_wrapper* 어트리뷰트로 지정된 클래스의 인스턴스일 때만 호출됩니다. 파일을 성공적으로 전송할 수 있었으면 참값을 반환해야 합니다. 그러면 기본 전송 코드가 실행되지 않습니다. 이 메서드의 기본 구현은 단지 거짓 값을 반환합니다.

기타 메서드와 어트리뷰트:

origin_server

특별한 `Status:` 헤더를 통해 HTTP 상태를 원하는 CGI와 같은 게이트웨이 프로토콜을 통하는 것이 아니라, 처리기의 *_write()*와 *_flush()*가 클라이언트와 직접 통신하는 데 사용되는 경우 이 어트리뷰트를 참으로 설정해야 합니다.

*BaseHandler*에서는 이 어트리뷰트의 기본값이 참이지만, *BaseCGIHandler*와 *CGIHandler*에서는 거짓입니다.

http_version

*origin_server*가 참이면, 이 문자열 어트리뷰트를 사용하여 클라이언트로 보내는 응답 집합의 HTTP 버전을 설정합니다. 기본값은 `"1.0"`입니다.

wsgiref.handlers.**read_environ**()

CGI 변수를 `os.environ`에서 **PEP 3333** "유니코드에 들어있는 바이트열" 문자열로 변환하여, 새 딕셔너리를 반환합니다. 이 함수는 `os.environ`을 직접 사용하는 대신 *CGIHandler*와 *IISCGIHandler*에서 사용됩니다. `os.environ`은 파이썬 3을 사용하는 모든 플랫폼과 웹 서버에서 WSGI를 준수한다는 보장이 없습니다 – 구체적으로, OS의 실제 환경이 유니코드인 곳(가령 윈도우)이나 환경은 바이트열이지만 파이썬이 디코딩하기 위해 사용하는 시스템 인코딩이 ISO-8859-1 이외의 것인 곳(예를 들어 UTF-8을 사용하는 유닉스 시스템).

여러분 자신의 CGI 기반 처리기를 구현한다면, `os.environ`에서 직접 값을 복사하는 대신 이 루틴을 사용하는 것이 좋습니다.

Added in version 3.2.

22.2.6 wsgiref.types — 정적 형 검사기를 위한 WSGI 형

This module provides various types for static type checking as described in **PEP 3333**.

Added in version 3.11.

class wsgiref.types.**StartResponse**

A *typing.Protocol* describing **start_response()** callables (**PEP 3333**).

wsgiref.types.**WSGIEnvironment**

A type alias describing a WSGI environment dictionary.

wsgiref.types.**WSGIApplication**

WSGI 응용 프로그램 콜러블을 기술하는 형 에일리어스.

class wsgiref.types.**InputStream**

A *typing.Protocol* describing a **WSGI Input Stream**.

class wsgiref.types.**ErrorStream**

A *typing.Protocol* describing a **WSGI Error Stream**.

class wsgiref.types.**FileWrapper**

A *typing.Protocol* describing a **file wrapper**. See *wsgiref.util.FileWrapper* for a concrete implementation of this protocol.

22.2.7 예제

이것은 작동하는 "Hello World" WSGI 응용 프로그램으로, *start_response* 콜러블은 *StartResponse* 프로토콜을 따라야 합니다:

```
"""
모든 WSGI 응용 프로그램은 응용 프로그램 객체를 가져야
합니다 - 두 개의 인자를 받아들이는 콜러블. 이를 위해,
함수를 사용하려고 합니다 (함수로 제한되지는 않습니다,
예를 들어 클래스를 사용할 수 있습니다). 함수에 전달되는
첫 번째 인자는 CGI 스타일 환경 변수를 포함하는 딕셔너리이고,
두 번째 변수는 콜러블 객체입니다.
"""
from wsgiref.simple_server import make_server


def hello_world_app(environ, start_response):
    status = "200 OK"  # HTTP Status
    headers = [("Content-type", "text/plain; charset=utf-8")]  # HTTP 헤더
    start_response(status, headers)

    # 반환된 객체가 인쇄될 것입니다.
    return [b"Hello World"]

with make_server("", 8000, hello_world_app) as httpd:
    print("Serving on port 8000...")

    # 프로세스를 죽일 때까지 서빙합니다
    httpd.serve_forever()
```

현재 디렉터리를 제공하는 WSGI 응용 프로그램의 예, 명령 줄에서 선택적 디렉터리와 포트 번호(기본값: 8000)를 받아들입니다:

```
"""
작은 wsgiref 기반 웹 서버. 서비스할 경로와 선택적
포트 번호(기본값 8000)를 취한 다음, 파일을 서비스하려고
시도합니다. 파일 이름에서 MIME 유형을 추측하고, 파일을
찾을 수 없으면 404 에러를 발생시킵니다.
"""
import mimetypes
import os
import sys
from wsgiref import simple_server, util


def app(environ, respond):
    # 파일 이름과 MIME 유형을 가져옵니다
    fn = os.path.join(path, environ["PATH_INFO"][1:])
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    if "." not in fn.split(os.path.sep)[-1]:
        fn = os.path.join(fn, "index.html")
    mime_type = mimetypes.guess_file_type(fn)[0]

    # 파일이 존재하면 200 OK 를, 그렇지 않으면 404 Not Found 를 돌려줍니다
    if os.path.exists(fn):
        respond("200 OK", [("Content-Type", mime_type)])
        return util.FileWrapper(open(fn, "rb"))
    else:
        respond("404 Not Found", [("Content-Type", "text/plain")])
        return [b"not found"]


if __name__ == "__main__":
    # 명령줄 인자에서 경로와 포트를 가져옵니다
    path = sys.argv[1] if len(sys.argv) > 1 else os.getcwd()
    port = int(sys.argv[2]) if len(sys.argv) > 2 else 8000

    # control-c 를 누를 때까지 서버를 만들고 시작합니다
    httpd = simple_server.make_server("", port, app)
    print(f"Serving {path} on port {port}, control-C to stop")
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        print("Shutting down.")
        httpd.server_close()
```

22.3 urllib — URL 처리 모듈

소스 코드: Lib/urllib/

---

urllib는 URL 작업을 위한 여러 모듈을 모은 패키지입니다.:

- URL을 열고 읽기 위한 *urllib.request*
- *urllib.request*에 의해 발생하는 예외를 포함하는 *urllib.error*
- URL 구문 분석을 위한 *urllib.parse*
- robots.txt 파일을 구문 분석하기 위한 *urllib.robotparser*

22.4 urllib.request — URL을 열기 위한 확장 가능한 라이브러리

소스 코드: Lib/urllib/request.py

---

urllib.request 모듈은 복잡한 세계에서 URL(대부분 HTTP)을 여는 데 도움이 되는 함수와 클래스를 정의합니다 — 기본(basic)과 다이제스트 인증, 리디렉션, 쿠키 등.

> **더 보기**
>
> 더 고수준 HTTP 클라이언트 인터페이스로 Requests 패키지를 권장합니다.

> **경고**
>
> On macOS it is unsafe to use this module in programs using `os.fork()` because the `getproxies()` implementation for macOS uses a higher-level system API. Set the environment variable `no_proxy` to `*` to avoid this problem (e.g. `os.environ["no_proxy"] = "*"`).

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

`urllib.request` 모듈은 다음 함수를 정의합니다:

`urllib.request.`**`urlopen`**(*url*, *data=None*, [*timeout*,]*, *context=None*)

유효하고, 올바르게 인코딩된 URL을 포함하는 문자열이나 `Request` 객체일 수 있는, *url*을 엽니다.

*data*는 서버로 전송할 추가 데이터를 지정하는 객체이거나, 그러한 데이터가 필요하지 않으면 `None` 이어야 합니다. 자세한 내용은 `Request`를 참조하십시오.

urllib.request 모듈은 HTTP/1.1을 사용하고 HTTP 요청에 `Connection:close` 헤더를 포함합니다.

선택적 *timeout* 매개 변수는 연결 시도와 같은 연산을 블로킹하기 위한 시간제한을 초 단위로 지정합니다 (지정하지 않으면 전역 기본 시간제한 설정이 사용됩니다). 이것은 실제로는 HTTP, HTTPS 및 FTP 연결에서만 작동합니다.

*context*가 지정되면, 다양한 SSL 옵션을 기술하는 `ssl.SSLContext` 인스턴스이어야 합니다. 자세한 내용은 `HTTPSConnection`을 참조하십시오.

이 함수는 항상 컨텍스트 관리자로 작동할 수 있고 *url*, *headers* 및 *status* 프로퍼티를 가진 객체를 반환합니다. 이러한 프로퍼티에 대한 자세한 내용은 `urllib.response.addinfourl`을 참조하십시오.

HTTP 및 HTTPS URL의 경우, 이 함수는 약간 수정된 `http.client.HTTPResponse` 객체를 반환합니다. 위의 세 가지 새로운 메서드 외에도, msg 어트리뷰트에는 `HTTPResponse` 설명서에 지정된 대로 응답 헤더 대신 `reason` 어트리뷰트와 — 서버가 반환한 이유 문구 — 같은 정보가 포함됩니다.

For FTP, file, and data URLs, this function returns a `urllib.response.addinfourl` object.

프로토콜 에러 시 `URLError`를 발생시킵니다.

아무런 처리기도 요청을 처리하지 않으면 `None`이 반환될 수 있습니다 (기본 설치된 전역 `OpenerDirector`는 `UnknownHandler`를 사용하여 이러한 상황이 발생하지 않도록 합니다).

또한, 프락시 설정이 감지되면 (예를 들어, `http_proxy`와 같은 `*_proxy` 환경 변수가 설정될 때), `ProxyHandler`가 기본적으로 설치되어 프락시를 통해 요청이 처리되도록 합니다.

파이썬 2.6 및 이전 버전의 레거시 `urllib.urlopen` 함수는 중단되었습니다; `urllib.request.urlopen()`는 이전 `urllib2.urlopen`에 해당합니다. 딕셔너리 매개 변수를 `urllib.urlopen`에 전달하여 수행되었던 프락시 처리는 `ProxyHandler` 객체를 사용하여 얻을 수 있습니다.

기본 오프너는 요청 객체에서 취한 인자 `fullurl`, `data`, `headers`, `method`로 감사 이벤트 `urllib.Request`를 발생시킵니다.

버전 3.2에서 변경: *cafile*과 *capath*가 추가되었습니다.

가능하다면 (즉, `ssl.HAS_SNI`가 참이라면) HTTPS 가상 호스트가 지원됩니다.

*data*는 이터러블 객체일 수 있습니다.

버전 3.3에서 변경: *cadefault*가 추가되었습니다.

버전 3.4.3에서 변경: *context*가 추가되었습니다.

버전 3.10에서 변경: HTTPS connection now send an ALPN extension with protocol indicator `http/1.1` when no *context* is given. Custom *context* should set ALPN protocols with `set_alpn_protocols()`.

버전 3.13에서 변경: Remove *cafile*, *capath* and *cadefault* parameters: use the *context* parameter instead.

urllib.request.**install_opener**(*opener*)

OpenerDirector 인스턴스를 기본 전역 오프너로 설치합니다. 오프너 설치는 urlopen이 해당 오프너를 사용하도록 하려는 경우에만 필요합니다; 그렇지 않으면 단순히 urlopen() 대신 OpenerDirector.open()을 호출하십시오. 코드는 실제 OpenerDirector를 확인하지 않으며, 적절한 인터페이스를 가진 클래스면 모두 작동합니다.

urllib.request.**build_opener**([*handler*, ...])

주어진 순서대로 처리기를 연결하는 OpenerDirector 인스턴스를 반환합니다. *handler*는 BaseHandler의 인스턴스이거나 BaseHandler의 서브 클래스(이 경우 매개 변수 없이 생성자를 호출할 수 있어야 합니다)일 수 있습니다. 다음과 같은 클래스들의 인스턴스는 *handler*에 그들, 그들의 인스턴스 또는 그들의 서브 클래스가 포함되지 않는 한 *handler* 앞에 있습니다: ProxyHandler (프락시 설정이 감지되는 경우), UnknownHandler, HTTPHandler, HTTPDefaultErrorHandler, HTTPRedirectHandler, FTPHandler, FileHandler, HTTPErrorProcessor.

파이썬 설치에 SSL 지원이 있으면 (즉, ssl 모듈을 임포트 할 수 있으면) HTTPSHandler도 추가됩니다.

BaseHandler 서브 클래스는 또한 handler_order 어트리뷰트를 변경하여 처리기 리스트에서 자신의 위치를 수정할 수 있습니다.

urllib.request.**pathname2url**(*path*, *, *add_scheme=False*)

Convert the given local path to a file: URL. This function uses quote() function to encode the path.

If *add_scheme* is false (the default), the return value omits the file: scheme prefix. Set *add_scheme* to true to return a complete URL.

This example shows the function being used on Windows:

```
>>> from urllib.request import pathname2url
>>> path = 'C:\\Program Files'
>>> pathname2url(path, add_scheme=True)
'file:///C:/Program%20Files'
```

버전 3.14에서 변경: Windows drive letters are no longer converted to uppercase, and : characters not following a drive letter no longer cause an OSError exception to be raised on Windows.

버전 3.14에서 변경: Paths beginning with a slash are converted to URLs with authority sections. For example, the path /etc/hosts is converted to the URL ///etc/hosts.

버전 3.14에서 변경: The *add_scheme* parameter was added.

urllib.request.**url2pathname**(*url*, *, *require_scheme=False*, *resolve_host=False*)

Convert the given file: URL to a local path. This function uses unquote() to decode the URL.

If *require_scheme* is false (the default), the given value should omit a file: scheme prefix. If *require_scheme* is set to true, the given value should include the prefix; a URLError is raised if it doesn' t.

The URL authority is discarded if it is empty, localhost, or the local hostname. Otherwise, if *resolve_host* is set to true, the authority is resolved using socket.gethostbyname() and discarded if it matches a local IP address (as per **RFC 8089 §3**). If the authority is still unhandled, then on Windows a UNC path is returned, and on other platforms a URLError is raised.

This example shows the function being used on Windows:

```
>>> from urllib.request import url2pathname
>>> url = 'file:///C:/Program%20Files'
>>> url2pathname(url, require_scheme=True)
'C:\\Program Files'
```

버전 3.14에서 변경: Windows drive letters are no longer converted to uppercase, and : characters not following a drive letter no longer cause an OSError exception to be raised on Windows.

버전 3.14에서 변경: The URL authority is discarded if it matches the local hostname. Otherwise, if the authority isn' t empty or `localhost`, then on Windows a UNC path is returned (as before), and on other platforms a `URLError` is raised.

버전 3.14에서 변경: The URL query and fragment components are discarded if present.

버전 3.14에서 변경: The *require_scheme* and *resolve_host* parameters were added.

`urllib.request.`**`getproxies`**`()`

이 도우미 함수는 스킴에서 프락시 서버 URL로 매핑하는 딕셔너리를 반환합니다. 먼저 모든 운영 체제에서 대소 문자를 구분하지 않는 방식으로 `<scheme>_proxy`라는 변수를 환경에서 스캔하고, 찾을 수 없으면 macOS의 시스템 구성과 윈도우의 윈도우 시스템 레지스트리에서 프락시 정보를 찾습니다. 소문자와 대문자 환경 변수가 모두 존재하면 (그리고 다른 값을 가지면), 소문자가 선호됩니다.

> **참고**
>
> 일반적으로 스크립트가 CGI 환경에서 실행 중임을 나타내는 환경 변수 `REQUEST_METHOD`가 설정되면, 환경 변수 `HTTP_PROXY`(대문자 `_PROXY`)는 무시됩니다. 이 변수는 "Proxy:" HTTP 헤더를 사용하여 클라이언트가 주입할 수 있기 때문입니다. CGI 환경에서 HTTP 프락시를 사용해야 하면, `ProxyHandler`를 명시적으로 사용하거나 변수 이름이 소문자(또는 적어도 `_proxy` 접미사)가 되도록 하십시오.

다음과 같은 클래스가 제공됩니다:

`class` `urllib.request.`**`Request`**`(`*url, data=None, headers={}, origin_req_host=None, unverifiable=False, method=None*`)`

이 클래스는 URL 요청의 추상화입니다.

*url*은 유효하고 올바르게 인코딩된 URL을 포함하는 문자열이어야 합니다.

*data*는 서버로 전송할 추가 데이터를 지정하는 객체이거나, 그러한 데이터가 필요하지 않으면 `None`이어야 합니다. 현재 HTTP 요청은 *data*를 사용하는 유일한 요청입니다. 지원되는 객체 형에는 바이트열, 파일류 객체 및 바이트열류 객체의 이터러블이 포함됩니다. `Content-Length`와 `Transfer-Encoding` 헤더 필드가 모두 제공되지 않으면, `HTTPHandler`는 *data*의 형에 따라 이러한 헤더를 설정합니다. `Content-Length`는 바이트열 객체를 보내는 데 사용되는 반면, **RFC 7230**, 섹션 3.3.1에 지정된 `Transfer-Encoding: chunked`는 파일과 다른 이터러블을 보내는 데 사용됩니다.

HTTP POST 요청 메서드의 경우, *data*는 표준 *application/x-www-form-urlencoded* 형식의 버퍼여야 합니다. `urllib.parse.urlencode()` 함수는 매핑이나 2-튜플의 시퀀스를 취하고 이 형식의 ASCII 문자열을 반환합니다. *data* 매개 변수로 사용되기 전에 바이트열로 인코딩되어야 합니다.

*headers*는 딕셔너리이어야 하며, 각 키와 값을 인자로 사용하여 `add_header()`가 호출된 것처럼 처리됩니다. 이것은 종종 브라우저가 자신을 식별하는 데 사용하는 `User-Agent` 헤더 값을 "스푸핑"하는 데 사용됩니다 – 일부 HTTP 서버는 스크립트가 아닌 일반 브라우저에서 오는 요청만 허용합니다. 예를 들어, Mozilla Firefox는 자신을 `"Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"`로 식별하는 반면, `urllib`의 기본 사용자 에이전트 문자열은 `"Python-urllib/2.6"` (파이썬 2.6에서) 입니다. 모든 헤더 키는 카멜 표기법(camel case)으로 전송됩니다.

data 인자가 있으면 적절한 `Content-Type` 헤더가 포함되어야 합니다. 이 헤더가 제공되지 않고 *data*가 `None`이 아니면, `Content-Type: application/x-www-form-urlencoded`가 기본값으로 추가됩니다.

다음 두 인자는 제삼자 HTTP 쿠키를 올바르게 처리하는 데에만 관심이 있습니다:

*origin_req_host*는 **RFC 2965**에 의해 정의된 대로 오리진 트랜잭션의 요청 호스트여야 합니다. 기본값은 `http.cookiejar.request_host(self)` 입니다. 이것은 사용자가 시작한 원래 요청의 호스트 이름이나 IP 주소입니다. 예를 들어, HTML 문서의 이미지에 대한 요청이면, 이미지가 포함된 페이지에 대한 요청의 요청 호스트여야 합니다.

*unverifiable*은 **RFC 2965**에서 정의한 대로 요청을 확인할 수 없는지를 표시해야 합니다. 기본값은 `False`입니다. 확인할 수 없는 요청은 사용자에게 URL에 대한 승인 옵션이 없는 요청입니다. 예를 들어, HTML 문서의 이미지에 대한 요청이고, 사용자에게 이미지의 자동 가져오기를 승인할 수 있는 옵션이 없으면, 이것은 참이어야 합니다.

*method*는 사용될 HTTP 요청 메서드를 나타내는 문자열이어야 합니다 (예를 들어 `'HEAD'`). 제공되면, 해당 값은 `method` 어트리뷰트에 저장되고 `get_method()`에서 사용됩니다. 기본값은 *data*가 `None`이면 `'GET'`이고, 그렇지 않으면 `'POST'`입니다. 서브 클래스는 클래스 자체에서 `method` 어트리뷰트를 설정하여 다른 기본 메서드를 나타낼 수 있습니다.

> **참고**
>
> data 객체가 콘텐츠를 두 번 이상 (예를 들어 콘텐츠를 한 번만 생성 할 수 있는 파일이나 이터러블) 전달할 수 없고 요청이 HTTP 리디렉션이나 인증을 위해 재시도되는 경우, 요청이 예상대로 작동하지 않습니다. *data*는 헤더 바로 다음에 HTTP 서버로 전송됩니다. 라이브러리에서 100-continue 예상(expectation)을 지원하지 않습니다.

버전 3.3에서 변경: `Request.method` 인자가 Request 클래스에 추가됩니다.

버전 3.4에서 변경: 클래스 수준에서 기본 `Request.method`를 지정할 수 있습니다.

버전 3.6에서 변경: `Content-Length`가 제공되지 않고 *data*가 `None`이나 바이트열 객체가 아닐 때 에러를 발생시키지 않습니다. 대신 청크 전송 인코딩(chunked transfer encoding)으로 폴백 합니다.

class `urllib.request.`**`OpenerDirector`**

`OpenerDirector` 클래스는 서로 연결된 `BaseHandler`들을 통해 URL을 엽니다. 처리기 연결과 에러 복구를 관리합니다.

class `urllib.request.`**`BaseHandler`**

이것은 등록된 모든 처리기의 베이스 클래스이며 — 간단한 등록 메커니즘만 처리합니다.

class `urllib.request.`**`HTTPDefaultErrorHandler`**

HTTP 에러 응답에 대한 기본 처리기를 정의하는 클래스; 모든 응답은 `HTTPError` 예외로 바뀝니다.

class `urllib.request.`**`HTTPRedirectHandler`**

리디렉션을 처리하는 클래스.

class `urllib.request.`**`HTTPCookieProcessor`**(*cookiejar=None*)

HTTP 쿠키를 처리하는 클래스.

class `urllib.request.`**`ProxyHandler`**(*proxies=None*)

요청이 프락시를 거치게 합니다. *proxies*가 제공되면, 프로토콜 이름을 프락시 URL로 매핑하는 딕셔너리여야 합니다. 기본값은 환경 변수 `<protocol>_proxy`에서 프락시 목록을 읽는 것입니다. 프락시 환경 변수가 설정되어 있지 않으면, 윈도우 환경에서는 레지스트리의 인터넷 설정 섹션에서 프락시 설정을 가져오고, macOS 환경에서는 프락시 정보를 시스템 구성 프레임워크에서 가져옵니다.

자동 감지 프락시를 비활성화하려면 빈 딕셔너리를 전달하십시오.

`no_proxy` 환경 변수를 사용하여 프락시를 통해 도달해서는 안 되는 호스트를 지정할 수 있습니다; 설정되면, 쉼표로 구분된 호스트 이름 접미사의 목록이어야 하며, 선택적으로 `:port`가 추가됩니다, 예를 들어 `cern.ch,ncsa.uiuc.edu,some.host:8080`.

> **참고**
>
> 변수 `REQUEST_METHOD`가 설정되면 `HTTP_PROXY`는 무시됩니다; `getproxies()`의 설명서를 참조하십시오.

class `urllib.request.`**`HTTPPasswordMgr`**

`(realm, uri) -> (user, password)` 매핑 데이터베이스를 유지합니다.

class `urllib.request.`**`HTTPPasswordMgrWithDefaultRealm`**

`(realm, uri) -> (user, password)` 매핑 데이터베이스를 유지합니다. `None` realm은 포괄(catch-all) 영역으로 간주하며 다른 영역에 맞지 않으면 검색됩니다.

class urllib.request.**HTTPPasswordMgrWithPriorAuth**

uri -> is_authenticated 매핑 데이터베이스도 포함하는 *HTTPPasswordMgrWithDefaultRealm*의 변형. BasicAuth 처리기에서 401 응답을 먼저 기다리는 대신 인증 자격 증명을 언제 보낼 것인지 결정하는 데 사용할 수 있습니다.

Added in version 3.5.

class urllib.request.**AbstractBasicAuthHandler**(*password_mgr=None*)

원격 호스트와 프락시 모두에서 HTTP 인증을 돕는 믹스인 클래스입니다. *password_mgr*이 주어지면 *HTTPPasswordMgr* 과 호환되는 것이어야 합니다; 지원해야 하는 인터페이스에 대한 정보는 섹션 *HTTPPasswordMgr 객체*를 참조하십시오. *passwd_mgr*이 is_authenticated와 update_authenticated 메서드도 제공하면 (*HTTPPasswordMgrWithPriorAuth 객체*를 참조하십시오), 처리기는 지정된 URI에 대해 is_authenticated 결과를 사용하여 요청과 함께 인증 자격 증명을 보낼지를 판별합니다. is_authenticated가 URI에 대해 True를 반환하면, 자격 증명이 전송됩니다. is_authenticated가 False이면, 자격 증명이 전송되지 않으며, 그런 다음 401 응답이 수신되면 요청이 인증 자격 증명과 함께 다시 전송됩니다. 인증이 성공하면, 이 URI에 대해 is_authenticated를 True로 설정하기 위해 update_authenticated가 호출되어서, 이 URI나 모든 슈퍼 URI에 대한 후속 요청에 인증 자격 증명이 자동으로 포함됩니다.

Added in version 3.5: is_authenticated 지원이 추가되었습니다.

class urllib.request.**HTTPBasicAuthHandler**(*password_mgr=None*)

원격 호스트와의 인증을 처리합니다. *password_mgr*이 주어지면 *HTTPPasswordMgr* 과 호환되는 것이어야 합니다; 지원해야 하는 인터페이스에 대한 정보는 섹션 *HTTPPasswordMgr 객체*를 참조하십시오. 잘못된 인증 스킴(Authentication scheme)을 제시하면 HTTPBasicAuthHandler 가 *ValueError*를 발생시킵니다.

class urllib.request.**ProxyBasicAuthHandler**(*password_mgr=None*)

프락시와의 인증을 처리합니다. *password_mgr*이 주어지면 *HTTPPasswordMgr* 과 호환되는 것이어야 합니다; 지원해야 하는 인터페이스에 대한 정보는 섹션 *HTTPPasswordMgr 객체*를 참조하십시오.

class urllib.request.**AbstractDigestAuthHandler**(*password_mgr=None*)

원격 호스트와 프락시 모두에서 HTTP 인증을 돕는 믹스인 클래스입니다. *password_mgr*이 주어지면 *HTTPPasswordMgr* 과 호환되는 것이어야 합니다; 지원해야 하는 인터페이스에 대한 정보는 섹션 *HTTPPasswordMgr 객체*를 참조하십시오.

버전 3.14에서 변경: Added support for HTTP digest authentication algorithm SHA-256.

class urllib.request.**HTTPDigestAuthHandler**(*password_mgr=None*)

원격 호스트와의 인증을 처리합니다. *password_mgr*이 주어지면 *HTTPPasswordMgr* 과 호환되는 것이어야 합니다; 지원해야 하는 인터페이스에 대한 정보는 섹션 *HTTPPasswordMgr 객체*를 참조하십시오. 다이제스트 인증 처리기와 기본 인증 처리기가 모두 추가되면, 다이제스트 인증이 항상 먼저 시도됩니다. 다이제스트 인증이 다시 40x 응답을 반환하면, 기본 인증 처리기로 보내 처리됩니다. 이 처리기 메서드는 Digest나 Basic 이외의 인증 스킴(authentication scheme)이 제공될 때 *ValueError*를 발생시킵니다.

버전 3.3에서 변경: 지원되지 않는 인증 스킴에 대해 *ValueError*를 발생시킵니다.

class urllib.request.**ProxyDigestAuthHandler**(*password_mgr=None*)

프락시와의 인증을 처리합니다. *password_mgr*이 주어지면 *HTTPPasswordMgr* 과 호환되는 것이어야 합니다; 지원해야 하는 인터페이스에 대한 정보는 섹션 *HTTPPasswordMgr 객체*를 참조하십시오.

class urllib.request.**HTTPHandler**

HTTP URL 열기를 처리하는 클래스.

class urllib.request.**HTTPSHandler**(*debuglevel=0, context=None, check_hostname=None*)

HTTPS URL 열기를 처리하는 클래스. *context*와 *check_hostname*은 *http.client.HTTPSConnection*과 같은 의미입니다.

버전 3.2에서 변경: *context*와 *check_hostname*이 추가되었습니다.

class urllib.request.**FileHandler**

로컬 파일을 엽니다.

class urllib.request.**DataHandler**

데이터 URL을 엽니다.

Added in version 3.4.

class urllib.request.**FTPHandler**

FTP URL을 엽니다.

class urllib.request.**CacheFTPHandler**

지연 시간을 최소화하기 위해 열린 FTP 연결의 캐시를 유지하면서, FTP URL을 엽니다.

class urllib.request.**UnknownHandler**

알 수 없는 URL을 처리하기 위한 포괄적인(catch-all) 클래스.

class urllib.request.**HTTPErrorProcessor**

HTTP 에러 응답을 처리합니다.

22.4.1 Request 객체

다음 메서드는 *Request*의 공용 인터페이스를 설명하므로, 서브 클래스에서 모두 재정의될 수 있습니다. 또한 클라이언트가 구문 분석된 요청을 검사하는 데 사용할 수 있는 몇 가지 공용 어트리뷰트를 정의합니다.

Request.**full_url**

생성자에 전달된 원래 URL.

버전 3.4에서 변경.

Request.full_url은 setter, getter 및 deleter가 있는 프로퍼티입니다. *full_url*을 읽으면 프래그먼트가 있는 원래 요청 URL을 반환합니다 (있다면).

Request.**type**

URI 스킴.

Request.**host**

URI 주체(authority), 일반적으로 호스트이지만 콜론으로 구분된 포트를 포함할 수도 있습니다.

Request.**origin_req_host**

포트가 없는, 요청의 원래 호스트.

Request.**selector**

URI 경로. *Request*가 프락시를 사용하면, selector는 프락시로 전달되는 전체 URL이 됩니다.

Request.**data**

요청의 엔티티 바디, 또는 지정되지 않으면 None.

버전 3.4에서 변경: *Request.data*의 값을 변경하면 이제 "Content-Length" 헤더가 이전에 설정되거나 계산되었다면 삭제됩니다.

Request.**unverifiable**

불리언, **RFC 2965**에서 정의한 대로 요청을 확인할 수 없는지를 나타냅니다.

Request.**method**

사용할 HTTP 요청 메서드. 기본적으로 값은 *None*입니다. 이는 *get_method()*가 사용될 메서드의 일반적인 계산을 수행함을 뜻합니다. *Request* 서브 클래스의 클래스 수준에서 값을 설정해서 기본값을 제공하거나, *method* 인자를 통해 *Request* 생성자에 값을 전달하여 값을 설정할 수 있습니다 (그래서 *get_method()*의 기본 계산을 무시합니다).

Added in version 3.3.

버전 3.4에서 변경: 서브 클래스에서 이제 기본값을 설정할 수 있습니다; 이전에는 생성자 인자를 통해서만 설정할 수 있었습니다.

Request.**get_method**()

HTTP 요청 메서드를 나타내는 문자열을 반환합니다. *Request.method*가 None이 아니면, 그 값을 반환하고, 그렇지 않으면 *Request.data*가 None이면 'GET'을 반환하거나 그렇지 않으면 'POST'를 반환합니다. 이것은 HTTP 요청에만 의미가 있습니다.

버전 3.3에서 변경: get_method는 이제 *Request.method*의 값을 조사합니다.

Request.**add_header**(*key*, *val*)

요청에 다른 헤더를 추가합니다. 헤더는 현재 HTTP 처리기를 제외한 모든 처리기에서 무시되며, HTTP 처리기에서는 서버로 전송되는 헤더 리스트에 추가됩니다. 같은 이름을 가진 헤더를 두 개 이상 가질 수 없으며, *key*가 충돌하는 경우 후속 호출은 이전 호출을 덮어씁니다. 현재, 두 번 이상 사용될 때 의미가 있는 모든 헤더는 하나의 헤더만 사용하여 같은 기능을 얻는 (헤더 별) 방식을 가지므로 HTTP 기능의 손실은 없습니다. 이 메서드로 추가된 헤더는 리디렉션 된 요청에도 추가됨에 유의하십시오.

Request.**add_unredirected_header**(*key*, *header*)

리디렉션 된 요청에 추가되지 않을 헤더를 추가합니다.

Request.**has_header**(*header*)

인스턴스에 명명된 헤더가 있는지를 반환합니다 (일반과 리디렉션되지 않는 것을 모두 확인합니다).

Request.**remove_header**(*header*)

요청 인스턴스에서 명명된 헤더를 제거합니다 (일반과 리디렉션되지 않은 헤더 모두).

Added in version 3.4.

Request.**get_full_url**()

생성자에 제공된 URL을 반환합니다.

버전 3.4에서 변경.

*Request.full_url*을 반환합니다

Request.**set_proxy**(*host*, *type*)

프락시 서버에 연결하여 요청을 준비합니다. *host*와 *type*은 인스턴스의 것을 대체하고, 인스턴스의 selector는 생성자에 제공된 원래 URL이 됩니다.

Request.**get_header**(*header_name*, *default=None*)

지정된 헤더의 값을 반환합니다. 헤더가 없으면, default 값을 반환합니다.

Request.**header_items**()

요청 헤더의 튜플 (header_name, header_value) 리스트를 반환합니다.

버전 3.4에서 변경: 3.3부터 폐지된 add_data, has_data, get_data, get_type, get_host, get_selector, get_origin_req_host 및 is_unverifiable 요청 메서드가 제거되었습니다.

22.4.2 OpenerDirector 객체

OpenerDirector 인스턴스에는 다음과 같은 메서드가 있습니다:

OpenerDirector.**add_handler**(*handler*)

*handler*는 *BaseHandler*의 인스턴스여야 합니다. 다음 메서드가 검색되어, 가능한 체인에 추가됩니다 (HTTP 에러는 특별한 경우임에 유의하십시오). 다음에서 *protocol*은 처리할 실제 프로토콜로 바꿔야 합니다, 예를 들어 http_response()는 HTTP 프로토콜 응답 처리기입니다. 또한 *type*은 실제 HTTP 코드로 대체해야 합니다, 예를 들어 http_error_404()는 HTTP 404 에러를 처리합니다.

- <protocol>_open() — 처리기가 *protocol* URL을 여는 방법을 알고 있음을 알립니다.

  자세한 정보는 *BaseHandler.<protocol>_open()*을 참조하십시오.

- http_error_<type>() — 처리기가 HTTP 에러 코드 *type*을 갖는 HTTP 에러를 처리하는 방법을 알고 있음을 알립니다.

  자세한 정보는 *BaseHandler.http_error_<nnn>()*을 참조하십시오.

- <protocol>_error() — 처리기가 (http가 아닌) *protocol*의 에러를 처리하는 방법을 알고 있음을 알립니다.

- `<protocol>_request()` — 처리기가 *protocol* 요청을 전처리(pre-process)하는 방법을 알고 있음을 알립니다.

  자세한 정보는 `BaseHandler.<protocol>_request()`를 참조하십시오.

- `<protocol>_response()` — 처리기가 *protocol* 응답을 후처리(post-process)하는 방법을 알고 있음을 알립니다.

  자세한 정보는 `BaseHandler.<protocol>_response()`를 참조하십시오.

`OpenerDirector.`**`open`**(*url*, *data=None*[, *timeout*])

주어진 *url*(요청 객체나 문자열일 수 있습니다)을 열고, 선택적으로 주어진 *data*를 전달합니다. 인자, 반환 값 및 발생하는 예외는 `urlopen()`과 같습니다 (단순히 현재 설치된 전역 `OpenerDirector`의 `open()` 메서드를 호출합니다). 선택적 *timeout* 매개 변수는 연결 시도와 같은 연산을 블로킹하기 위한 시간제한을 초 단위로 지정합니다 (지정하지 않으면 전역 기본 시간제한 설정이 사용됩니다). 시간제한 기능은 실제로는 HTTP, HTTPS 및 FTP 연결에서만 작동합니다.

`OpenerDirector.`**`error`**(*proto*, **args*)

주어진 프로토콜의 에러를 처리합니다. 이것은 주어진 프로토콜에 대해 등록된 에러 처리기를 주어진 인자(프로토콜 특정입니다)로 호출합니다. HTTP 프로토콜은 HTTP 응답 코드를 사용하여 특정 에러 처리기를 결정하는 특수한 경우입니다; 처리기 클래스의 `http_error_<type>()` 메서드를 참조하십시오.

반환 값과 발생하는 예외는 `urlopen()`과 같습니다.

OpenerDirector 객체는 다음 3단계로 URL을 엽니다:

각 단계에서 이러한 메서드가 호출되는 순서는 처리기 인스턴스를 정렬하여 결정됩니다.

1. `<protocol>_request()`와 같은 이름의 메서드를 가진 모든 처리기가 요청을 전처리하기 위해 해당 메서드가 호출됩니다.
2. `<protocol>_open()`과 같은 이름의 메서드를 가진 처리기가 요청을 처리하기 위해 호출됩니다. 이 단계는 처리기가 `None`이 아닌 값(즉, 응답)을 반환하거나, 예외(보통 `URLError`)를 발생시킬 때 종료됩니다. 예외 전파가 허용됩니다.

   사실, 위의 알고리즘은 `default_open()`이라는 메서드를 먼저 시도됩니다. 이러한 모든 메서드가 `None`을 반환하면, 알고리즘은 `<protocol>_open()`과 같은 이름의 메서드에 대해 반복합니다. 이러한 모든 메서드가 `None`을 반환하면, 알고리즘은 `unknown_open()`이라는 메서드에 대해 반복합니다.

   이러한 메서드의 구현은 부모 `OpenerDirector` 인스턴스의 `open()`과 `error()` 메서드의 호출을 수반할 수 있음에 유의하십시오.
3. `<protocol>_response()`와 같은 이름의 메서드가 있는 모든 처리기는 응답을 후처리하기 위해 해당 메서드가 호출됩니다.

22.4.3 BaseHandler 객체

`BaseHandler` 객체는 직접적으로 유용한 몇 가지 메서드와 파생 클래스에서 사용하기 위한 다른 메서드를 제공합니다. 다음은 직접 사용하기 위한 것입니다:

`BaseHandler.`**`add_parent`**(*director*)

director를 부모로 추가합니다.

`BaseHandler.`**`close`**()

모든 부모를 제거합니다.

다음 어트리뷰트와 메서드는 `BaseHandler`에서 파생된 클래스에서만 사용해야 합니다.

> **참고**
>
> `<protocol>_request()`나 `<protocol>_response()` 메서드를 정의하는 서브 클래스는 `*Processor`라고 이름 붙이는 규칙이 채택되었습니다; 다른 모든 것들의 이름은 `*Handler`입니다.

BaseHandler.**parent**

다른 프로토콜을 사용하여 열거나 에러를 처리하는 데 사용할 수 있는 유효한 *OpenerDirector*.

BaseHandler.**default_open**(*req*)

이 메서드는 *BaseHandler*에 정의되지 않았지만, 서브 클래스가 모든 URL을 포착하려면 이를 정의해야 합니다.

구현되면 이 메서드는 부모 *OpenerDirector*에 의해 호출됩니다. *OpenerDirector*의 *open()* 메서드의 반환 값에 설명된 대로 파일류 객체를 반환하거나 None을 반환해야 합니다. 진짜 예외적인 상황이 발생하지 않는 한 (예를 들어, *MemoryError*는 *URLError*로 매핑하지 않아야 합니다), *URLError*를 발생해야 합니다.

이 메서드는 프로토콜별 open 메서드보다 먼저 호출됩니다.

BaseHandler.<protocol>_open(req)

이 메서드는 *BaseHandler* 에 정의되지 않았지만, 서브 클래스가 주어진 프로토콜로 URL을 처리하려면 이를 정의해야 합니다.

정의되면, 이 메서드는 부모 *OpenerDirector*에 의해 호출됩니다. 반환 값은 *default_open()*과 같아야 합니다.

BaseHandler.**unknown_open**(*req*)

이 메서드는 *BaseHandler* 에 정의되지 않았지만, 서브 클래스는 등록된 특정 처리기가 없는 모든 URL을 잡아서 열려면 이를 정의해야 합니다.

구현되면, 이 메서드는 *parent OpenerDirector*에 의해 호출됩니다. 반환 값은 *default_open()*과 같아야 합니다.

BaseHandler.**http_error_default**(*req*, *fp*, *code*, *msg*, *hdrs*)

이 메서드는 *BaseHandler* 에 정의되지 않았지만, 서브 클래스는 달리 처리되지 않은 HTTP 에러에 대해 포괄적인 처리를 제공하려면 이를 재정의해야 합니다. 에러가 발생하는 *OpenerDirector*에 의해 자동으로 호출되며, 다른 상황에서는 일반적으로 호출되지 않아야 합니다.

OpenerDirector will call this method with five positional arguments:

1. a *Request* object,
2. a file-like object with the HTTP error body,
3. the three-digit code of the error, as a string,
4. the user-visible explanation of the code, as a string, and
5. the headers of the error, as a mapping object.

반환 값과 발생하는 예외는 *urlopen()*의 것과 같아야 합니다.

BaseHandler.http_error_<nnn>(req, fp, code, msg, hdrs)

*nnn*은 3자리 HTTP 에러 코드여야 합니다. 이 메서드도 *BaseHandler* 에 정의되어 있지 않지만, 존재한다면 코드가 *nnn* 인 HTTP 에러가 발생할 때 서브 클래스의 인스턴스에 대해 호출됩니다.

특정 HTTP 에러를 처리하려면 서브 클래스가 이 메서드를 재정의해야 합니다.

인자, 반환 값 및 발생하는 예외는 *http_error_default()*와 같아야 합니다.

BaseHandler.<protocol>_request(req)

이 메서드는 *BaseHandler* 에 정의되지 않았지만, 서브 클래스는 주어진 프로토콜의 요청을 전처리하려면 이를 정의해야 합니다.

정의되면, 이 메서드는 부모 상위 *OpenerDirector*에 의해 호출됩니다. *req*는 *Request* 객체가 됩니다. 반환 값은 *Request* 객체여야 합니다.

BaseHandler.<protocol>_response(req, response)

이 메서드는 *BaseHandler* 에 정의되지 않았지만, 서브 클래스는 주어진 프로토콜의 응답을 후처리하려면 이를 정의해야 합니다.

정의되면, 이 메서드는 부모 *OpenerDirector*에 의해 호출됩니다. *req*는 *Request* 객체가 됩니다. *response*는 *urlopen()*의 반환 값과 같은 인터페이스를 구현하는 객체가 됩니다. 반환 값은 *urlopen()*의 반환 값과 같은 인터페이스를 구현해야 합니다.

22.4.4 HTTPRedirectHandler 객체

> **참고**
>
> 일부 HTTP 리디렉션은 이 모듈의 클라이언트 코드로부터의 액션을 요구합니다. 이 경우, *HTTPError*가 발생합니다. 다양한 리디렉션 코드의 정확한 의미에 대한 자세한 내용은 **RFC 2616**을 참조하십시오.
>
> HTTPRedirectHandler 에 HTTP, HTTPS 또는 FTP URL이 아닌 리디렉션 된 URL이 제공되면 보안을 고려하여 *HTTPError* 예외가 발생했습니다.

HTTPRedirectHandler.**redirect_request**(*req*, *fp*, *code*, *msg*, *hdrs*, *newurl*)

리디렉션에 대한 응답으로 *Request*나 None을 반환합니다. 이것은 서버로부터 리디렉션이 수신될 때 http_error_30*() 메서드의 기본 구현에 의해 호출됩니다. 리디렉션이 일어나야 하면, http_error_30*() 이 *newurl*로 리디렉션을 수행할 수 있도록 새 *Request*를 반환합니다. 그렇지 않으면 다른 처리기가 이 URL을 처리하려고 시도하지 않아야 한다면 *HTTPError*를 발생시키고, 자신은 할 수 없지만 다른 처리기가 처리할 수 있다면 None을 반환합니다.

> **참고**
>
> 이 메서드의 기본 구현은 **RFC 2616**을 엄격하게 따르지 않습니다. 즉, POST 요청에 대한 301과 302 응답은 사용자의 확인 없이 자동으로 리디렉션되지 않아야 합니다. 실제로는, 브라우저들이 POST를 GET으로 변경하여 이러한 응답의 자동 리디렉션을 허용하며, 기본 구현은 이 동작을 재현합니다.

HTTPRedirectHandler.**http_error_301**(*req*, *fp*, *code*, *msg*, *hdrs*)

Location: 이나 URI: URL로 리디렉션 합니다. 이 메서드는 HTTP 'moved permanently(영구적으로 이전했음)' 응답을 받을 때 부모 *OpenerDirector*에 의해 호출됩니다.

HTTPRedirectHandler.**http_error_302**(*req*, *fp*, *code*, *msg*, *hdrs*)

*http_error_301()*과 같지만, 'found(발견됨)' 응답에 대해 호출됩니다.

HTTPRedirectHandler.**http_error_303**(*req*, *fp*, *code*, *msg*, *hdrs*)

*http_error_301()*과 같지만, 'see other(다른 곳을 보세요)' 응답에 대해 호출됩니다.

HTTPRedirectHandler.**http_error_307**(*req*, *fp*, *code*, *msg*, *hdrs*)

*http_error_301()*과 같지만, 'temporary redirect(임시 리디렉션)' 응답에 대해 호출됩니다. 요청 메서드를 POST에서 GET으로 바꾸는 것을 허락하지 않습니다.

HTTPRedirectHandler.**http_error_308**(*req*, *fp*, *code*, *msg*, *hdrs*)

*http_error_301()*과 같지만, 'permanent redirect(영구 리디렉션)' 응답에 대해 호출됩니다. 요청 메서드를 POST에서 GET으로 바꾸는 것을 허락하지 않습니다.

Added in version 3.11.

22.4.5 HTTPCookieProcessor 객체

HTTPCookieProcessor 인스턴스에는 하나의 어트리뷰트가 있습니다:

HTTPCookieProcessor.**cookiejar**

쿠키가 저장되는 *http.cookiejar.CookieJar*.

22.4.6 ProxyHandler 객체

ProxyHandler.<protocol>_open(request)

*ProxyHandler*에는 생성자에 지정된 *proxies* 딕셔너리에 프락시가 있는 모든 *protocol*에 대해 `<protocol>_open()` 메서드가 있습니다. 이 메서드는 `request.set_proxy()`를 호출하여 요청이 프락시를 통과하도록 수정하고, 체인에 있는 다음 처리기를 호출하여 실제로 프로토콜을 실행합니다.

22.4.7 HTTPPasswordMgr 객체

이 메서드는 *HTTPPasswordMgr*과 *HTTPPasswordMgrWithDefaultRealm* 객체에서 사용 가능합니다.

HTTPPasswordMgr.**add_password**(*realm*, *uri*, *user*, *passwd*)

*uri*는 단일 URI이거나 URI의 시퀀스일 수 있습니다. *realm*, *user* 및 *passwd*는 문자열이어야 합니다. 이는 *realm*과 지정된 URI 중 어느 하나의 슈퍼 URI에 대한 인증이 주어질 때 `(user, passwd)`가 인증 토큰으로 사용되도록 합니다.

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)

주어진 realm과 URI에 대한 사용자/암호를 (있다면) 가져옵니다. 일치하는 사용자/암호가 없으면 이 메서드는 `(None, None)`을 반환합니다.

HTTPPasswordMgrWithDefaultRealm 객체의 경우, 주어진 *realm*에 일치하는 사용자/암호가 없으면 영역 `None`이 검색됩니다.

22.4.8 HTTPPasswordMgrWithPriorAuth 객체

이 암호 관리자는 *HTTPPasswordMgrWithDefaultRealm*를 확장하여 인증 자격 증명을 항상 보내야 하는 URI를 추적하는 것을 지원합니다.

HTTPPasswordMgrWithPriorAuth.**add_password**(*realm*, *uri*, *user*, *passwd*, *is_authenticated=False*)

realm, *uri*, *user*, *passwd*는 *HTTPPasswordMgr.add_password()*와 같습니다. *is_authenticated*는 주어진 URI나 URI 리스트에 대한 `is_authenticated` 플래그의 초깃값을 설정합니다. *is_authenticated*가 `True`로 지정되면, *realm*는 무시됩니다.

HTTPPasswordMgrWithPriorAuth.**find_user_password**(*realm*, *authuri*)

HTTPPasswordMgrWithDefaultRealm 객체와 같습니다

HTTPPasswordMgrWithPriorAuth.**update_authenticated**(*self*, *uri*, *is_authenticated=False*)

주어진 *uri*나 URI 리스트에 대해 `is_authenticated` 플래그를 갱신합니다.

HTTPPasswordMgrWithPriorAuth.**is_authenticated**(*self*, *authuri*)

주어진 URI에 대한 `is_authenticated` 플래그의 현재 상태를 반환합니다.

22.4.9 AbstractBasicAuthHandler 객체

AbstractBasicAuthHandler.**http_error_auth_reqed**(*authreq*, *host*, *req*, *headers*)

사용자/암호 쌍을 가져오고 요청을 다시 시도하여 인증 요청을 처리합니다. *authreq*는 영역(realm)에 대한 정보가 요청에 포함된 헤더의 이름이어야 하고, *host*는 인증할 URL과 경로를 지정하고, *req*는 (실패한) *Request* 객체여야 하며, *headers*는 에러 헤더여야 합니다.

*host*는 주체(예를 들어 `"python.org"`)거나 주체 구성 요소를 포함하는 URL(예를 들어 `"http://python.org/"`)입니다. 어느 경우이든, 주체는 userinfo 구성 요소를 포함하지 않아야 합니다 (따라서, `"python.org"`와 `"python.org:80"`은 좋지만, `"joe:password@python.org"`는 유효하지 않습니다).

22.4.10 HTTPBasicAuthHandler 객체

HTTPBasicAuthHandler.**http_error_401**(*req*, *fp*, *code*, *msg*, *hdrs*)

가능하다면 인증 정보로 요청을 재시도합니다.

22.4.11 ProxyBasicAuthHandler 객체

`ProxyBasicAuthHandler.`**`http_error_407`**(*req, fp, code, msg, hdrs*)

가능하다면 인증 정보로 요청을 재시도합니다.

22.4.12 AbstractDigestAuthHandler 객체

`AbstractDigestAuthHandler.`**`http_error_auth_reqed`**(*authreq, host, req, headers*)

*authreq*는 영역(realm)에 대한 정보가 요청에 포함된 헤더의 이름이어야 하고, *host*는 인증할 호스트여야 하고, *req*는 (실패한) `Request` 객체여야 하며, *headers*는 에러 헤더여야 합니다.

22.4.13 HTTPDigestAuthHandler 객체

`HTTPDigestAuthHandler.`**`http_error_401`**(*req, fp, code, msg, hdrs*)

가능하다면 인증 정보로 요청을 재시도합니다.

22.4.14 ProxyDigestAuthHandler 객체

`ProxyDigestAuthHandler.`**`http_error_407`**(*req, fp, code, msg, hdrs*)

가능하다면 인증 정보로 요청을 재시도합니다.

22.4.15 HTTPHandler 객체

`HTTPHandler.`**`http_open`**(*req*)

Send an HTTP request, which can be either GET or POST, depending on `req.data`.

22.4.16 HTTPSHandler 객체

`HTTPSHandler.`**`https_open`**(*req*)

Send an HTTPS request, which can be either GET or POST, depending on `req.data`.

22.4.17 FileHandler 객체

`FileHandler.`**`file_open`**(*req*)

호스트 이름이 없거나, 호스트 이름이 `'localhost'`인 경우 파일을 로컬에서 엽니다.

버전 3.2에서 변경: 이 메서드는 로컬 호스트 명에만 적용할 수 있습니다. 원격 호스트 이름이 제공되면, `URLError`가 발생합니다.

22.4.18 DataHandler 객체

`DataHandler.`**`data_open`**(*req*)

데이터 URL을 읽습니다. 이러한 종류의 URL에는 URL 자체에 인코딩된 콘텐츠가 포함됩니다. 데이터 URL 문법은 **RFC 2397**에 지정되어 있습니다. 이 구현은 base64로 인코딩된 데이터 URL의 공백을 무시하기 때문에 URL은 소스 파일과 관계없이 줄 넘김 될 수 있습니다. 그러나 일부 브라우저가 base64로 인코딩된 데이터 URL 끝에 패딩이 누락된 것에 대해 신경 쓰지 않지만, 이 구현은 이 경우 `ValueError`를 발생시킵니다.

22.4.19 FTPHandler 객체

`FTPHandler.`**`ftp_open`**(*req*)

*req*로 표시된 FTP 파일을 엽니다. 로그인은 항상 빈 사용자 이름과 암호로 수행됩니다.

22.4.20 CacheFTPHandler 객체

CacheFTPHandler 객체는 다음과 같은 추가 메서드가 있는 *FTPHandler* 객체입니다:

CacheFTPHandler.**setTimeout**(*t*)

연결 시간제한을 *t* 초로 설정합니다.

CacheFTPHandler.**setMaxConns**(*m*)

캐시 된 최대 연결 수를 *m*으로 설정합니다.

22.4.21 UnknownHandler 객체

UnknownHandler.**unknown_open**()

URLError 예외를 발생시킵니다.

22.4.22 HTTPErrorProcessor 객체

HTTPErrorProcessor.**http_response**(*request*, *response*)

HTTP 에러 응답을 처리합니다.

200 에러 코드의 경우, 응답 객체가 즉시 반환됩니다.

200 이 아닌 에러 코드의 경우, *OpenerDirector.error()*를 통해 단순히 작업을 http_error_<type>() 처리기 메서드로 전달합니다. 결국, 다른 처리기가 에러를 처리하지 않으면 *HTTPDefaultErrorHandler*는 *HTTPError*를 발생시킵니다.

HTTPErrorProcessor.**https_response**(*request*, *response*)

HTTPS 에러 응답을 처리합니다.

동작은 *http_response()*와 같습니다.

22.4.23 예

아래 예 외에도 urllib-howto에는 더 많은 예가 나와 있습니다.

이 예제는 python.org 메인 페이지를 가져와서 첫 300바이트를 표시합니다:

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!doctype html>\n<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9
↪">   <![endif]-->\n<!--[if IE 7]>      <html class="no-js ie7 lt-ie8 lt-ie9">   ␣
↪      <![endif]-->\n<!--[if IE 8]>      <html class="no-js ie8 lt-ie9">
```

urlopen은 바이트열 객체를 반환함에 유의하십시오. 이는 urlopen이 HTTP 서버로부터 수신한 바이트 스트림의 인코딩을 자동으로 결정할 방법이 없기 때문입니다. 일반적으로, 프로그램은 일단 적절한 인코딩을 결정하거나 추측하면 반환된 바이트열 객체를 문자열로 디코딩합니다.

다음 HTML 명세 문서 https://html.spec.whatwg.org/#charset 은 HTML이나 XML 문서가 인코딩 정보를 지정할 수 있는 다양한 방법을 나열합니다.

For additional information, see the W3C document: https://www.w3.org/International/questions/qa-html-encoding-declarations.

python.org 웹 사이트는 메타 태그에 지정된 대로 *utf-8* 인코딩을 사용하므로, 바이트열 객체를 디코딩할 때도 이를 사용합니다:

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
...
<!doctype html>
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">   <![endif]-->
<!-
```

컨텍스트 관리자 방식을 사용하지 않고도 같은 결과를 얻을 수 있습니다:

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> try:
...     print(f.read(100).decode('utf-8'))
... finally:
...     f.close()
...
<!doctype html>
<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">   <![endif]-->
<!--
```

다음 예에서는, CGI의 표준 입력으로 데이터 스트림을 전송하고 반환되는 데이터를 읽습니다. 이 예제는 파이썬 설치가 SSL을 지원할 때만 작동함에 유의하십시오.

```
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-bin/test.cgi',
...                              data=b'This data is passed to stdin of the CGI')
>>> with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
...
Got Data: "This data is passed to stdin of the CGI"
```

위 예제에서 사용한 샘플 CGI의 코드는 다음과 같습니다:

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

다음은 *Request*를 사용하여 PUT 요청을 수행하는 예입니다:

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA, method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

기본 HTTP 인증 사용:

```
import urllib.request
# 기본 HTTP 인증을 지원하는 OpenerDirector를 만듭니다...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
# ...그리고 urlopen과 함께 사용할 수 있도록 전역으로 설치합니다.
urllib.request.install_opener(opener)
with urllib.request.urlopen('http://www.example.com/login.html') as f:
    print(f.read().decode('utf-8'))
```

*build_opener()*는 기본적으로 *ProxyHandler*를 포함하여 많은 처리기를 제공합니다. 기본적으로, *ProxyHandler*는 <scheme>_proxy라는 이름의 환경 변수를 사용합니다, 여기서 <scheme>은 관련된 URL 스킴입니다. 예를 들어, HTTP 프락시의 URL을 얻기 위해 `http_proxy` 환경 변수를 읽습니다.

이 예는 기본 *ProxyHandler*를 프로그래밍 방식으로 제공되는 프락시 URL을 사용하는 것으로 대체하고, *ProxyBasicAuthHandler*로 프락시 인증 지원을 추가합니다.

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/
↪'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
# 이번에는, OpenerDirector를 설치하지 않고 직접 사용합니다:
with opener.open('http://www.example.com/login.html') as f:
   print(f.read().decode('utf-8'))
```

HTTP 헤더 추가하기:

Request 생성자에 *headers* 인자를 사용하십시오, 또는:

```
import urllib.request
req = urllib.request.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
with urllib.request.urlopen(req) as f:
    print(f.read().decode('utf-8'))
```

*OpenerDirector*는 모든 *Request*에 *User-Agent* 헤더를 자동으로 추가합니다. 이것을 바꾸려면:

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
with opener.open('http://www.example.com/') as f:
   print(f.read().decode('utf-8'))
```

또한, *Request*가 *urlopen()*(또는 *OpenerDirector.open()*)으로 전달될 때 몇 가지 표준 헤더(*Content-Length*, *Content-Type* 및 *Host*)가 추가됨을 기억하십시오.

다음은 `GET` 메서드를 사용하여 파라미터가 포함된 URL을 가져오는 예제 세션입니다:

```
>>> import urllib.request
>>> import urllib.parse
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> url = "http://www.musi-cal.com/cgi-bin/query?%s" % params
>>> with urllib.request.urlopen(url) as f:
...     print(f.read().decode('utf-8'))
...
```

다음 예제는 대신 `POST` 메서드를 사용합니다. urlencode의 파라미터 출력이 데이터로 urlopen에 보내기 전에 바이트열로 인코딩됨에 유의하십시오:

```
>>> import urllib.request
>>> import urllib.parse
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr", data) as f:
...     print(f.read().decode('utf-8'))
...
```

다음 예제는 명시적으로 지정된 HTTP 프락시를 사용하여 환경 설정을 대체합니다:

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.request.build_opener(urllib.request.ProxyHandler(proxies))
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
...
```

다음 예제는 프락시를 전혀 사용하지 않도록 환경 설정을 대체합니다:

```
>>> import urllib.request
>>> opener = urllib.request.build_opener(urllib.request.ProxyHandler({}))
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
...
```

22.4.24 레거시 인터페이스

다음 함수와 클래스는 파이썬 2 모듈 `urllib`(`urllib2`가 아니라)에서 이식됩니다. 나중에 언젠가 폐지될 수 있습니다.

`urllib.request.`**`urlretrieve`**(*url*, *filename=None*, *reporthook=None*, *data=None*)

URL로 표시된 네트워크 객체를 로컬 파일로 복사합니다. URL이 로컬 파일을 가리키면, 파일 이름을 제공하지 않는 한 객체가 복사되지 않습니다. 튜플 `(filename, headers)`를 반환합니다. 여기서 *filename*은 객체를 찾을 수 있는 로컬 파일 이름이며, *headers*는 (원격 객체에 대해) *`urlopen()`*이 반환한 객체의 `info()` 메서드가 반환한 것입니다. 예외는 *`urlopen()`*과 같습니다.

있다면, 두 번째 인자는 복사할 파일 위치를 지정합니다 (없으면, 위치는 생성된 이름을 가진 임시 파일이 됩니다). 있다면, 세 번째 인자는 네트워크 연결이 이루어질 때 한 번 호출되고 그 이후에 각 블록을 읽을 때마다 한 번씩 호출되는 콜러블입니다. 콜러블에는 세 개의 인자가 전달됩니다; 지금까지 전송된 블록 수, 바이트 단위의 블록 크기 및 파일의 전체 크기. 세 번째 인자는 가져오기 요청에 대한 응답으로 파일 크기를 반환하지 않는 구형 FTP 서버에서 `-1`일 수 있습니다.

다음 예는 가장 일반적인 사용 시나리오를 보여줍니다:

```
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

*url*이 `http:` 스킴 식별자를 사용하면, `POST` 요청을 지정하기 위해 선택적 *data* 인자가 제공될 수 있습니다 (일반적으로 요청형은 `GET`입니다). *data* 인자는 표준 *`application/x-www-form-urlencoded`* 형식의 바이트열 객체여야 합니다; *`urllib.parse.urlencode()`* 함수를 참조하십시오.

*`urlretrieve()`*는 사용 가능한 데이터양이 예상 양(*Content-Length* 헤더에 의해 보고된 크기)보다 작은 것을 감지하면 *`ContentTooShortError`*를 발생시킵니다. 예를 들어, 다운로드가 중단된 경우에 발생할 수 있습니다.

*Content-Length*는 하한값으로 취급됩니다: 읽을 데이터가 더 있으면, urlretrieve는 더 많은 데이터를 읽지만, 사용 가능한 데이터가 부족하면 예외가 발생합니다.

이 경우에도 다운로드된 데이터를 여전히 가져올 수 있으며, 예외 인스턴스의 `content` 어트리뷰트에 저장됩니다.

Content-Length 헤더가 제공되지 않으면, urlretrieve는 다운로드 한 데이터의 크기를 확인할 수 없고, 그냥 반환합니다. 이 경우 다운로드가 성공했다고 가정해야 합니다.

`urllib.request.`**`urlcleanup`**()

*`urlretrieve()`*에 대한 이전 호출로 남겨졌을 수 있는 임시 파일을 정리합니다.

22.4.25 urllib.request 제약 사항

- 현재, 다음과 같은 프로토콜만 지원됩니다: HTTP (버전 0.9와 1.0), FTP, 로컬 파일 및 데이터 URL.

  버전 3.4에서 변경: 데이터 URL에 대한 지원이 추가되었습니다.

- *urlretrieve()*의 캐싱 기능은 누군가가 만료 시간 헤더의 적절한 처리를 해킹할 시간을 찾을 때까지 비활성화되었습니다.
- 특정 URL이 캐시에 있는지를 조회하는 함수가 있어야 합니다.
- 이전 버전과의 호환성을 위해, URL이 로컬 파일을 가리키는 것으로 보이지만 파일을 열 수 없으면, FTP 프로토콜을 사용하여 URL을 다시 해석합니다. 이로 인해 때때로 혼란스러운 에러 메시지가 발생할 수 있습니다.
- *urlopen()*과 *urlretrieve()* 함수는 네트워크 연결이 이루어지기를 기다리는 동안 임의의 긴 지연을 유발할 수 있습니다. 이는 스레드를 사용하지 않고 이러한 함수를 사용하여 대화식 웹 클라이언트를 구축하기 어렵다는 것을 뜻합니다.
- *urlopen()*이나 *urlretrieve()*가 반환한 데이터는 서버가 반환한 원시 데이터입니다. 바이너리 데이터 (가령 이미지), 평문 텍스트 또는 (예를 들어) HTML일 수 있습니다. HTTP 프로토콜은 응답 헤더에 유형 정보를 제공하는데, *Content-Type* 헤더를 통해 검사할 수 있습니다. 반환된 데이터가 HTML이면, *html.parser* 모듈을 사용하여 구문 분석할 수 있습니다.
- The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a `/`, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing `/` has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the *ftplib* module.

22.5 urllib.response — urllib가 사용하는 응답 클래스

urllib.response 모듈은 `read()`와 `readline()`을 포함하여 최소한의 파일류 인터페이스를 정의하는 함수와 클래스를 정의합니다. 이 모듈에 의해 정의된 함수는 *urllib.request* 모듈에 의해 내부적으로 사용됩니다. 일반적인 응답 객체는 *urllib.response.addinfourl* 인스턴스입니다:

class urllib.response.**addinfourl**

url

가져온 자원의 URL, 일반적으로 리디렉션을 따라갔는지 판별하는 데 사용됩니다.

headers

EmailMessage 인스턴스 형식으로 응답의 헤더를 반환합니다.

status

Added in version 3.9.

서버가 반환한 상태 코드.

geturl()

버전 3.9부터 폐지됨: 폐지되었고 *url*로 대체되었습니다.

info()

버전 3.9부터 폐지됨: 폐지되었고 *headers*로 대체되었습니다.

code

버전 3.9부터 폐지됨: 폐지되었고 *status*로 대체되었습니다.

getcode()

버전 3.9부터 폐지됨: 폐지되었고 *status*로 대체되었습니다.

22.6 urllib.parse — URL을 구성 요소로 구문 분석

소스 코드: Lib/urllib/parse.py

이 모듈은 URL(Uniform Resource Locator) 문자열을 구성 요소(주소 지정 체계, 네트워크 위치, 경로 등)로 분리하고, 구성 요소를 다시 URL 문자열로 결합하고, "상대 URL"을 주어진 "기본 URL"에 따라 절대 URL로 변환하는 표준 인터페이스를 정의합니다.

이 모듈은 상대 URL(Relative Uniform Resource Locators)의 인터넷 RFC와 일치하도록 설계되었습니다. 다음과 같은 URL 스킴을 지원합니다: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `itms-services`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtsps`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`, `ws`, `wss`.

CPython 구현 상세: The inclusion of the `itms-services` URL scheme can prevent an app from passing Apple' s App Store review process for the macOS and iOS App Stores. Handling for the `itms-services` scheme is always removed on iOS; on macOS, it *may* be removed if CPython has been built with the `--with-app-store-compliance` option.

urllib.parse 모듈은 두 가지 넓은 범주에 속하는 함수들을 정의합니다: URL 구문 분석과 URL 인용(quoting). 이에 대해서는 다음 섹션에서 자세히 설명합니다.

This module' s functions use the deprecated term `netloc` (or `net_loc`), which was introduced in **RFC 1808**. However, this term has been obsoleted by **RFC 3986**, which introduced the term `authority` as its replacement. The use of `netloc` is continued for backward compatibility.

22.6.1 URL 구문 분석

URL 구문 분석 함수는 URL 문자열을 구성 요소로 분할하거나 URL 구성 요소를 URL 문자열로 결합하는 데 중점을 둡니다.

`urllib.parse.`**`urlparse`**(*urlstring*, *scheme=''*, *allow_fragments=True*)

URL을 6개의 구성 요소로 구문 분석하여, 6개 항목 네임드 튜플을 반환합니다. 이는 URL의 일반적인 구조에 해당합니다: `scheme://netloc/path;parameters?query#fragment`. 각 튜플 항목은 문자열이며 비어있을 수 있습니다. 구성 요소는 더 작은 부분으로 나뉘지 않으며 (예를 들어 네트워크 위치는 단일 문자열입니다), % 이스케이프는 확장되지 않습니다. 위에 표시된 분리 문자는 *path* 구성 요소의 선행 슬래시를 제외하고는 결과의 일부가 아니며 존재하면 유지됩니다. 예를 들면:

```
>>> from urllib.parse import urlparse
>>> urlparse("scheme://netloc/path;parameters?query#fragment")
ParseResult(scheme='scheme', netloc='netloc', path='/path;parameters', params='
↪',
            query='query', fragment='fragment')
>>> o = urlparse("http://docs.python.org:80/3/library/urllib.parse.html?"
...              "highlight=params#url-parsing")
>>> o
ParseResult(scheme='http', netloc='docs.python.org:80',
            path='/3/library/urllib.parse.html', params='',
            query='highlight=params', fragment='url-parsing')
>>> o.scheme
'http'
>>> o.netloc
'docs.python.org:80'
>>> o.hostname
'docs.python.org'
>>> o.port
80
>>> o._replace(fragment="").geturl()
'http://docs.python.org:80/3/library/urllib.parse.html?highlight=params'
```

RFC 1808의 문법 명세에 따라, urlparse는 '//' 로 올바르게 도입되었을 때만 netloc을 인식합니다. 그렇지 않으면 입력은 상대 URL인 것으로 간주하고 path 구성 요소로 시작합니다.

```
>>> from urllib.parse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
            query='', fragment='')
```

scheme 인자는 URL이 지정하지 않은 경우에만 사용될 기본 주소 지정 체계를 제공합니다. 기본값 `''`가 항상 허용되고 필요하면 자동으로 `b''`로 변환된다는 점을 제외하고, *urlstring*과 같은 형(텍스트나 바이트열)이어야 합니다.

allow_fragments 인자가 거짓이면, 프래그먼트 식별자는 인식되지 않습니다. 대신, path, parameters 또는 query 구성 요소의 일부로 구문 분석되고 `fragment`는 반환 값에서 빈 문자열로 설정됩니다.

반환 값은 네임드 튜플입니다. 즉, 다음과 같은 인덱스나 이름있는 어트리뷰트로 해당 항목에 액세스 할 수 있습니다:

어트리뷰트	인덱스	값	존재하지 않을 때의 값
`scheme`	0	URL 스킴 지정자	*scheme* 매개 변수
`netloc`	1	네트워크 위치 부분	빈 문자열
`path`	2	계층적 경로	빈 문자열
`params`	3	마지막 경로 요소의 파라미터	빈 문자열
`query`	4	쿼리 구성 요소	빈 문자열
`fragment`	5	프래그먼트 식별자	빈 문자열
`username`		사용자 이름	*None*
`password`		비밀번호	*None*
`hostname`		호스트 이름 (소문자)	*None*
`port`		존재하면, 정수로 표시되는 포트 번호	*None*

URL에 잘못된 포트가 지정된 경우 `port` 어트리뷰트를 읽으면 *ValueError*가 발생합니다. 결과 객체에 대한 자세한 내용은 구조화된 구문 분석 결과 섹션을 참조하십시오.

`netloc` 어트리뷰트에서 대괄호(square brackets)가 일치하지 않으면 *ValueError*가 발생합니다.

(IDNA 인코딩에서 사용되는) NFKC 정규화에서 `/`, `?`, `#`, `@` 또는 `:` 중 하나로 분해(decompose)되는 `netloc` 어트리뷰트의 문자는 *ValueError*를 발생시킵니다. 구문 분석하기 전에 URL이 분해(decompose)되면, 에러가 발생하지 않습니다.

모든 네임드 튜플의 경우와 마찬가지로, 서브 클래스는 특히 유용한 몇 가지 추가 메서드와 어트리뷰트를 갖습니다. 그러한 메서드 중 하나는 `_replace()` 입니다. `_replace()` 메서드는 지정된 필드를 새로운 값으로 대체한 새로운 ParseResult 객체를 반환합니다.

```
>>> from urllib.parse import urlparse
>>> u = urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
>>> u
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> u._replace(scheme='http')
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html
↪',
            params='', query='', fragment='')
```

> **경고**
>
> `urlparse()` does not perform validation. See *URL parsing security* for details.

버전 3.2에서 변경: IPv6 URL 구문 분석 기능이 추가되었습니다.

버전 3.3에서 변경: **RFC 3986**에 따라, 프래그먼트는 이제 모든 URL 스킴에 대해 구문 분석됩니다 (*allow_fragments*가 거짓이 아니라면). 이전에는, 프래그먼트를 지원하는 스킴의 수락 목록이 있었습니다.

버전 3.6에서 변경: 범위를 벗어난 포트 번호는 이제 `None`을 반환하는 대신 `ValueError`를 발생시킵니다.

버전 3.8에서 변경: NFKC 정규화에서 netloc 구문 분석에 영향을 주는 문자는 이제 `ValueError`를 발생시킵니다.

`urllib.parse.`**`parse_qs`**(*qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='replace', max_num_fields=None, separator='&'*)

문자열 인자(`application/x-www-form-urlencoded` 유형의 데이터)로 제공된 쿼리 문자열을 구문 분석합니다. 데이터는 딕셔너리로 반환됩니다. 딕셔너리 키는 고유한 쿼리 변수 이름이고 값은 각 이름의 값 리스트입니다.

선택적 인자 *keep_blank_values*는 퍼센트 인코딩된 쿼리의 빈 값을 빈 문자열로 처리해야 하는지를 나타내는 플래그입니다. 참값은 빈 값을 빈 문자열로 유지해야 함을 나타냅니다. 기본 거짓 값은 빈 값이 무시되고 포함되지 않은 것처럼 처리됨을 나타냅니다.

선택적 인자 *strict_parsing*은 구문 분석 에러 시 수행할 작업을 나타내는 플래그입니다. 거짓(기본값)이면, 에러가 조용히 무시됩니다. 참이면, 에러는 `ValueError` 예외를 발생시킵니다.

선택적 *encoding*과 *errors* 매개 변수는 `bytes.decode()` 메서드에서 받아들이는 것처럼 퍼센트 인코딩된 시퀀스를 유니코드 문자로 디코딩하는 방법을 지정합니다.

선택적 인자 *max_num_fields*는 읽을 최대 필드 수입니다. 설정되면, *max_num_fields* 필드보다 많은 것을 읽을 때 `ValueError`가 발생합니다.

선택적 인자 *separator*는 쿼리 인자를 구분하는 데 사용할 기호입니다. 기본값은 `&` 입니다.

이러한 딕셔너리를 쿼리 문자열로 변환하려면 `urllib.parse.urlencode()` 함수를 (`doseq` 매개 변수를 `True`로 설정해서) 사용하십시오.

버전 3.2에서 변경: *encoding*과 *errors* 매개 변수를 추가합니다.

버전 3.8에서 변경: *max_num_fields* 매개 변수를 추가했습니다.

버전 3.10에서 변경: 기본값이 `&` 인 *separator* 매개 변수를 추가했습니다. 파이썬 3.10 이전의 파이썬 버전에서는 쿼리 매개 변수 구분 기호로 `;` 와 `&`를 모두 사용할 수 있었습니다. `&`를 기본 구분자 기호로 사용하여, 단일 구분자 키만 허용하도록 변경되었습니다.

버전 3.14부터 폐지됨: Accepting objects with false values (like `0` and `[]`) except empty strings and byte-like objects and `None` is now deprecated.

`urllib.parse.`**`parse_qsl`**(*qs, keep_blank_values=False, strict_parsing=False, encoding='utf-8', errors='replace', max_num_fields=None, separator='&'*)

문자열 인자(`application/x-www-form-urlencoded` 유형의 데이터)로 제공된 쿼리 문자열을 구문 분석합니다. 데이터는 이름, 값 쌍의 리스트로 반환됩니다.

선택적 인자 *keep_blank_values*는 퍼센트 인코딩된 쿼리의 빈 값을 빈 문자열로 처리해야 하는지를 나타내는 플래그입니다. 참값은 빈 값을 빈 문자열로 유지해야 함을 나타냅니다. 기본 거짓 값은 빈 값이 무시되고 포함되지 않은 것처럼 처리됨을 나타냅니다.

선택적 인자 *strict_parsing*은 구문 분석 에러 시 수행할 작업을 나타내는 플래그입니다. 거짓(기본값)이면, 에러가 조용히 무시됩니다. 참이면, 에러는 `ValueError` 예외를 발생시킵니다.

선택적 *encoding*과 *errors* 매개 변수는 `bytes.decode()` 메서드에서 받아들이는 것처럼 퍼센트 인코딩된 시퀀스를 유니코드 문자로 디코딩하는 방법을 지정합니다.

선택적 인자 *max_num_fields*는 읽을 최대 필드 수입니다. 설정되면, *max_num_fields* 필드보다 많은 것을 읽을 때 `ValueError`가 발생합니다.

선택적 인자 *separator*는 쿼리 인자를 구분하는 데 사용할 기호입니다. 기본값은 `&` 입니다.

이러한 쌍의 리스트를 쿼리 문자열로 변환하려면 `urllib.parse.urlencode()` 함수를 사용하십시오.

버전 3.2에서 변경: *encoding*과 *errors* 매개 변수를 추가합니다.

버전 3.8에서 변경: *max_num_fields* 매개 변수를 추가했습니다.

버전 3.10에서 변경: 기본값이 `&` 인 *separator* 매개 변수를 추가했습니다. 파이썬 3.10 이전의 파이썬 버전에서는 쿼리 매개 변수 구분 기호로 `;` 와 `&`를 모두 사용할 수 있었습니다. `&`를 기본 구분자 기호로 사용하여, 단일 구분자 키만 허용하도록 변경되었습니다.

`urllib.parse.`**`urlunparse`**(*parts*)

`urlparse()` 에 의해 반환된 튜플에서 URL을 구성합니다. *parts* 인자는 임의의 6개 항목 이터러블일 수 있습니다. 구문 분석된 원래 URL에 불필요한 구분자가 있는 경우 (예를 들어 비어있는 쿼리가 있는 `?`; RFC는 이들이 동등하다고 말합니다) 약간 다르지만 동등한 URL이 만들어질 수 있습니다.

`urllib.parse.`**`urlsplit`**(*urlstring*, *scheme=''*, *allow_fragments=True*)

이것은 `urlparse()`와 유사하지만, URL에서 파라미터를 분할하지 않습니다. URL의 *path* 부분의 각 세그먼트에 파라미터를 적용할 수 있는 최신 URL 문법(**RFC 2396**을 참조하십시오)이 필요하면 일반적으로 `urlparse()` 대신 사용해야 합니다. 경로 세그먼트와 파라미터를 분리하려면 별도의 함수가 필요합니다. 이 함수는 5개 항목 네임드 튜플을 반환합니다:

```
(addressing scheme, network location, path, query, fragment identifier).
```

반환 값은 네임드 튜플입니다, 항목은 인덱스나 이름 붙은 어트리뷰트로 액세스 할 수 있습니다:

어트리뷰트	인덱스	값	존재하지 않을 때의 값
`scheme`	0	URL 스킴 지정자	*scheme* 매개 변수
`netloc`	1	네트워크 위치 부분	빈 문자열
`path`	2	계층적 경로	빈 문자열
`query`	3	쿼리 구성 요소	빈 문자열
`fragment`	4	프래그먼트 식별자	빈 문자열
`username`		사용자 이름	`None`
`password`		비밀번호	`None`
`hostname`		호스트 이름 (소문자)	`None`
`port`		존재하면, 정수로 표시되는 포트 번호	`None`

URL에 잘못된 포트가 지정된 경우 `port` 어트리뷰트를 읽으면 `ValueError`가 발생합니다. 결과 객체에 대한 자세한 내용은 구조화된 구문 분석 결과 섹션을 참조하십시오.

`netloc` 어트리뷰트에서 대괄호(square brackets)가 일치하지 않으면 `ValueError`가 발생합니다.

(IDNA 인코딩에서 사용되는) NFKC 정규화에서 `/`, `?`, `#`, `@` 또는 `:` 중 하나로 분해(decompose)되는 `netloc` 어트리뷰트의 문자는 `ValueError`를 발생시킵니다. 구문 분석하기 전에 URL이 분해(decompose)되면, 에러가 발생하지 않습니다.

Following some of the WHATWG spec that updates RFC 3986, leading C0 control and space characters are stripped from the URL. `\n`, `\r` and tab `\t` characters are removed from the URL at any position.

> **경고**
>
> `urlsplit()` does not perform validation. See *URL parsing security* for details.

버전 3.6에서 변경: 범위를 벗어난 포트 번호는 이제 `None`을 반환하는 대신 `ValueError`를 발생시킵니다.

버전 3.8에서 변경: NFKC 정규화에서 netloc 구문 분석에 영향을 주는 문자는 이제 `ValueError`를 발생시킵니다.

버전 3.10에서 변경: ASCII newline and tab characters are stripped from the URL.

버전 3.12에서 변경: Leading WHATWG C0 control and space characters are stripped from the URL.

`urllib.parse.`**`urlunsplit`**(*parts*)

`urlsplit()`에 의해 반환된 튜플 요소를 완전한 URL 문자열로 결합합니다. *parts* 인자는 임의의 5개 항목 이터러블일 수 있습니다. 구문 분석된 원래 URL에 불필요한 구분자가 있는 경우 (예를 들어 비어있는 쿼리가 있는 ?; RFC는 이들이 동등하다고 말합니다) 약간 다르지만 동등한 URL이 만들어질 수 있습니다.

`urllib.parse.`**`urljoin`**(*base, url, allow_fragments=True*)

"기본 URL" (*base*)을 다른 URL(*url*)과 결합하여 전체 ("절대") URL을 구성합니다. 비형식적으로, 이것은 기본 URL의 구성 요소, 특히 주소 지정 체계, 네트워크 위치 및 경로(의 일부)를 사용하여 상대 URL에 누락된 구성 요소를 제공합니다. 예를 들면:

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

allow_fragments 인자는 `urlparse()`와 같은 의미와 기본값을 갖습니다.

참고

*url*이 절대 URL이면 (즉, `//`나 `scheme://`로 시작하면), *url*의 호스트명 및/또는 스킴이 결과에 나타납니다. 예를 들면:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

이런 동작을 원하지 않으면, *url*을 `urlsplit()`과 `urlunsplit()`으로 사전 처리하여 가능한 *scheme*과 *netloc* 부분을 제거하십시오.

경고

Because an absolute URL may be passed as the `url` parameter, it is generally **not secure** to use `urljoin` with an attacker-controlled `url`. For example in, `urljoin("https://website.com/users/", username)`, if `username` can contain an absolute URL, the result of `urljoin` will be the absolute URL.

버전 3.5에서 변경: **RFC 3986**에 정의된 의미론과 일치하도록 동작이 갱신되었습니다.

`urllib.parse.`**`urldefrag`**(*url*)

*url*에 프래그먼트 식별자가 포함되면, 프래그먼트 식별자 없는 *url*의 수정된 버전과 프래그먼트 식별자를 별도의 문자열로 반환합니다. *url*에 프래그먼트 식별자가 없으면, 수정되지 않은 *url*과 빈 문자열을 반환합니다.

반환 값은 네임드 튜플입니다, 항목은 인덱스나 이름 붙은 어트리뷰트로 액세스 할 수 있습니다:

어트리뷰트	인덱스	값	존재하지 않을 때의 값
`url`	0	프래그먼트 없는 URL	빈 문자열
`fragment`	1	프래그먼트 식별자	빈 문자열

결과 객체에 대한 자세한 정보는 섹션 구조화된 구문 분석 결과를 참조하십시오.

버전 3.2에서 변경: 결과는 단순한 2-튜플이 아닌 구조화된 객체입니다.

urllib.parse.**unwrap**(*url*)

래핑 된 URL(즉, <URL:scheme://host/path>, <scheme://host/path>, URL:scheme://host/path 또는 scheme://host/path 형식의 문자열)에서 URL을 추출합니다. *url*이 래핑 된 URL이 아니면, 변경 없이 반환됩니다.

22.6.2 URL 구문 분석 보안

The *urlsplit()* and *urlparse()* APIs do not perform **validation** of inputs. They may not raise errors on inputs that other applications consider invalid. They may also succeed on some inputs that might not be considered URLs elsewhere. Their purpose is for practical functionality rather than purity.

Instead of raising an exception on unusual input, they may instead return some component parts as empty strings. Or components may contain more than perhaps they should.

We recommend that users of these APIs where the values may be used anywhere with security implications code defensively. Do some verification within your code before trusting a returned component part. Does that scheme make sense? Is that a sensible path? Is there anything strange about that hostname? etc.

What constitutes a URL is not universally well defined. Different applications have different needs and desired constraints. For instance the living WHATWG spec describes what user facing web clients such as a web browser require. While **RFC 3986** is more general. These functions incorporate some aspects of both, but cannot be claimed compliant with either. The APIs and existing user code with expectations on specific behaviors predate both standards leading us to be very cautious about making API behavior changes.

22.6.3 ASCII로 인코딩된 바이트열 구문 분석

URL 구문 분석 함수는 원래 문자열에서만 작동하도록 설계되었습니다. 실제로는, 적절히 인용되고 인코딩된 URL을 ASCII 바이트 시퀀스로 조작할 수 있으면 유용합니다. 따라서, 이 모듈의 URL 구문 분석 함수는 모두 *str* 객체 외에 *bytes*와 *bytearray* 객체에서 작동합니다.

str 데이터가 전달되면, 결과에는 *str* 데이터만 포함됩니다. *bytes*나 *bytearray* 데이터가 전달되면, 결과에는 *bytes* 데이터만 포함됩니다.

단일 함수 호출에서 *str* 데이터를 *bytes*나 *bytearray*와 혼합하려고 시도하면 *TypeError*가 발생하는 반면, 비 ASCII 바이트 값을 전달하면 *UnicodeDecodeError*가 트리거 됩니다.

*str*과 *bytes* 간에 결과 객체를 쉽게 변환 할 수 있도록, URL 구문 분석 함수의 모든 반환 값은 encode() 메서드(결과에 *str* 데이터가 포함될 때)나 decode() 메서드(결과에 *bytes* 데이터가 포함될 때)를 제공합니다. 이러한 메서드의 서명은 해당 *str*와 *bytes* 메서드의 서명과 일치합니다(기본 인코딩이 'utf-8'가 아니라 'ascii'라는 점은 예외입니다). 각각은 *bytes* 데이터(encode() 메서드의 경우)나 *str* 데이터(decode() 메서드의 경우)를 포함하는 해당 형의 값을 생성합니다.

ASCII가 아닌 데이터를 포함할 수 있는 잘못 인용된 URL에서 작동할 가능성이 있는 응용 프로그램은 URL 구문 분석 메서드를 호출하기 전에 바이트열에서 문자로 자체 디코딩을 수행할 필요가 있습니다.

이 섹션에서 설명하는 동작은 URL 구문 분석 함수에만 적용됩니다. URL 인용 함수는 개별 URL 인용 함수의 설명서에 기술된 대로 바이트 시퀀스를 생성하거나 소비할 때 자체 규칙을 사용합니다.

버전 3.2에서 변경: URL 구문 분석 함수는 이제 ASCII 인코딩된 바이트 시퀀스를 받아들입니다.

22.6.4 구조화된 구문 분석 결과

urlparse(), *urlsplit()* 및 *urldefrag()* 함수의 결과 객체는 *tuple* 형의 서브 클래스입니다. 이 서브 클래스는 해당 함수에 대한 설명서에 나열된 어트리뷰트, 이전 섹션에서 설명한 인코딩과 디코딩 지원 및 추가 메서드를 추가합니다:

urllib.parse.SplitResult.**geturl**()

원래 URL의 재결합된 버전을 문자열로 반환합니다. 스킴이 소문자로 정규화되고 빈 구성 요소가 삭제될 수 있다는 점에서 원래 URL과 다를 수 있습니다. 특히, 빈 파라미터, 쿼리 및 프래그먼트 식별자가 제거됩니다.

urldefrag() 결과의 경우, 빈 프래그먼트 식별자만 제거됩니다. *urlsplit()*과 *urlparse()* 결과의 경우, 이 메서드가 반환한 URL에 대해 언급된 모든 변경 사항이 적용됩니다.

이 메서드의 결과는 원래 구문 분석 함수를 통해 다시 전달될 때 변경되지 않은 상태로 유지됩니다:

```
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

다음 클래스는 `str` 객체에서 작동할 때 구조화된 구문 분석 결과의 구현을 제공합니다:

class urllib.parse.**DefragResult**(*url*, *fragment*)

`str` 데이터를 포함하는 `urldefrag()` 결과의 구상 클래스. `encode()` 메서드는 `DefragResultBytes` 인스턴스를 반환합니다.

Added in version 3.2.

class urllib.parse.**ParseResult**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

`str` 데이터를 포함하는 `urlparse()` 결과의 구상 클래스. `encode()` 메서드는 `ParseResultBytes` 인스턴스를 반환합니다.

class urllib.parse.**SplitResult**(*scheme*, *netloc*, *path*, *query*, *fragment*)

`str` 데이터를 포함하는 `urlsplit()` 결과의 구상 클래스. `encode()` 메서드는 `SplitResultBytes` 인스턴스를 반환합니다.

다음 클래스는 `bytes`나 `bytearray` 객체에서 작동할 때 구문 분석 결과의 구현을 제공합니다:

class urllib.parse.**DefragResultBytes**(*url*, *fragment*)

`bytes` 데이터를 포함하는 `urldefrag()` 결과의 구상 클래스. `decode()` 메서드는 `DefragResult` 인스턴스를 반환합니다.

Added in version 3.2.

class urllib.parse.**ParseResultBytes**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)

`bytes` 데이터를 포함하는 `urlparse()` 결과의 구상 클래스. `decode()` 메서드는 `ParseResult` 인스턴스를 반환합니다.

Added in version 3.2.

class urllib.parse.**SplitResultBytes**(*scheme*, *netloc*, *path*, *query*, *fragment*)

`bytes` 데이터를 포함하는 `urlsplit()` 결과의 구상 클래스. `decode()` 메서드는 `SplitResult` 인스턴스를 반환합니다.

Added in version 3.2.

22.6.5 URL 인용

URL 인용(quoting) 함수는 특수 문자를 인용하고 비 ASCII 텍스트를 적절히 인코딩하여 프로그램 데이터를 취해서 URL 구성 요소로 안전하게 사용할 수 있도록 하는 데 중점을 둡니다. 또한 해당 작업이 위의 URL 구문 분석 함수로 처리되지 않는 경우 URL 구성 요소의 내용에서 원래 데이터를 다시 만들기 위해 이러한 작업을 뒤집는 것도 지원합니다.

urllib.parse.**quote**(*string*, *safe='/'*, *encoding=None*, *errors=None*)

`%xx` 이스케이프를 사용하여 *string*의 특수 문자를 치환합니다. 글자, 숫자 및 문자 `'_.-~'`는 절대 인용되지 않습니다. 기본적으로, 이 함수는 URL의 경로 섹션을 인용하기 위한 것입니다. 선택적 *safe* 매개 변수는 인용해서는 안 되는 추가 ASCII 문자를 지정합니다 — 기본값은 `'/'`입니다.

*string*은 `str`이나 `bytes` 객체일 수 있습니다.

버전 3.7에서 변경: URL 문자열 인용을 **RFC 2396**에서 **RFC 3986**으로 옮겼습니다. "~"는 이제 예약되지 않은 문자 집합에 포함됩니다.

선택적 *encoding*과 *errors* 매개 변수는 `str.encode()` 메서드에서 받아들이는 것처럼 비 ASCII 문자를 처리하는 방법을 지정합니다. *encoding*의 기본값은 `'utf-8'`입니다. *errors*의 기본값은 `'strict'`로, 지원되지 않는 문자는 `UnicodeEncodeError`를 발생시킴을 의미합니다. *string*이 `bytes`이면 *encoding*과 *errors*를 제공해서는 안 됩니다, 그렇지 않으면 `TypeError`가 발생합니다.

`quote(string, safe, encoding, errors)`는 `quote_from_bytes(string.encode(encoding, errors), safe)`와 동등함에 유의하십시오.

예: `quote('/El Niño/')`는 `'/El%20Ni%C3%B1o/'`를 산출합니다.

urllib.parse.**quote_plus**(*string*, *safe=''*, *encoding=None*, *errors=None*)

`quote()`와 유사하지만, URL로 이동하기 위한 쿼리 문자열을 만들 때 HTML 폼값을 인용하는 데 필요한 대로 스페이스를 더하기 부호로 치환하기도 합니다. *safe*에 포함되지 않으면 원래 문자열의 더하기 부호가 이스케이프 됩니다. 또한 *safe*의 기본값은 `'/'`가 아닙니다.

예: `quote_plus('/El Niño/')`는 `'%2FEl+Ni%C3%B1o%2F'`를 산출합니다.

urllib.parse.**quote_from_bytes**(*bytes*, *safe='/'*)

`quote()`와 유사하지만, `str` 대신 `bytes` 객체를 받아들이고, 문자열을 바이트열로 인코딩하지 않습니다.

예: `quote_from_bytes(b'a&\xef')`는 `'a%26%EF'`를 산출합니다.

urllib.parse.**unquote**(*string*, *encoding='utf-8'*, *errors='replace'*)

`%xx` 이스케이프를 동등한 단일 문자로 대체합니다. 선택적 *encoding*과 *errors* 매개 변수는 `bytes.decode()` 메서드에서 받아들이는 것처럼 퍼센트 인코딩된 시퀀스를 유니코드 문자로 디코딩하는 방법을 지정합니다.

*string*은 `str`이나 `bytes` 객체일 수 있습니다.

*encoding*의 기본값은 `'utf-8'`입니다. *errors*의 기본값은 `'replace'`로, 유효하지 않은 시퀀스는 자리 표시자 문자(placeholder character)로 대체됩니다.

예: `unquote('/El%20Ni%C3%B1o/')`는 `'/El Niño/'`를 산출합니다.

버전 3.9에서 변경: *string* 매개 변수는 바이트열과 문자열 객체를 지원합니다 (이전에는 문자열만 지원했습니다).

urllib.parse.**unquote_plus**(*string*, *encoding='utf-8'*, *errors='replace'*)

`unquote()`와 유사하지만, HTML 폼값을 인용 해제할 때 요구되는 것처럼, 더하기 부호를 스페이스로 치환하기도 합니다.

*string*은 `str`이어야 합니다.

예: `unquote_plus('/El+Ni%C3%B1o/')`는 `'/El Niño/'`를 산출합니다.

urllib.parse.**unquote_to_bytes**(*string*)

`%xx` 이스케이프를 해당 단일 옥텟(octet)으로 대체하고, `bytes` 객체를 반환합니다.

*string*은 `str`이나 `bytes` 객체일 수 있습니다.

`str`이면, *string*의 이스케이프 되지 않은 비 ASCII 문자는 UTF-8 바이트열로 인코딩됩니다.

예: `unquote_to_bytes('a%26%EF')`는 `b'a&\xef'`를 산출합니다.

urllib.parse.**urlencode**(*query*, *doseq=False*, *safe=''*, *encoding=None*, *errors=None*, *quote_via=quote_plus*)

`str`이나 `bytes` 객체를 포함할 수 있는 매핑 객체나 두 요소 튜플의 시퀀스를 퍼센트 인코딩된 ASCII 텍스트 문자열로 변환합니다. 결과 문자열을 `urlopen()` 함수를 사용하여 POST 연산을 위한 *data*로 사용하려면, 바이트열로 인코딩해야 합니다, 그렇지 않으면 `TypeError`가 발생합니다.

결과 문자열은 `'&'` 문자로 구분된 일련의 `key=value` 쌍이고, 여기서 *key*와 *value*는 *quote_via* 함수를 사용하여 인용됩니다. 기본적으로, `quote_plus()`가 값을 인용하는 데 사용되는데, 스페이스는 `'+'` 문자로 인용되고 '/' 문자는 `%2F`로 인코딩되어 GET 요청 표준(`application/x-www-form-urlencoded`)을 따름을 뜻합니다. *quote_via*로 전달될 수 있는 대체 함수는 `quote()`이며, 스페이스를 `%20`로 인코딩하고 '/' 문자를 인코딩하지 않습니다. 무엇을 인용할지를 최대한 제어하려면, `quote`를 사용하고 *safe*의 값을 지정하십시오.

query 인자에 두 요소 튜플의 시퀀스가 사용될 때, 각 튜플의 첫 번째 요소는 키이고 두 번째 요소는 값입니다. 값 요소 자체는 시퀀스일 수 있으며, 이 경우 선택적 매개 변수 *doseq*가 `True`로 평가되면, `'&'`로 구분된 개별 `key=value` 쌍이 키에 대한 값 시퀀스의 각 요소에 대해 생성됩니다. 인코딩된 문자열의 파라미터 순서는 시퀀스의 파라미터 튜플 순서와 일치합니다.

safe, *encoding* 및 *errors* 매개 변수는 *quote_via*로 전달됩니다 (*encoding*과 *errors* 매개 변수는 쿼리 요소가 `str`일 때만 전달됩니다).

이 인코딩 프로세스를 뒤집기 위해, 쿼리 문자열을 파이썬 데이터 구조로 구문 분석하기 위해 이 모듈에서 `parse_qs()`와 `parse_qsl()`이 제공됩니다.

`urllib.parse.urlencode()` 메서드를 사용하여 URL의 쿼리 문자열이나 POST 요청의 데이터를 생성하는 방법을 알아보려면 *urllib* 예제를 참조하십시오.

버전 3.2에서 변경: *query* 매개 변수는 바이트열과 문자열 객체를 지원합니다.

버전 3.5에서 변경: *quote_via* 매개 변수를 추가했습니다.

버전 3.14부터 폐지됨: Accepting objects with false values (like `0` and `[]`) except empty strings and byte-like objects and `None` is now deprecated.

더 보기

WHATWG - URL Living standard
Working Group for the URL Standard that defines URLs, domains, IP addresses, the application/x-www-form-urlencoded format, and their API.

RFC 3986 - Uniform Resource Identifiers
이것이 현재 표준입니다 (STD66). urllib.parse 모듈에 대한 모든 변경 사항은 이를 준수해야 합니다. 특정 편차가 관찰될 수 있는데, 이는 대부분 이전 버전과의 호환성과 주요 브라우저에서 일반적으로 관찰되는 사실상의 구문 분석 요구 사항을 위한 것입니다.

RFC 2732 - Format for Literal IPv6 Addresses in URL' s.
IPv6 URL의 구문 분석 요구 사항을 지정합니다.

RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax
URN(Uniform Resource Names)과 URL(Uniform Resource Locator)에 대한 일반적인 문법 요구 사항을 설명하는 문서.

RFC 2368 - The mailto URL scheme.
mailto URL 스킴에 대한 구문 분석 요구 사항.

RFC 1808 - Relative Uniform Resource Locators
이 RFC는 경계 사례의 처리를 정의하는 꽤 많은 수의 "비정상적인 예"를 포함하여, 절대와 상대 URL을 결합하는 규칙을 포함합니다.

RFC 1738 - Uniform Resource Locators (URL)
절대 URL의 형식 문법과 의미를 지정합니다.

22.7 `urllib.error` — urllib.request에 의해 발생하는 예외 클래스

소스 코드: Lib/urllib/error.py

`urllib.error`모듈은 `urllib.request`에 의해 발생하는 예외에 대한 예외 클래스를 정의합니다. 베이스 예외 클래스는 `URLError`입니다.

`urllib.error`에 의해 다음과 같은 예외가 적절하게 발생합니다.

exception `urllib.error.`**`URLError`**

처리기가 문제에 봉착하는 경우, 처리기는 해당 예외(또는 파생된 예외)를 발생시킵니다. 이 예외는 `OSError`의 서브 클래스입니다.

reason

이 에러가 발생한 원인입니다. 메시지 문자열이거나 다른 예외 인스턴스가 될 수 있습니다.

버전 3.3에서 변경: 이전에는 `URLError`가 `IOError`의 하위 형이었으나, 이제는 `OSError`의 별칭이 되었습니다.

exception `urllib.error.`**HTTPError**(*url*, *code*, *msg*, *hdrs*, *fp*)

`HTTPError`는 `URLError`의 서브 클래스로 예외 클래스이긴 하지만, 예외가 아닌 파일류 반환 값(`urlopen()`의 반환 값과 동일한 값)으로도 작동할 수 있습니다. 이 방법은 인증 요청 같은 독특한 (exotic) HTTP 에러를 처리할 때 유용합니다.

url

Contains the request URL. An alias for *filename* attribute.

code

RFC 2616에 정의된 HTTP 상태 코드입니다. 이 숫자 값은 `http.server.BaseHTTPRequestHandler.responses`에서 찾을 수 있는 상태 코드 딕셔너리에 있는 값에 해당합니다.

reason

일반적으로 이 에러의 원인을 설명하는 문자열입니다. *msg* 어트리뷰트의 별칭.

headers

`HTTPError`를 발생시킨 HTTP 요청의 응답 헤더입니다. *hdrs* 어트리뷰트의 별칭.

Added in version 3.4.

fp

A file-like object where the HTTP error body can be read from.

exception `urllib.error.`**ContentTooShortError**(*msg*, *content*)

이 예외는 다운로드받은 데이터양이 *Content-Length* 헤더 값을 통해 예상한 양보다 적은 것을 `urlretrieve()` 함수가 감지했을 때 발생합니다.

content

다운로드받은 (그리고 아마도 잘린) 데이터.

22.8 `urllib.robotparser` — robots.txt 구문 분석기

소스 코드: Lib/urllib/robotparser.py

이 모듈은 클래스 하나 `RobotFileParser`를 제공하는데, 특정 사용자 에이전트가 `robots.txt` 파일을 게시한 웹 사이트에서 URL을 가져올 수 있는지에 대한 질문에 대답합니다. `robots.txt` 파일의 구조에 대한 자세한 내용은 http://www.robotstxt.org/orig.html 을 참조하십시오.

class `urllib.robotparser.`**RobotFileParser**(*url=''*)

이 클래스는 *url*에 있는 `robots.txt` 파일을 읽고, 구문 분석하고, 그에 대한 질문에 대답하는 메서드를 제공합니다.

set_url(*url*)

`robots.txt` 파일을 가리키는 URL을 설정합니다.

read()

`robots.txt` URL을 읽어서 구문 분석기에 넘깁니다.

parse(*lines*)

lines 인자를 구문 분석합니다.

can_fetch(*useragent*, *url*)

구문 분석된 robots.txt 파일에 포함된 규칙에 따라, *useragent*가 *url*를 가져올 수 있으면 True를 반환합니다.

mtime()

robots.txt 파일을 마지막으로 가져온 시간을 반환합니다. 이것은 새 robots.txt 파일을 주기적으로 확인해야 하는 장기 실행 웹 스파이더에 유용합니다.

modified()

robots.txt 파일을 마지막으로 가져온 시간을 현재 시각으로 설정합니다.

crawl_delay(*useragent*)

robots.txt에서 해당 *useragent*에 대한 Crawl-delay 파라미터의 값을 반환합니다. 해당 파라미터가 없거나, 지정된 *useragent*에 적용되지 않거나, 이 파라미터에 대한 robots.txt 항목이 잘못된 구문이면 None을 반환합니다.

Added in version 3.6.

request_rate(*useragent*)

robots.txt에서 Request-rate 파라미터의 내용을 네임드 튜플 RequestRate(requests, seconds)로 반환합니다. 해당 파라미터가 없거나, 지정된 *useragent*에 적용되지 않거나, 이 파라미터에 대한 robots.txt 항목이 잘못된 구문이면 None을 반환합니다.

Added in version 3.6.

site_maps()

robots.txt에서 Sitemap 매개 변수의 내용을 *list()* 형식으로 반환합니다. 해당 매개 변수가 없거나 robots.txt의 이 매개 변수 항목이 잘못된 문법이면 None을 반환합니다.

Added in version 3.8.

다음 예제는 *RobotFileParser* 클래스의 기본 사용을 보여줍니다:

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rrate = rp.request_rate("*")
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("*")
6
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

22.9 http — HTTP 모듈

소스 코드: Lib/http/__init__.py

---

*http*는 하이퍼텍스트 전송 프로토콜로 작업 하기 위한 여러 모듈을 수집하는 패키지입니다:

- *http.client*는 저수준 HTTP 프로토콜 클라이언트입니다. 고수준의 URL 열기는 *urllib.request*를 사용합니다
- *http.server*는 *socketserver*에 기반을 둔 기본적인 HTTP 서버 클래스를 포함합니다
- *http.cookies*는 쿠키를 사용하여 상태 관리를 구현하는 유틸리티가 있습니다

- *http.cookiejar*는 쿠키의 지속성을 제공합니다

또한 *http* 모듈은 http 관련 코드 작업에 도움이 되는 다음 열거형을 정의합니다:

class http.**HTTPStatus**

Added in version 3.5.

HTTP 상태 코드, 이유 구문 그리고 긴 영문 설명의 집합을 정의하는 *enum.IntEnum*의 서브 클래스입니다.

사용법:

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK
HTTPStatus.OK
>>> HTTPStatus.OK == 200
True
>>> HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[HTTPStatus.CONTINUE, HTTPStatus.SWITCHING_PROTOCOLS, ...]
```

22.9.1 HTTP 상태 코드

*http.HTTPStatus*에서 지원하는 IANA에 등록된 상태 코드는 다음과 같습니다:

코드	열거 이름	세부 사항
100	CONTINUE	HTTP Semantics **RFC 9110**, 섹션 15.2.1
101	SWITCHING_PROTOCOLS	HTTP Semantics **RFC 9110**, 섹션 15.2.2
102	PROCESSING	WebDAV **RFC 2518**, 섹션 10.1
103	EARLY_HINTS	힌트를 나타내는 HTTP 상태 코드 **RFC 8297**
200	OK	HTTP Semantics **RFC 9110**, 섹션 15.3.1
201	CREATED	HTTP Semantics **RFC 9110**, 섹션 15.3.2
202	ACCEPTED	HTTP Semantics **RFC 9110**, 섹션 15.3.3
203	NON_AUTHORITATIVE_INFORMATION	HTTP Semantics **RFC 9110**, 섹션 15.3.4
204	NO_CONTENT	HTTP Semantics **RFC 9110**, 섹션 15.3.5
205	RESET_CONTENT	HTTP Semantics **RFC 9110**, 섹션 15.3.6
206	PARTIAL_CONTENT	HTTP Semantics **RFC 9110**, 섹션 15.3.7
207	MULTI_STATUS	WebDAV **RFC 4918**, 섹션 11.1
208	ALREADY_REPORTED	WebDAV 바인딩 확장 **RFC 5842**, 섹션 7.1 (실험적)
226	IM_USED	HTTP의 델타 인코딩 **RFC 3229**, 섹션 10.4.1
300	MULTIPLE_CHOICES	HTTP Semantics **RFC 9110**, 섹션 15.4.1
301	MOVED_PERMANENTLY	HTTP Semantics **RFC 9110**, 섹션 15.4.2
302	FOUND	HTTP Semantics **RFC 9110**, 섹션 15.4.3
303	SEE_OTHER	HTTP Semantics **RFC 9110**, 섹션 15.4.4
304	NOT_MODIFIED	HTTP Semantics **RFC 9110**, 섹션 15.4.5
305	USE_PROXY	HTTP Semantics **RFC 9110**, 섹션 15.4.6
307	TEMPORARY_REDIRECT	HTTP Semantics **RFC 9110**, 섹션 15.4.8
308	PERMANENT_REDIRECT	HTTP Semantics **RFC 9110**, 섹션 15.4.9
400	BAD_REQUEST	HTTP Semantics **RFC 9110**, 섹션 15.5.1
401	UNAUTHORIZED	HTTP Semantics **RFC 9110**, 섹션 15.5.2
402	PAYMENT_REQUIRED	HTTP Semantics **RFC 9110**, 섹션 15.5.3
403	FORBIDDEN	HTTP Semantics **RFC 9110**, 섹션 15.5.4
404	NOT_FOUND	HTTP Semantics **RFC 9110**, 섹션 15.5.5

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

코드	열거 이름	세부 사항
405	METHOD_NOT_ALLOWED	HTTP Semantics **RFC 9110**, 섹션 15.5.6
406	NOT_ACCEPTABLE	HTTP Semantics **RFC 9110**, 섹션 15.5.7
407	PROXY_AUTHENTICATION_REQUIRED	HTTP Semantics **RFC 9110**, 섹션 15.5.8
408	REQUEST_TIMEOUT	HTTP Semantics **RFC 9110**, 섹션 15.5.9
409	CONFLICT	HTTP Semantics **RFC 9110**, 섹션 15.5.10
410	GONE	HTTP Semantics **RFC 9110**, 섹션 15.5.11
411	LENGTH_REQUIRED	HTTP Semantics **RFC 9110**, 섹션 15.5.12
412	PRECONDITION_FAILED	HTTP Semantics **RFC 9110**, 섹션 15.5.13
413	CONTENT_TOO_LARGE	HTTP Semantics **RFC 9110**, 섹션 15.5.14
414	URI_TOO_LONG	HTTP Semantics **RFC 9110**, 섹션 15.5.15
415	UNSUPPORTED_MEDIA_TYPE	HTTP Semantics **RFC 9110**, 섹션 15.5.16
416	RANGE_NOT_SATISFIABLE	HTTP Semantics **RFC 9110**, 섹션 15.5.17
417	EXPECTATION_FAILED	HTTP Semantics **RFC 9110**, 섹션 15.5.18
418	IM_A_TEAPOT	HTCPCP/1.0 **RFC 2324**, 섹션 2.3.2
421	MISDIRECTED_REQUEST	HTTP Semantics **RFC 9110**, 섹션 15.5.20
422	UNPROCESSABLE_CONTENT	HTTP Semantics **RFC 9110**, 섹션 15.5.21
423	LOCKED	WebDAV **RFC 4918**, 섹션 11.3
424	FAILED_DEPENDENCY	WebDAV **RFC 4918**, 섹션 11.4
425	TOO_EARLY	HTTP에서 초기 데이터(Early Data) 사용 **RFC 8470**
426	UPGRADE_REQUIRED	HTTP Semantics **RFC 9110**, 섹션 15.5.22
428	PRECONDITION_REQUIRED	추가 HTTP 상태 코드 **RFC 6585**
429	TOO_MANY_REQUESTS	추가 HTTP 상태 코드 **RFC 6585**
431	REQUEST_HEADER_FIELDS_TOO_LARGE	추가 HTTP 상태 코드 **RFC 6585**
451	UNAVAILABLE_FOR_LEGAL_REASONS	법적 장애를 보고하는 HTTP 상태 코드 **RFC 7725**
500	INTERNAL_SERVER_ERROR	HTTP Semantics **RFC 9110**, 섹션 15.6.1
501	NOT_IMPLEMENTED	HTTP Semantics **RFC 9110**, 섹션 15.6.2
502	BAD_GATEWAY	HTTP Semantics **RFC 9110**, 섹션 15.6.3
503	SERVICE_UNAVAILABLE	HTTP Semantics **RFC 9110**, 섹션 15.6.4
504	GATEWAY_TIMEOUT	HTTP Semantics **RFC 9110**, 섹션 15.6.5
505	HTTP_VERSION_NOT_SUPPORTED	HTTP Semantics **RFC 9110**, 섹션 15.6.6
506	VARIANT_ALSO_NEGOTIATES	HTTP의 투명한 콘텐츠 협상 **RFC 2295**, 섹션 8.1 (실험적)
507	INSUFFICIENT_STORAGE	WebDAV **RFC 4918**, 섹션 11.5
508	LOOP_DETECTED	WebDAV 바인딩 확장 **RFC 5842**, 섹션 7.2 (실험적)
510	NOT_EXTENDED	HTTP 확장 프레임워크 **RFC 2774**, 섹션 7 (실험적)
511	NETWORK_AUTHENTICATION_REQUIRED	추가 HTTP 상태 코드 **RFC 6585**, 섹션 6

이전 버전과의 호환성을 유지하기 위해 열거값은 *http.client* 모듈에 상수 형태로도 있습니다. 열거명과 상수명은 동일합니다 (즉, `http.HTTPStatus.OK`는 `http.client.OK`로도 사용 가능합니다).

버전 3.7에서 변경: `421 MISDIRECTED_REQUEST` 상태 코드 추가.

Added in version 3.8: `451 UNAVAILABLE_FOR_LEGAL_REASONS` 상태 코드가 추가.

Added in version 3.9: `103 EARLY_HINTS`, `418 IM_A_TEAPOT` 및 `425 TOO_EARLY` 상태 코드가 추가되었습니다.

버전 3.13에서 변경: Implemented RFC9110 naming for status constants. Old constant names are preserved for backwards compatibility.

22.9.2 HTTP 상태 범주

Added in version 3.12.

The enum values have several properties to indicate the HTTP status category:

프로퍼티	나타내는 것	세부 사항
`is_informational`	`100 <= status <= 199`	HTTP Semantics **RFC 9110**, 섹션 15
`is_success`	`200 <= status <= 299`	HTTP Semantics **RFC 9110**, 섹션 15
`is_redirection`	`300 <= status <= 399`	HTTP Semantics **RFC 9110**, 섹션 15
`is_client_error`	`400 <= status <= 499`	HTTP Semantics **RFC 9110**, 섹션 15
`is_server_error`	`500 <= status <= 599`	HTTP Semantics **RFC 9110**, 섹션 15

사용법:

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK.is_success
True
>>> HTTPStatus.OK.is_client_error
False
```

class http.**HTTPMethod**

Added in version 3.11.

HTTP 메서드와 영문 설명의 집합을 정의하는 *enum.StrEnum*의 서브 클래스입니다.

사용법:

```
>>> from http import HTTPMethod
>>>
>>> HTTPMethod.GET
<HTTPMethod.GET>
>>> HTTPMethod.GET == 'GET'
True
>>> HTTPMethod.GET.value
'GET'
>>> HTTPMethod.GET.description
'Retrieve the target.'
>>> list(HTTPMethod)
[<HTTPMethod.CONNECT>,
 <HTTPMethod.DELETE>,
 <HTTPMethod.GET>,
 <HTTPMethod.HEAD>,
 <HTTPMethod.OPTIONS>,
 <HTTPMethod.PATCH>,
 <HTTPMethod.POST>,
 <HTTPMethod.PUT>,
 <HTTPMethod.TRACE>]
```

22.9.3 HTTP 메서드

*http.HTTPMethod*에서 지원하는 IANA에 등록된 메서드는 다음과 같습니다:

메서드	열거 이름	세부 사항
`GET`	`GET`	HTTP Semantics **RFC 9110**, 섹션 9.3.1
`HEAD`	`HEAD`	HTTP Semantics **RFC 9110**, 섹션 9.3.2
`POST`	`POST`	HTTP Semantics **RFC 9110**, 섹션 9.3.3
`PUT`	`PUT`	HTTP Semantics **RFC 9110**, 섹션 9.3.4
`DELETE`	`DELETE`	HTTP Semantics **RFC 9110**, 섹션 9.3.5
`CONNECT`	`CONNECT`	HTTP Semantics **RFC 9110**, 섹션 9.3.6
`OPTIONS`	`OPTIONS`	HTTP Semantics **RFC 9110**, 섹션 9.3.7
`TRACE`	`TRACE`	HTTP Semantics **RFC 9110**, 섹션 9.3.8
`PATCH`	`PATCH`	HTTP/1.1 **RFC 5789**

22.10 http.client — HTTP 프로토콜 클라이언트

소스 코드: Lib/http/client.py

이 모듈은 HTTP 및 HTTPS 프로토콜의 클라이언트 측을 구현하는 클래스를 정의합니다. 일반적으로 직접 사용되지 않습니다 — `urllib.request` 모듈은 이를 사용하여 HTTP와 HTTPS를 사용하는 URL을 처리합니다.

> **더 보기**
>
> 더 고수준 HTTP 클라이언트 인터페이스로 Requests 패키지를 권장합니다.

> **참고**
>
> HTTPS 지원은 파이썬이 SSL 지원으로 컴파일된 경우에만 사용 가능합니다 (`ssl` 모듈을 통해).

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

이 모듈은 다음과 같은 클래스를 제공합니다:

class `http.client.`**`HTTPConnection`**(*host*, *port=None*, [*timeout*,]*source_address=None*, *blocksize=8192*)

`HTTPConnection` 인스턴스는 HTTP 서버와의 하나의 트랜잭션을 나타냅니다. 호스트와 선택적 포트 번호를 전달하여 인스턴스화해야 합니다. 포트 번호가 전달되지 않으면, host 문자열이 `host:port` 형식이면 여기에서 포트가 추출됩니다, 그렇지 않으면 기본 HTTP 포트(80)가 사용됩니다. 선택적 *timeout* 매개 변수가 제공되면, 블로킹 연산(연결 시도와 같은)이 지정된 초 후에 시간제한으로 종료됩니다 (제공되지 않으면, 전역 기본 시간제한 설정이 사용됩니다). 선택적 *source_address* 매개 변수는 HTTP 연결의 소스 주소로 사용할 (호스트, 포트)의 튜플일 수 있습니다. 선택적 *blocksize* 매개 변수는 파일류 메시지 바디를 보내는 데 필요한 버퍼 크기를 바이트 단위로 설정합니다.

예를 들어, 다음 호출은 모두 같은 호스트와 포트에 있는 서버에 연결하는 인스턴스를 만듭니다:

```
>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80, timeout=10)
```

버전 3.2에서 변경: *source_address*가 추가되었습니다.

버전 3.4에서 변경: *strict* 매개 변수가 제거되었습니다. HTTP 0.9 스타일 "간단한 응답"은 더는 지원되지 않습니다.

버전 3.7에서 변경: *blocksize* 매개 변수가 추가되었습니다.

class `http.client.`**`HTTPSConnection`**(*host*, *port=None*, *, [*timeout*,]*source_address=None*, *context=None*, *blocksize=8192*)

보안 서버와의 통신에 SSL을 사용하는 `HTTPConnection`의 서브 클래스. 기본 포트는 `443`입니다. *context*가 지정되면, 다양한 SSL 옵션을 기술하는 `ssl.SSLContext` 인스턴스여야 합니다.

모범 사례에 대한 자세한 내용은 보안 고려 사항을 참조하십시오.

버전 3.2에서 변경: *source_address*, *context* 및 *check_hostname*이 추가되었습니다.

버전 3.2에서 변경: 이 클래스는 이제 가능하면 (즉, `ssl.HAS_SNI`가 참이면) HTTPS 가상 호스트를 지원합니다.

버전 3.4에서 변경: *strict* 매개 변수가 제거되었습니다. HTTP 0.9 스타일 "간단한 응답"은 더는 지원되지 않습니다.

버전 3.4.3에서 변경: 이 클래스는 이제 기본적으로 필요한 모든 인증서와 호스트 이름 검사를 수행합니다. 이전의 검사하지 않는 동작으로 되돌리려면 `ssl._create_unverified_context()` 를 *context* 매개 변수로 전달할 수 있습니다.

버전 3.8에서 변경: 이 클래스는 이제 기본 *context*나 *cert_file*이 사용자 정의 *context*와 함께 전달될 때 TLS 1.3 `ssl.SSLContext.post_handshake_auth`를 활성화합니다.

버전 3.10에서 변경: This class now sends an ALPN extension with protocol indicator `http/1.1` when no *context* is given. Custom *context* should set ALPN protocols with `set_alpn_protocols()`.

버전 3.12에서 변경: 폐지된 *key_file*, *cert_file* 및 *check_hostname* 매개 변수를 제거했습니다.

class `http.client.`**`HTTPResponse`**(*sock*, *debuglevel=0*, *method=None*, *url=None*)

성공적으로 연결되면 반환되는 인스턴스의 클래스. 사용자가 직접 인스턴스화 하지 않습니다.

버전 3.4에서 변경: *strict* 매개 변수가 제거되었습니다. HTTP 0.9 스타일 "간단한 응답" 은 더는 지원되지 않습니다.

이 모듈은 다음 함수를 제공합니다:

`http.client.`**`parse_headers`**(*fp*)

HTTP 요청/응답을 나타내는 파일 포인터 *fp*에서 헤더를 구문 분석합니다. 파일은 `BufferedIOBase` 판독기(reader)여야 하며 (즉 텍스트가 아닙니다) 유효한 **RFC 2822** 스타일 헤더를 제공해야 합니다.

이 함수는 헤더 필드를 담은 `http.client.HTTPMessage` 인스턴스를 반환하지만, 페이 로드는 반환하지 않습니다 (`HTTPResponse.msg`와 `http.server.BaseHTTPRequestHandler.headers`와 같습니다). 반환 후, 파일 포인터 *fp*는 HTTP 바디를 읽을 준비가 되었습니다.

> **참고**
>
> `parse_headers()`는 HTTP 메시지의 시작 줄을 구문 분석하지 않습니다; `Name: value` 줄만 구문 분석합니다. 파일은 이러한 필드 줄을 읽을 준비가 되어 있어야 해서, 함수를 호출하기 전에 첫 번째 줄이 이미 소비되었어야 합니다.

다음과 같은 예외가 적절하게 발생합니다:

exception `http.client.`**`HTTPException`**

이 모듈에 있는 다른 예외의 베이스 클래스입니다. `Exception`의 서브 클래스입니다.

exception `http.client.`**`NotConnected`**

`HTTPException`의 서브 클래스.

exception `http.client.`**`InvalidURL`**

포트가 제공되고 숫자가 아니거나 비어있을 때 발생하는 `HTTPException`의 서브 클래스.

exception `http.client.`**`UnknownProtocol`**

`HTTPException`의 서브 클래스.

exception `http.client.`**`UnknownTransferEncoding`**

`HTTPException`의 서브 클래스.

exception `http.client.`**`UnimplementedFileMode`**

`HTTPException`의 서브 클래스.

exception `http.client.`**`IncompleteRead`**

`HTTPException`의 서브 클래스.

exception `http.client.`**`ImproperConnectionState`**

`HTTPException`의 서브 클래스.

exception `http.client.`**`CannotSendRequest`**

`ImproperConnectionState`의 서브 클래스.

exception http.client.**CannotSendHeader**

*ImproperConnectionState*의 서브 클래스.

exception http.client.**ResponseNotReady**

*ImproperConnectionState*의 서브 클래스.

exception http.client.**BadStatusLine**

*HTTPException*의 서브 클래스. 이해하지 못하는 HTTP 상태 코드로 서버가 응답하면 발생합니다.

exception http.client.**LineTooLong**

*HTTPException*의 서브 클래스. 서버에서 HTTP 프로토콜로 너무 긴 줄이 수신되면 발생합니다.

exception http.client.**RemoteDisconnected**

*ConnectionResetError*와 *BadStatusLine*의 서브 클래스. *HTTPConnection.getresponse()*가 응답을 읽으려고 시도할 때 연결에서 아무런 데이터를 읽지 못하여, 원격 끝이 연결을 닫았음을 표시하면 발생합니다.

Added in version 3.5: 이전에는, *BadStatusLine*('')가 발생했습니다.

이 모듈에 정의된 상수는 다음과 같습니다:

http.client.**HTTP_PORT**

HTTP 프로토콜의 기본 포트 (항상 80).

http.client.**HTTPS_PORT**

HTTPS 프로토콜의 기본 포트 (항상 443).

http.client.**responses**

이 딕셔너리는 HTTP 1.1 상태 코드를 W3C 이름으로 매핑합니다.

예: http.client.responses[http.client.NOT_FOUND]는 'Not Found'입니다.

이 모듈에서 상수로 사용 가능한 HTTP 상태 코드 목록은 *HTTP 상태 코드*를 참조하십시오.

22.10.1 HTTPConnection 객체

HTTPConnection 인스턴스에는 다음과 같은 메서드가 있습니다:

HTTPConnection.**request**(*method*, *url*, *body=None*, *headers={}*, **, encode_chunked=False*)

HTTP 요청 메서드 *method*와 요청 URI *url*을 사용하여 서버로 요청을 보냅니다. 제공된 *url*은 (HTTP 프록시 서버에 연결하거나 OPTIONS나 CONNECT 메서드를 사용하지 않는 한) **RFC 2616 §5.1.2**를 준수하는 절대 경로여야 합니다.

*body*가 지정되면, 헤더가 완료된 후 지정된 데이터가 전송됩니다. *str*, 바이트열류 객체, 열린 파일 객체 또는 *bytes*의 이터러블일 수 있습니다. *body*가 문자열이면, HTTP의 기본값인 ISO-8859-1로 인코딩됩니다. 바이트열류 객체이면, 바이트열은 그대로 전송됩니다. 파일 객체이면, 파일의 내용이 전송됩니다; 이 파일 객체는 최소한 read() 메서드를 지원해야 합니다. 파일 객체가 *io.TextIOBase*의 인스턴스이면, read() 메서드에서 반환된 데이터는 ISO-8859-1로 인코딩되고, 그렇지 않으면 read()에서 반환된 데이터가 그대로 전송됩니다. *body*가 이터러블이면, 이터러블이 소진될 때까지 이터러블의 요소가 그대로 전송됩니다.

headers 인자는 요청과 함께 보낼 추가 HTTP 헤더의 매핑이어야 합니다. (HTTP 프록시 서버에 연결하거나 OPTIONS나 CONNECT 메서드를 사용하지 않는 한) **Host 헤더**는 **RFC 2616 §5.1.2**를 준수하도록 제공되어야 합니다.

*headers*에 Content-Length도 Transfer-Encoding도 없지만, 요청 바디가 있으면, 이 헤더 필드 중 하나가 자동으로 추가됩니다. *body*가 None이면, 바디를 기대하는 메서드(PUT, POST 및 PATCH)의 경우 Content-Length 헤더가 0으로 설정됩니다. *body*가 문자열이거나 파일이 아닌 바이트열류 객체이면, Content-Length 헤더가 그것의 길이로 설정됩니다. 다른 모든 형의 *body*(파일과 이터러블 일반)는 청크 인코딩되며 Content-Length 대신 Transfer-Encoding 헤더가 자동으로 설정됩니다.

encode_chunked 인자는 Transfer-Encoding이 *headers*에 지정된 경우에만 유효합니다. *encode_chunked*가 False이면, HTTPConnection 객체는 모든 인코딩이 호출하는 코드에서 처리된다고 가정합니다. True이면, 바디가 청크 인코딩됩니다.

For example, to perform a GET request to https://docs.python.org/3/:

```
>>> import http.client
>>> host = "docs.python.org"
>>> conn = http.client.HTTPSConnection(host)
>>> conn.request("GET", "/3/", headers={"Host": host})
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200 OK
```

> **참고**
>
> 청크 전송 인코딩은 HTTP 프로토콜 버전 1.1에 추가되었습니다. HTTP 서버가 HTTP 1.1을 처리하는 것으로 알려지지 않은 한, 호출자는 Content-Length를 지정하거나, 바디 표현으로 *str*이나 파일이 아닌 바이트열류 객체를 전달해야 합니다.

버전 3.2에서 변경: *body*는 이제 이터러블일 수 있습니다.

버전 3.6에서 변경: Content-Length도 Transfer-Encoding도 *headers*에 설정되지 않으면, 파일과 이터러블 *body* 객체는 이제 청크 인코딩됩니다. *encode_chunked* 인자가 추가되었습니다. 파일 객체의 Content-Length를 결정하려는 시도는 없습니다.

HTTPConnection.**getresponse**()

요청을 보낸 후에 서버에서 응답을 받기 위해 호출해야 합니다. *HTTPResponse* 인스턴스를 반환합니다.

> **참고**
>
> 서버에 새 요청을 보내기 전에 전체 응답을 읽어야 함에 유의하십시오.

버전 3.5에서 변경: *ConnectionError*나 서브 클래스가 발생하면, 새 요청이 전송될 때 *HTTPConnection* 객체는 다시 연결할 준비가 됩니다.

HTTPConnection.**set_debuglevel**(*level*)

디버깅 수준을 설정합니다. 기본 디버그 수준은 0이며, 디버깅 출력이 인쇄되지 않음을 의미합니다. 0보다 큰 값을 지정하면 현재 정의된 모든 디버그 출력이 표준 출력으로 인쇄됩니다. debuglevel은 만들어지는 모든 새로운 *HTTPResponse* 객체로 전달됩니다.

Added in version 3.1.

HTTPConnection.**set_tunnel**(*host, port=None, headers=None*)

HTTP Connect 터널링(tunnelling)을 위한 host와 및 port를 설정합니다. 프락시 서버를 통해 연결을 실행할 수 있도록 합니다.

*host*와 *port* 인자는 터널링 된 연결의 말단(즉 CONNECT 요청에 포함되는 주소, 프락시 서버의 주소가 아닙니다)을 지정합니다.

headers 인자는 CONNECT 요청과 함께 보낼 추가 HTTP 헤더의 매핑이어야 합니다.

As HTTP/1.1 is used for HTTP CONNECT tunnelling request, as per the RFC, a HTTP Host: header must be provided, matching the authority-form of the request target provided as the destination for the CONNECT request. If a HTTP Host: header is not provided via the headers argument, one is generated and transmitted automatically.

예를 들어, 포트 8080에서 로컬로 실행되는 HTTPS 프락시 서버를 통해 터널링 하려면, 프락시 주소를 *HTTPSConnection* 생성자에 전달하고, 최종적으로 *set_tunnel()* 메서드에 도달하려는 호스트 주소를 전달합니다:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

Added in version 3.2.

버전 3.12에서 변경: HTTP CONNECT tunnelling requests use protocol HTTP/1.1, upgraded from protocol HTTP/1.0. `Host:` HTTP headers are mandatory for HTTP/1.1, so one will be automatically generated and transmitted if not provided in the headers argument.

`HTTPConnection.`**`get_proxy_response_headers`**`()`

Returns a dictionary with the headers of the response received from the proxy server to the CONNECT request.

If the CONNECT request was not sent, the method returns `None`.

Added in version 3.12.

`HTTPConnection.`**`connect`**`()`

객체가 만들어질 때 지정된 서버에 연결합니다. 기본적으로, 클라이언트가 이미 연결되지 않았다면 요청 시 자동으로 호출됩니다.

Raises an *auditing event* `http.client.connect` with arguments `self`, `host`, `port`.

`HTTPConnection.`**`close`**`()`

서버로의 연결을 닫습니다.

`HTTPConnection.`**`blocksize`**

파일류 메시지 바디를 보내기 위한 바이트 단위의 버퍼 크기.

Added in version 3.7.

위에서 설명한 `request()` 메서드를 사용하는 대신, 아래 네 가지 함수를 사용하여 단계별로 요청을 보낼 수도 있습니다.

`HTTPConnection.`**`putrequest`**`(`*method, url, skip_host=False, skip_accept_encoding=False*`)`

서버에 연결한 후 첫 번째 호출이어야 합니다. *method* 문자열, *url* 문자열 및 HTTP 버전(`HTTP/1.1`)으로 구성된 줄을 서버로 보냅니다. `Host:` 나 `Accept-Encoding:` 헤더의 자동 전송을 비활성화하려면 (예를 들어 추가 콘텐츠 인코딩을 허용하려면), False가 아닌 값으로 *skip_host* 나 *skip_accept_encoding* 을 지정하십시오.

`HTTPConnection.`**`putheader`**`(`*header, argument*[, ...]`)`

RFC 822 스타일 헤더를 서버에 보냅니다. header, 콜론과 공백 및 첫 번째 인자로 구성된 줄을 서버로 보냅니다. 더 많은 인자가 제공되면, 탭과 인자로 구성된 연속 줄(continuation lines)이 전송됩니다.

`HTTPConnection.`**`endheaders`**`(`*message_body=None, *, encode_chunked=False*`)`

헤더의 끝을 알리는 빈 줄을 서버에 보냅니다. 선택적 *message_body* 인자를 사용하여 요청과 연관된 메시지 바디를 전달할 수 있습니다.

encode_chunked 가 `True` 이면, *message_body* 의 각 이터레이션 결과는 **RFC 7230**, 섹션 3.3.1에 지정된 대로 청크 인코딩됩니다. 데이터의 인코딩 방식은 *message_body* 의 형에 따라 다릅니다. *message_body* 가 버퍼 인터페이스를 구현하면, 인코딩은 단일 청크를 만듭니다. *message_body* 가 `collections.abc.Iterable` 이면, *message_body* 의 각 이터레이션이 청크가 됩니다. *message_body* 가 파일 객체이면, 각 `.read()` 호출마다 청크가 됩니다. 이 메서드는 *message_body* 직후에 청크 인코딩된 데이터의 끝을 자동으로 알립니다.

> **참고**
>
> 청크 인코딩 명세로 인해, 이터레이터 바디에서 산출된 빈 청크는 청크 인코더에서 무시됩니다. 이는 형식이 잘못된 인코딩으로 인해 대상 서버의 요청 읽기가 조기에 종료되지 않도록 하려는 것입니다.

버전 3.6에서 변경: 청크 인코딩 지원과 *encode_chunked* 매개 변수가 추가되었습니다.

`HTTPConnection.send(data)`

서버로 data를 보냅니다. 이것은 `endheaders()` 메서드가 호출된 후, 그리고 `getresponse()`가 호출되기 전에만 직접 사용해야 합니다.

Raises an *auditing event* `http.client.send` with arguments `self`, `data`.

22.10.2 HTTPResponse 객체

`HTTPResponse` 인스턴스는 서버의 HTTP 응답을 감쌉니다. 요청 헤더와 엔티티 바디에 대한 액세스를 제공합니다. 응답은 이터러블 객체이며 with 문에서 사용할 수 있습니다.

버전 3.5에서 변경: `io.BufferedIOBase` 인터페이스가 이제 구현되었으며 이것의 모든 판독기(reader) 연산이 지원됩니다.

`HTTPResponse.read([amt])`

응답 바디나 다음 최대 *amt* 바이트를 읽고 반환합니다.

`HTTPResponse.readinto(b)`

응답 바디의 다음 최대 len(b) 바이트를 버퍼 *b*로 읽습니다. 읽은 바이트 수를 반환합니다.

Added in version 3.3.

`HTTPResponse.getheader(name, default=None)`

헤더 *name*의 값을 반환하거나, *name*과 일치하는 헤더가 없으면 *default*를 반환합니다. 이름이 *name*인 헤더가 둘 이상 있으면, ',' 로 연결한 모든 값을 반환합니다. *default*가 단일 문자열 이외의 이터러블이면, 해당 요소들도 마찬가지로 쉼표로 연결되어 반환됩니다.

`HTTPResponse.getheaders()`

(헤더, 값) 튜플의 리스트를 반환합니다.

`HTTPResponse.fileno()`

하부 소켓의 `fileno`를 반환합니다.

`HTTPResponse.msg`

응답 헤더를 포함하는 `http.client.HTTPMessage` 인스턴스. `http.client.HTTPMessage`는 `email.message.Message`의 서브 클래스입니다.

`HTTPResponse.version`

서버가 사용하는 HTTP 프로토콜 버전. HTTP/1.0의 경우 10, HTTP/1.1의 경우 11.

`HTTPResponse.url`

가져온 자원의 URL, 일반적으로 리디렉션을 따라갔는지 판별하는 데 사용됩니다.

`HTTPResponse.headers`

`email.message.EmailMessage` 인스턴스 형식의 응답 헤더.

`HTTPResponse.status`

서버가 반환한 상태 코드.

`HTTPResponse.reason`

서버가 반환한 이유 문구.

`HTTPResponse.debuglevel`

디버깅 훅. `debuglevel`이 0보다 크면, 응답을 읽고 구문 분석할 때 메시지가 표준 출력으로 인쇄됩니다.

`HTTPResponse.closed`

스트림이 닫혔으면 `True`입니다.

`HTTPResponse.geturl()`

버전 3.9부터 폐지됨: 폐지되었고 `url`로 대체되었습니다.

HTTPResponse.**info**()

버전 3.9부터 폐지됨: 폐지되었고 *headers*로 대체되었습니다.

HTTPResponse.**getcode**()

버전 3.9부터 폐지됨: 폐지되었고 *status*로 대체되었습니다.

22.10.3 예

GET 메서드를 사용하는 예제 세션은 다음과 같습니다:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # 전체 내용을 반환합니다.
>>> # 다음 예제는 청크로 데이터를 읽는 방법을 보여줍니다.
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while chunk := r1.read(200):
...     print(repr(chunk))
b'<!doctype html>\n<!--[if"...
...
>>> # 유효하지 않은 요청의 예
>>> conn = http.client.HTTPSConnection("docs.python.org")
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

다음은 HEAD 메서드를 사용하는 예제 세션입니다. HEAD 메서드는 어떤 데이터도 반환하지 않음에 유의하십시오.

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
>>> print(len(data))
0
>>> data == b''
True
```

POST 메서드를 사용하는 예제 세션은 다음과 같습니다:

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issue', '@action
↪': 'show'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="https://bugs.python.org/issue12524">https://bugs.python.
↪org/issue12524</a>'
>>> conn.close()
```

클라이언트 측 HTTP PUT 요청은 POST 요청과 매우 유사합니다. 차이점은 HTTP 서버가 PUT 요청을 통해 리소스를 만들도록 허락하는 서버 측에만 있습니다. 사용자 정의 HTTP 메서드는 적절한 method 어트리뷰트를 설정함으로써 *urllib.request.Request*에서도 처리된다는 점에 주목해야 합니다. 다음은 PUT 메서드를 사용하는 예제 세션입니다:

```
>>> # http://localhost:8080/file 리소스의 동봉된 표현으로
>>> # BODY의 내용이 포함된
>>> # HTTP 메시지를 만듭니다
...
>>> import http.client
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
```

22.10.4 HTTPMessage 객체

class http.client.**HTTPMessage**(*email.message.Message*)

http.client.HTTPMessage 인스턴스는 HTTP 응답의 헤더를 보유합니다. *email.message.Message* 클래스를 사용하여 구현됩니다.

22.11 ftplib — FTP 프로토콜 클라이언트

소스 코드: Lib/ftplib.py

이 모듈은 *FTP* 클래스와 몇 가지 관련 항목을 정의합니다. *FTP* 클래스는 FTP 프로토콜의 클라이언트 쪽을 구현합니다. 이것을 사용하여 다른 FTP 서버 미러링과 같은 다양한 자동화된 FTP 작업을 수행하는 파이썬 프로그램을 작성할 수 있습니다. 또한 *urllib.request* 모듈에서 FTP를 사용하는 URL을 처리하는 데 사용됩니다. FTP(File Transfer Protocol)에 대한 자세한 내용은 인터넷 **RFC 959**를 참조하십시오.

RFC 2640에 따라, 기본 인코딩은 UTF-8입니다.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

ftplib 모듈을 사용한 샘플 세션은 다음과 같습니다:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.us.debian.org')  # 기본 포트로 호스트에 연결합니다
>>> ftp.login()                     # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')               # "debian" 디렉터리로 변경합니다
'250 Directory successfully changed.'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> ftp.retrlines('LIST')           # 디렉터리 내용의 목록을 얻습니다
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> with open('README', 'wb') as fp:
>>>     ftp.retrbinary('RETR README', fp.write)
'226 Transfer complete.'
>>> ftp.quit()
'221 Goodbye.'
```

22.11.1 레퍼런스

FTP 객체

class ftplib.**FTP**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *timeout=None*, *source_address=None*, **, encoding='utf-8'*)

FTP 클래스의 새 인스턴스를 반환합니다.

매개변수

- **host** (str) – 연결할 호스트명. 주어지면, 생성자가 `connect(host)` 를 묵시적으로 호출합니다.
- **user** (str) – The username to log in with (default: `'anonymous'`). 주어지면, 생성자가 `login(host, passwd, acct)` 를 묵시적으로 호출합니다.
- **passwd** (str) – The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (str) – Account information to be used for the `ACCT` FTP command. Few systems implement this. See RFC-959 for more details.
- **timeout** (float | *None*) – *connect()* 와 같은 블로킹 연산에 대한 시간제한을 초로 지정 (기본값: 전역 기본 시간제한 설정).
- **source_address** (tuple | *None*) – A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.
- **encoding** (str) – The encoding for directories and filenames (default: `'utf-8'`).

FTP 클래스는 `with` 문을 지원합니다, 예를 들어:

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
>>>
```

버전 3.2에서 변경: `with` 문에 대한 지원이 추가되었습니다.

버전 3.3에서 변경: *source_address* 매개 변수가 추가되었습니다.

버전 3.9에서 변경: *timeout* 매개 변수가 0으로 설정되면, 비 블로킹 소켓이 만들어지지 않도록 `ValueError`를 발생시킵니다. *encoding* 매개 변수가 추가되었으며, 기본값은 Latin-1 에서 UTF-8 로 변경되어 **RFC 2640**을 따릅니다.

여러 `FTP` 메서드가 두 가지 스타일로 제공됩니다: 텍스트 파일을 처리하는 것과 바이너리 파일을 위한 것. 명령 뒤에 텍스트 버전의 경우 `lines`, 바이너리 버전의 경우 `binary`를 붙여서 이 메서드들의 이름을 지정했습니다.

`FTP` 인스턴스에는 다음과 같은 메서드가 있습니다:

set_debuglevel(*level*)

인스턴스의 디버깅 수준을 `int`로 설정합니다. 인쇄되는 디버깅 출력의 양을 제어합니다. 디버깅 수준은 다음과 같습니다:

- `0` (기본값): 디버깅 출력을 생성하지 않습니다.
- `1`: 적당한 양의 디버깅 출력, 일반적으로 요청당 한 줄을 생성합니다.
- `2` 이상: 제어 연결에서 보내고 받는 각 줄을 로깅하여 최대량의 디버깅 출력을 생성합니다.

connect(*host=''*, *port=0*, *timeout=None*, *source_address=None*)

주어진 host와 port에 연결합니다. 이 함수는 각 인스턴스에 대해 한 번만 호출해야 합니다; `FTP` 인스턴스를 만들 때 *host* 인자가 제공되었으면, 호출되지 않아야 합니다. 다른 모든 `FTP` 메서드는 연결이 성공적으로 이루어진 후에만 호출할 수 있습니다.

매개변수

- **host** (`str`) – 연결할 호스트.
- **port** (`int`) – 연결할 TCP 포트 (기본값: `21`, FTP 프로토콜 명세에 지정된 대로). 다른 포트 번호를 지정할 필요는 거의 없습니다.
- **timeout** (`float` | `None`) – 연결 시도에 대한 시간제한 (기본값: 전역 기본 시간 제한 설정).
- **source_address** (`tuple` | `None`) – A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.

인자 `self`, `host`, `port`로 감사 이벤트 `ftplib.connect`를 발생시킵니다.

버전 3.3에서 변경: *source_address* 매개 변수가 추가되었습니다.

getwelcome()

초기 연결에 대한 응답으로 서버에서 보낸 환영 메시지를 반환합니다. (이 메시지에는 때때로 사용자와 관련될 수 있는 고지 사항이나 도움말 정보가 포함되어 있습니다.)

login(*user='anonymous'*, *passwd=''*, *acct=''*)

연결된 FTP 서버로 로그인합니다. 이 함수는 연결이 설정된 후 각 인스턴스에 마다 한 번만 호출해야 합니다; `FTP` 인스턴스가 만들어질 때 *host*와 *user* 인자가 제공되었으면 호출되지 않아야 합니다. 대부분의 FTP 명령은 클라이언트가 로그인한 후에만 허용됩니다.

매개변수

- **user** (`str`) – The username to log in with (default: `'anonymous'`).
- **passwd** (`str`) – The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (`str`) – Account information to be used for the `ACCT` FTP command. Few systems implement this. See RFC-959 for more details.

abort()

진행 중인 파일 전송을 중단합니다. 이것을 사용하는 것이 항상 동작하지는 않지만, 시도해 볼 가치가 있습니다.

sendcmd(*cmd*)

간단한 명령 문자열을 서버로 보내고 응답 문자열을 반환합니다.

인자 `self`, `cmd`로 감사 이벤트 `ftplib.sendcmd`를 발생시킵니다.

voidcmd(*cmd*)

간단한 명령 문자열을 서버로 보내고 응답을 처리합니다. 응답 코드가 성공(200–299 범위의 코드)에 해당하면 응답 문자열을 반환합니다. 그렇지 않으면 `error_reply`를 발생시킵니다.

인자 `self`, `cmd`로 감사 이벤트 `ftplib.sendcmd`를 발생시킵니다.

retrbinary(*cmd, callback, blocksize=8192, rest=None*)

바이너리 전송 모드로 파일을 가져옵니다.

매개변수

- **cmd** (`str`) – 적절한 `RETR` 명령: `"RETR filename"`.
- **callback** (*callable*) – 수신된 각 데이터 블록에 대해 호출되는 단일 매개 변수 콜러블로, 단일 인자는 `bytes` 형의 데이터입니다.
- **blocksize** (`int`) – 실제 전송을 수행하기 위해 만들어진 저수준 `socket` 객체에서 읽을 최대 청크 크기. *callback*에 전달되는 데이터의 최대 크기에 해당하기도 합니다. 기본값은 `8192`.
- **rest** (`int`) – 서버로 전송할 `REST` 명령. `transfercmd()` 메서드의 *rest* 매개 변수에 대한 설명서를 참조하세요.

retrlines(*cmd, callback=None*)

초기화 때 *encoding* 매개 변수로 지정된 인코딩으로 파일이나 디렉터리 목록을 가져옵니다. *cmd*는 적절한 `RETR` 명령(`retrbinary()`를 참조하십시오)이거나 `LIST`나 `NLST`와 같은 명령(보통 단지 문자열 `'LIST'`)이어야 합니다. `LIST`는 파일 목록과 해당 파일에 대한 정보를 가져옵니다. `NLST`는 파일 이름 목록을 가져옵니다. *callback* 함수는 각 줄에 대해 호출되며, 후행 CRLF가 제거된 줄을 포함하는 문자열을 인자로 제공합니다. 기본 *callback*은 줄을 `sys.stdout`으로 인쇄합니다.

set_pasv(*val*)

*val*이 참이면 "수동(passive)" 모드를 활성화하고, 그렇지 않으면 수동 모드를 비활성화합니다. 수동 모드는 기본적으로 켜져 있습니다.

storbinary(*cmd, fp, blocksize=8192, callback=None, rest=None*)

바이너리 전송 모드로 파일을 저장합니다.

매개변수

- **cmd** (`str`) – 적절한 `STOR` 명령: `"STOR filename"`.
- **fp** (*file object*) – (바이너리 모드로 열린) 파일 객체이며 저장될 데이터를 제공하기 위해 *blocksize* 크기의 블록으로 `read()` 메서드를 사용하여 EOF까지 읽힙니다.
- **blocksize** (`int`) – 읽기 블록 크기. 기본값은 `8192`.
- **callback** (*callable*) – 전송된 각 데이터 블록에 대해 호출되는 단일 매개 변수 콜러블로, 단일 인자는 `bytes` 형의 데이터입니다.
- **rest** (`int`) – 서버로 전송할 `REST` 명령. `transfercmd()` 메서드의 *rest* 매개 변수에 대한 설명서를 참조하세요.

버전 3.2에서 변경: *rest* 매개 변수가 추가되었습니다.

storlines(*cmd, fp, callback=None*)

줄 모드로 파일을 저장합니다. *cmd*는 적절한 `STOR` 명령이어야 합니다(`storbinary()`를 참조하십시오). 저장될 데이터를 제공하기 위해 `readline()` 메서드를 사용하여 (바이너리 모드로 열린) 파일 객체 *fp*에서 EOF까지 줄을 읽어 들입니다. *callback*은 줄마다 보내진 다음에 호출되는 선택적 단일 매개 변수 콜러블입니다.

transfercmd(*cmd*, *rest=None*)

데이터 연결을 통해 전송을 시작합니다. 전송이 활성화되면, EPRT 나 PORT 명령과 *cmd* 로 지정한 전송 명령을 보내고 연결을 받아들입니다. 서버가 수동(passive)이면, EPSV 나 PASV 명령을 전송하고, 서버에 연결한 다음 전송 명령을 시작합니다. 어느 쪽이든, 연결 소켓을 반환합니다.

선택적 *rest* 가 제공되면, REST 명령이 서버로 전송되고 *rest* 를 인자로 전달합니다. *rest* 는 일반적으로 요청된 파일에 대한 바이트 오프셋으로, 요청된 오프셋에서 파일 바이트 전송을 다시 시작하고 초기 바이트를 건너뛰도록 서버에 지시합니다. 그러나 `transfercmd()` 메서드는 초기화 때 지정된 *encoding* 매개 변수를 사용하여 *rest* 를 문자열로 변환하지만, 문자열의 내용은 점검하지 않음에 유의하십시오. 서버가 REST 명령을 인식하지 못하면, `error_reply` 예외가 발생합니다. 이 경우, 단순히 *rest* 인자 없이 `transfercmd()` 를 호출하십시오.

ntransfercmd(*cmd*, *rest=None*)

`transfercmd()` 와 비슷하지만, 데이터 연결과 데이터의 예상 크기의 튜플을 반환합니다. 예상 크기를 계산할 수 없으면, None 이 예상 크기로 반환됩니다. *cmd* 와 *rest* 는 `transfercmd()` 에서와 같은 의미입니다.

mlsd(*path=''*, *facts=[]*)

MLSD 명령(**RFC 3659**)을 사용하여 표준화된 형식으로 디렉터리를 나열합니다. *path* 가 생략되면 현재 디렉터리를 가정합니다. *facts* 는 원하는 정보 유형을 나타내는 문자열의 리스트입니다(예를 들어 `["type", "size", "perm"]`). 경로에서 발견된 모든 파일에 대해 두 요소의 튜플을 산출하는 제너레이터 객체를 반환합니다. 첫 번째 요소는 파일 이름이고, 두 번째 요소는 파일 이름에 대한 사실(facts)을 포함하는 딕셔너리입니다. 이 딕셔너리의 내용은 *facts* 인자에 의해 제한될 수 있지만, 서버가 요청된 모든 사실을 반환한다고 보장하지는 않습니다.

Added in version 3.3.

nlst(*argument*[, ...])

NLST 명령이 반환한 파일 이름 리스트를 반환합니다. 선택적 *argument* 는 나열할 디렉터리입니다(기본값은 현재 서버 디렉터리입니다). 비표준 옵션을 NLST 명령에 전달하기 위해 여러 인자를 사용할 수 있습니다.

> **참고**
>
> 서버가 명령을 지원한다면, `mlsd()` 가 더 나은 API를 제공합니다.

dir(*argument*[, ...])

LIST 명령이 반환한 디렉터리 목록을 생성하여 표준 출력으로 인쇄합니다. 선택적 *argument* 는 나열할 디렉터리입니다(기본값은 현재 서버 디렉터리입니다). 비표준 옵션을 LIST 명령에 전달하기 위해 여러 인자를 사용할 수 있습니다. 마지막 인자가 함수면, `retrlines()` 와 같이 *callback* 함수로 사용됩니다; 기본값은 `sys.stdout` 으로 인쇄합니다. 이 메서드는 None 을 반환합니다.

> **참고**
>
> 서버가 명령을 지원한다면, `mlsd()` 가 더 나은 API를 제공합니다.

rename(*fromname*, *toname*)

서버의 파일 *fromname* 을 *toname* 으로 이름을 바꿉니다.

delete(*filename*)

서버에서 *filename* 이라는 파일을 제거합니다. 성공하면, 응답 텍스트를 반환하고, 그렇지 않으면 권한 에러면 `error_perm` 을, 다른 에러면 `error_reply` 를 발생시킵니다.

cwd(*pathname*)

서버에서 현재 디렉터리를 설정합니다.

mkd(*pathname*)

서버에서 새 디렉터리를 만듭니다.

pwd()

서버에서 현재 디렉터리의 경로명을 반환합니다.

rmd(*dirname*)

서버에서 *dirname*이라는 디렉터리를 제거합니다.

size(*filename*)

서버에서 *filename*이라는 파일의 크기를 요청합니다. 성공하면, 파일 크기가 정수로 반환되고, 그렇지 않으면 `None`이 반환됩니다. `SIZE` 명령은 표준화되어 있지 않지만, 많은 일반적인 서버 구현에서 지원됨에 유의하십시오.

quit()

`QUIT` 명령을 서버로 보내고 연결을 닫습니다. 이는 연결을 닫는 "정중한" 방법이지만, 서버가 `QUIT` 명령에 대해 에러로 응답하면 예외가 발생할 수 있습니다. 이것은 묵시적인 `close()` 메서드 호출을 수반하며, 이는 후속 호출에 `FTP` 인스턴스를 쓸모없게 만듭니다 (아래를 참조하십시오).

close()

일방적으로 연결을 닫습니다. 이것은 이미 닫힌 연결에는 적용되지 않아야 합니다, 가령 `quit()`를 성공적으로 호출한 후에. 이 호출 후 `FTP` 인스턴스를 더는 사용하지 않아야 합니다 (`close()`나 `quit()` 호출 후 다른 `login()` 메서드를 사용해서 연결을 다시 열 수 없습니다).

FTP_TLS 객체

class ftplib.**FTP_TLS**(*host=''*, *user=''*, *passwd=''*, *acct=''*, **, context=None*, *timeout=None*, *source_address=None*, *encoding='utf-8'*)

RFC 4217에 설명된 대로 FTP에 TLS 지원을 추가하는 `FTP` 서브 클래스. 인증하기 전에 FTP 제어 연결을 묵시적으로 보안을 유지하면서 포트 21에 연결합니다.

> **참고**
>
> 사용자는 `prot_p()` 메서드를 호출하여 데이터 연결을 명시적으로 보호해야 합니다.

매개변수

- **host** (`str`) – 연결할 호스트명. 주어지면, 생성자가 `connect(host)`를 묵시적으로 호출합니다.
- **user** (`str`) – The username to log in with (default: `'anonymous'`). 주어지면, 생성자가 `login(host, passwd, acct)`를 묵시적으로 호출합니다.
- **passwd** (`str`) – The password to use when logging in. If not given, and if *passwd* is the empty string or `"-"`, a password will be automatically generated.
- **acct** (`str`) – Account information to be used for the `ACCT` FTP command. Few systems implement this. See RFC-959 for more details.
- **context** (`ssl.SSLContext`) – SSL 구성 옵션, 인증서 및 개인 키를 잠재적으로 오래 유지되는 단일 구조로 묶을 수 있는 SSL 컨텍스트 객체. 모범 사례를 보려면 보안 고려 사항을 읽으십시오.
- **timeout** (`float` | `None`) – `connect()`와 같은 블로킹 연산에 대한 시간제한을 초로 지정 (기본값: 전역 기본 시간제한 설정)
- **source_address** (`tuple` | `None`) – A 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.
- **encoding** (`str`) – The encoding for directories and filenames (default: `'utf-8'`).

Added in version 3.2.

버전 3.3에서 변경: *source_address* 매개 변수가 추가되었습니다.

버전 3.4에서 변경: 이 클래스는 이제 `ssl.SSLContext.check_hostname`과 서버 이름 표시(*Server Name Indication*)(`ssl.HAS_SNI`를 참조하십시오)으로 호스트명 확인을 지원합니다.

버전 3.9에서 변경: *timeout* 매개 변수가 0으로 설정되면, 비 블로킹 소켓이 만들어지지 않도록 `ValueError`를 발생시킵니다. *encoding* 매개 변수가 추가되었으며, 기본값은 Latin-1 에서 UTF-8 로 변경되어 **RFC 2640**을 따릅니다.

버전 3.12에서 변경: 폐지된 *keyfile*와 *certfile* 매개 변수를 제거했습니다.

`FTP_TLS` 클래스를 사용한 샘플 세션은 다음과 같습니다:

```
>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
'230 Anonymous user logged in'
>>> ftps.prot_p()
'200 Data protection level set to "private"'
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bsdcam', 'clockspeed', 'djbdns-
↪jedi', 'docs', 'eaccelerator-jedi', 'favicon.ico', 'francotone', 'fugu',
↪'ignore', 'libpuzzle', 'metalog', 'minidentd', 'misc', 'mysql-udf-global-
↪user-variables', 'php-jenkins-hash', 'php-skein-hash', 'php-webdav',
↪'phpaudit', 'phpbench', 'pincaster', 'ping', 'posto', 'pub', 'public',
↪'public_keys', 'pure-ftpd', 'qscan', 'qtc', 'sharedance', 'skycache', 'sound
↪', 'tmp', 'ucarp']
```

`FTP_TLS` 클래스는 `FTP`를 상속하여, 다음과 같은 추가 메서드와 어트리뷰트를 정의합니다:

ssl_version

사용할 SSL 버전 (기본값은 `ssl.PROTOCOL_SSLv23`입니다).

auth()

`ssl_version` 어트리뷰트에 지정된 내용에 따라, TLS나 SSL을 사용하여 보안 제어 연결을 설정합니다.

버전 3.4에서 변경: 이 메서드는 이제 `ssl.SSLContext.check_hostname`과 서버 이름 표시(*Server Name Indication*)로 호스트명 확인을 지원합니다 (`ssl.HAS_SNI`를 참조하십시오).

ccc()

제어 채널을 일반 텍스트로 되돌립니다. 고정 포트를 열지 않고 비보안 FTP로 NAT을 다루는 방법을 알고 있는 방화벽을 활용하는 데 유용할 수 있습니다.

Added in version 3.3.

prot_p()

보안 데이터 연결을 설정합니다.

prot_c()

일반 텍스트 데이터 연결을 설정합니다.

모듈 변수

exception ftplib.**error_reply**

서버에서 예기치 않은 응답이 수신될 때 발생하는 예외.

exception ftplib.**error_temp**

일시적 에러(400–499 범위의 응답 코드)를 나타내는 에러 코드가 수신될 때 발생하는 예외.

exception ftplib.**error_perm**

영구 에러(500–599 범위의 응답 코드)를 나타내는 에러 코드가 수신될 때 발생하는 예외.

exception ftplib.**error_proto**

서버로부터 FTP(File Transfer Protocol)의 응답 규격(즉, 1–5 범위의 숫자로 시작)에 맞지 않는 응답을 수신할 때 발생하는 예외.

ftplib.**all_errors**

FTP 인스턴스의 메서드가 (호출자로 인한 프로그래밍 에러가 아니라) FTP 연결 문제로 인해 발생시킬 수 있는 모든 예외의 집합 (튜플). 이 집합에는 위에 나열된 네 가지 예외뿐만 아니라 *OSError*와 *EOFError*가 포함됩니다.

더 보기

모듈 *netrc*

.netrc 파일 형식의 구문 분석기. .netrc 파일은 일반적으로 FTP 클라이언트가 사용자에게 프롬프트 하기 전에 사용자 인증 정보를 로드하는 데 사용됩니다.

22.12 poplib — POP3 프로토콜 클라이언트

소스 코드: Lib/poplib.py

이 모듈은 POP3 서버에 대한 연결을 캡슐화하고 **RFC 1939**에 정의된 대로 프로토콜을 구현하는 클래스 *POP3*를 정의합니다. *POP3* 클래스는 **RFC 1939**의 최소(minimal)와 선택적인(optional) 명령 집합을 모두 지원합니다. *POP3* 클래스는 이미 맺어진 연결에서 암호화된 통신을 활성화하기 위해 **RFC 2595**에서 도입된 STLS 명령도 지원합니다.

또한, 이 모듈은 SSL을 하부 프로토콜 계층으로 사용하는 POP3 서버에 연결하기 위한 지원을 제공하는 클래스 *POP3_SSL*을 제공합니다.

POP3는 광범위하게 지원되지만 노후화되었음에 유의하십시오. POP3 서버의 구현 품질은 매우 다양하며, 그중 너무 많은 것들이 형편없습니다. 여러분의 메일 서버가 IMAP을 지원한다면, IMAP 서버가 더 잘 구현되는 경향이 있으므로 *imaplib.IMAP4* 클래스를 사용하는 것이 좋습니다.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

poplib 모듈은 두 가지 클래스를 제공합니다:

class poplib.**POP3**(*host*, *port=POP3_PORT*[, *timeout*])

이 클래스는 실제 POP3 프로토콜을 구현합니다. 인스턴스가 초기화될 때 연결이 만들어집니다. *port*를 생략하면, 표준 POP3 포트(110)가 사용됩니다. 선택적 *timeout* 매개 변수는 연결 시도의 제한 시간을 초로 지정합니다 (지정하지 않으면, 전역 기본 제한 시간 설정이 사용됩니다).

self, host, port를 인자로 감사 이벤트(*auditing event*) poplib.connect를 발생시킵니다.

모든 명령은 self와 line을 인자로 감사 이벤트(*auditing event*) poplib.putline을 발생시킵니다. 여기서 line은 원격 호스트로 보낼 바이트열입니다.

버전 3.9에서 변경: *timeout* 매개 변수가 0으로 설정되면, 비 블로킹 소켓이 만들어지지 않도록 *ValueError*를 발생시킵니다.

class poplib.**POP3_SSL**(*host*, *port=POP3_SSL_PORT*, *, *timeout=None*, *context=None*)

SSL 암호화된 소켓을 통해 서버에 연결하는 *POP3*의 서브 클래스입니다. *port*가 지정되지 않으면, 표준 POP3-over-SSL 포트(995)가 사용됩니다. *timeout*은 *POP3* 생성자에서처럼 동작합니다. *context*는 선택적인 *ssl.SSLContext* 객체인데, SSL 구성 옵션, 인증서 및 개인 키를 단일 (잠재적으로 수명이 긴) 구조로 묶을 수 있도록 합니다. 모범 사례는 보안 고려 사항을 읽으십시오.

self, host, port를 인자로 감사 이벤트(*auditing event*) poplib.connect를 발생시킵니다.

모든 명령은 self와 line을 인자로 감사 이벤트(*auditing event*) poplib.putline을 발생시킵니다. 여기서 line은 원격 호스트로 보낼 바이트열입니다.

버전 3.2에서 변경: *context* 매개 변수가 추가되었습니다.

버전 3.4에서 변경: 클래스는 이제 `ssl.SSLContext.check_hostname`을 통한 호스트 이름 검사와 서버 이름 표시(*Server Name Indication*)를 지원합니다(`ssl.HAS_SNI`를 참조하십시오).

버전 3.9에서 변경: *timeout* 매개 변수가 0으로 설정되면, 비 블로킹 소켓이 만들어지지 않도록 `ValueError`를 발생시킵니다.

버전 3.12에서 변경: 폐지된 *keyfile* 과 *certfile* 매개 변수는 제거되었습니다.

한가지 예외가 `poplib` 모듈의 어트리뷰트로 정의됩니다:

exception poplib.**error_proto**

이 모듈로부터 비롯된 모든 에러에서 발생하는 예외 (`socket` 모듈에서 비롯된 에러는 잡지 않습니다). 예외의 이유는 문자열로 생성자에 전달됩니다.

더 보기

모듈 *imaplib*
표준 파이썬 IMAP 모듈.

Frequently Asked Questions About Fetchmail
fetchmail POP/IMAP 클라이언트에 대한 FAQ는 POP 프로토콜에 기반하는 응용 프로그램을 작성해야 할 때 유용할 수 있는 POP3 서버 다양성과 RFC 위반에 대한 정보를 수집합니다.

22.12.1 POP3 객체

모든 POP3 명령은 소문자로 바뀐 같은 이름의 메서드로 표현됩니다; 대부분 서버에서 보낸 응답 텍스트를 반환합니다.

`POP3` 인스턴스에는 다음과 같은 메서드가 있습니다:

POP3.**set_debuglevel**(*level*)

인스턴스의 디버깅 수준을 설정합니다. 이것은 인쇄되는 디버깅 출력의 양을 제어합니다. 기본값인 `0`은 디버깅 출력을 생성하지 않습니다. `1` 값은 적절한 양의 디버깅 출력을 생성하는데, 일반적으로 요청당 한 줄입니다. `2` 이상의 값은 제어 연결에서 보내고 받은 각 줄을 로깅 하여 최대량의 디버깅 출력을 생성합니다.

POP3.**getwelcome**()

POP3 서버가 보낸 인사말 문자열을 반환합니다.

POP3.**capa**()

RFC 2449에 지정된 대로 서버의 기능을 조회합니다. `{'name': ['param'...]}` 형식의 딕셔너리를 반환합니다.

Added in version 3.4.

POP3.**user**(*username*)

user 명령을 보냅니다, 응답은 암호가 필요함을 가리켜야 합니다.

POP3.**pass_**(*password*)

암호를 보냅니다, 응답에는 메시지 수와 우편함 크기가 포함됩니다. 참고: `quit()`가 호출될 때까지 서버의 우편함은 잠깁니다.

POP3.**apop**(*user*, *secret*)

POP3 서버에 로그인하기 위해 더 안전한 APOP 인증을 사용합니다.

POP3.**rpop**(*user*)

POP3 서버에 로그인하기 위해 RPOP 인증(유닉스 r-명령과 유사합니다)을 사용합니다.

POP3.**stat**()

우편함 상태를 가져옵니다. 결과는 2개의 정수의 튜플입니다: `(message count, mailbox size)`.

POP3.**list**([*which*])

메시지 목록을 요청합니다, 결과는 `(response, ['mesg_num octets', ...], octets)` 형식입니다. *which*가 설정되면, 목록에 표시할 메시지입니다.

POP3.**retr**(*which*)

전체 메시지 번호 *which*를 가져오고 읽었음을 알리는 플래그를 설정합니다. 결과는 `(response, ['line', ...], octets)` 형식입니다.

POP3.**dele**(*which*)

메시지 번호 *which*를 삭제로 표시합니다. 대부분 서버에서 삭제는 실제로 QUIT 때까지 수행되지 않습니다 (주된 예외는 Eudora QPOP 인데, 모든 연결 단절 시 계류 중인 삭제를 수행하여 의도적으로 RFC를 위반합니다).

POP3.**rset**()

우편함에 대한 모든 삭제 표시를 제거합니다.

POP3.**noop**()

아무것도 하지 않습니다. 연결 유지로 사용될 수 있습니다.

POP3.**quit**()

로그아웃: 변경 내용 커밋, 우편함 잠금 해제, 연결 끊기.

POP3.**top**(*which*, *howmuch*)

메시지 번호 *which*의 메시지 헤더와 메시지의 *howmuch* 개 줄을 가져옵니다. 결과는 `(response, ['line', ...], octets)` 형식입니다.

이 메서드가 사용하는 POP3 TOP 명령은, RETR 명령과 달리, 메시지의 읽었음을 알리는 플래그를 설정하지 않습니다; 불행히도, TOP은 RFC에서 부실하게 기술되어 있고 종종 유명하지 않은 서버에서 망가져 있습니다. 사용할 POP3 서버를 신뢰하기 전에 이 메서드를 수동으로 테스트하십시오.

POP3.**uidl**(*which=None*)

메시지 다이제스트 (고유 ID) 목록을 반환합니다. *which*가 지정되면, 결과에는 해당 메시지의 고유 ID가 `'response mesgnum uid` 형식으로 포함됩니다. 그렇지 않으면, 결과는 목록 `(response, ['mesgnum uid', ...], octets)` 입니다.

POP3.**utf8**()

UTF-8 모드로의 전환을 시도합니다. 성공하면 서버 응답을 반환하고, 그렇지 않으면 *error_proto*를 발생시킵니다. **RFC 6856**에서 정의되었습니다.

Added in version 3.5.

POP3.**stls**(*context=None*)

RFC 2595에 지정된 대로 활성 연결에서 TLS 세션을 시작합니다. 사용자 인증 전에만 허용됩니다.

context 매개 변수는 *ssl.SSLContext* 객체인데, SSL 구성 옵션, 인증서 및 개인 키를 단일 (잠재적으로 수명이 긴) 구조로 묶을 수 있도록 합니다. 모범 사례는 보안 고려 사항을 읽으십시오.

이 메서드는 *ssl.SSLContext.check_hostname*을 통한 호스트 이름 검사와 서버 이름 표시 (*Server Name Indication*)를 지원합니다 (*ssl.HAS_SNI*를 참조하십시오)

Added in version 3.4.

*POP3_SSL*의 인스턴스에는 추가 메서드가 없습니다. 이 서브 클래스의 인터페이스는 그 부모와 같습니다.

22.12.2 POP3 예제

다음은 우편함을 열고 모든 메시지를 가져와서 인쇄하는 (에러 검사 없는) 최소한의 예입니다:

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

모듈의 끝에는, 더욱 광범위한 사용 예제가 포함된 테스트 섹션이 있습니다.

22.13 imaplib — IMAP4 프로토콜 클라이언트

소스 코드: Lib/imaplib.py

이 모듈은 *IMAP4*, *IMAP4_SSL* 및 *IMAP4_stream*의 세 가지 클래스를 정의합니다. 이 클래스는 IMAP4 서버에 대한 연결을 캡슐화하고 **RFC 2060**에 정의된 대로 IMAP4rev1 클라이언트 프로토콜의 큰 부분 집합을 구현합니다. IMAP4 (**RFC 1730**) 서버와 하위 호환되지만, IMAP4에서는 `STATUS` 명령이 지원되지 않음에 유의하십시오.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

imaplib 모듈은 세 가지 클래스를 제공하며, *IMAP4*는 베이스 클래스입니다:

class imaplib.**IMAP4**(*host=''*, *port=IMAP4_PORT*, *timeout=None*)

이 클래스는 실제 IMAP4 프로토콜을 구현합니다. 연결이 만들어지고 인스턴스가 초기화될 때 프로토콜 버전(IMAP4 또는 IMAP4rev1)이 결정됩니다. *host*를 지정하지 않으면, `''`(로컬 호스트)가 사용됩니다. *port*를 생략하면, 표준 IMAP4 포트(143)가 사용됩니다. 선택적 *timeout* 매개 변수는 연결 시도에 대한 시간제한을 초로 지정합니다. timeout이 제공되지 않거나 `None`이면, 전역 기본 소켓 시간제한이 사용됩니다.

IMAP4 클래스는 `with` 문을 지원합니다. 이처럼 사용될 때, `with` 문을 빠져나갈 때 IMAP4 `LOGOUT` 명령이 자동으로 발행됩니다. 예를 들어:

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

버전 3.5에서 변경: `with` 문에 대한 지원이 추가되었습니다.

버전 3.9에서 변경: 선택적 *timeout* 매개 변수가 추가되었습니다.

세 가지 예외가 *IMAP4* 클래스의 어트리뷰트로 정의됩니다:

exception IMAP4.**error**

모든 에러에 대해 발생하는 예외. 예외의 이유는 문자열로 생성자에 전달됩니다.

exception IMAP4.**abort**

IMAP4 서버 에러는 이 예외를 발생시킵니다. 이것은 *IMAP4.error*의 서브 클래스입니다. 인스턴스를 닫고 새 인스턴스를 만들면 일반적으로 이 예외에서 복구할 수 있습니다.

exception IMAP4.**readonly**

쓰기 가능한 사서함(mailbox)의 상태가 서버에 의해 변경되면 이 예외가 발생합니다. 이것은 *IMAP4.error*의 서브 클래스입니다. 이제 일부 다른 클라이언트에는 쓰기 권한이 있으며, 쓰기 권한을 다시 얻으려면 사서함을 다시 열어야 합니다.

보안 연결을 위한 서브 클래스도 있습니다:

class imaplib.**IMAP4_SSL**(*host=''*, *port=IMAP4_SSL_PORT*, **, ssl_context=None*, *timeout=None*)

SSL 암호화 소켓을 통해 연결되는 *IMAP4*에서 파생된 서브 클래스입니다 (이 클래스를 사용하려면 SSL 지원과 함께 컴파일된 소켓 모듈이 필요합니다). *host*를 지정하지 않으면, `''`(로컬 호스트)가 사용됩니다. *port*를 생략하면, 표준 IMAP4-over-SSL 포트(993)가 사용됩니다. *ssl_context*는 SSL 구성 옵션, 인증서 및 개인 키를 단일 (잠재적으로 오래가는) 구조로 번들링 할 수 있는 *ssl.SSLContext* 객체입니다. 모범 사례는 보안 고려 사항을 읽으십시오.

선택적 *timeout* 매개 변수는 연결 시도에 대한 시간제한을 초로 지정합니다. timeout이 제공되지 않거나 `None`이면, 전역 기본 소켓 시간제한이 사용됩니다.

버전 3.3에서 변경: *ssl_context* 매개 변수가 추가되었습니다.

버전 3.4에서 변경: 이 클래스는 이제 *ssl.SSLContext.check_hostname*과 서버 이름 표시(*Server Name Indication*)로 호스트명 확인을 지원합니다 (*ssl.HAS_SNI*를 참조하십시오).

버전 3.9에서 변경: 선택적 *timeout* 매개 변수가 추가되었습니다.

버전 3.12에서 변경: 폐지된 *keyfile* 과 *certfile* 매개 변수를 제거했습니다.

두 번째 서브 클래스는 자식 프로세스가 만든 연결을 허용합니다:

class imaplib.**IMAP4_stream**(*command*)

이것은 *command*를 `subprocess.Popen()`에 전달하여 만들어진 `stdin/stdout` 파일 기술자에 연결하는 *IMAP4*에서 파생된 서브 클래스입니다.

다음과 같은 유틸리티 함수가 정의됩니다:

imaplib.**Internaldate2tuple**(*datestr*)

IMAP4 `INTERNALDATE` 문자열을 구문 분석하고 해당 지역 시간을 반환합니다. 반환 값은 *time.struct_time* 튜플이거나 문자열의 형식이 잘못되면 `None`입니다.

imaplib.**Int2AP**(*num*)

집합 [`A` .. `P`]의 문자를 사용하여 정수를 바이트열 표현으로 변환합니다.

imaplib.**ParseFlags**(*flagstr*)

IMAP4 `FLAGS` 응답을 개별 플래그의 튜플로 변환합니다.

imaplib.**Time2Internaldate**(*date_time*)

*date_time*을 IMAP4 `INTERNALDATE` 표현으로 변환합니다. 반환 값은 다음과 같은 형식의 문자열입니다: `"DD-Mmm-YYYY HH:MM:SS +HHMM"` (큰따옴표를 포함합니다). *date_time* 인자는 에포크 이후의 초(*time.time()*이 반환하는 것과 같습니다)를 나타내는 숫자 (int 또는 float), 지역 시간을 나타내는 9-튜플인 *time.struct_time*의 인스턴스 (*time.localtime()*이 반환하는 것과 같습니다), *datetime.datetime*의 어웨어(aware) 인스턴스 또는 큰따옴표로 인용된(double-quoted) 문자열일 수 있습니다. 마지막 경우에는, 이미 올바른 형식으로 되어 있다고 가정합니다.

사서함이 변경됨에 따라 IMAP4 메시지 번호가 변경됨에 유의하십시오. 특히, `EXPUNGE` 명령이 삭제를 수행한 후 나머지 메시지의 번호가 다시 매겨집니다. 따라서 UID 명령으로 UID를 대신 사용하는 것이 좋습니다.

모듈의 끝에는, 더 광범위한 사용 예제가 포함된 테스트 섹션이 있습니다.

더 보기

워싱턴 대학의 IMAP Information Center의 프로토콜을 설명하는 문서와 이를 구현하는 서버의 소스는 모두 (**소스 코드**) https://github.com/uw-imap/imap 에서 찾을 수 있습니다 (**유지 보수되지 않습니다**).

22.13.1 IMAP4 객체

모든 IMAP4rev1 명령은 대문자나 소문자의 같은 이름의 메서드로 표현됩니다.

`AUTHENTICATE`와 IMAP4 리터럴로 전달되는 `APPEND`에 대한 마지막 인자를 제외하고, 명령에 대한 모든 인자는 문자열로 변환됩니다. 필요하면 (문자열에 IMAP4 프로토콜에 참여하는 문자가 포함되고 괄호나 큰따옴표로 묶이지 않았으면) 각 문자열이 인용됩니다. 그러나, `LOGIN` 명령에 대한 *password* 인자는 항상

인용됩니다. 인자 문자열이 인용되지 않도록 하려면 (예를 들어: `STORE`에 대한 *flags* 인자) 문자열을 괄호로 묶으십시오 (예를 들어: `r'(\Deleted)'`).

Most commands return a tuple: `(type, [data, ...])` where *type* is usually `'OK'` or `'NO'`, and *data* is either the text from the command response, or mandated results from the command. Each *data* is either a `bytes`, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie: 'literal' value).

아래 명령에 대한 *message_set* 옵션은 수행할 하나 이상의 대상 메시지를 지정하는 문자열입니다. 단순 메시지 번호 (`'1'`), 메시지 번호 범위 (`'2:4'`) 또는 쉼표로 구분된 불연속 범위 그룹(`'1:3,6:9'`)일 수 있습니다. 범위에는 별표가 포함되어 무한 상한을 나타낼 수 있습니다 (`'3:*'`).

IMAP4 인스턴스에는 다음과 같은 메서드가 있습니다:

`IMAP4.`**`append`**(*mailbox*, *flags*, *date_time*, *message*)

명명된 사서함(mailbox)에 *message*를 추가합니다.

`IMAP4.`**`authenticate`**(*mechanism*, *authobject*)

인증 명령 — 응답 처리가 필요합니다.

*mechanism*은 사용할 인증 메커니즘을 지정합니다 - 인스턴스 변수 `capabilities`에 `AUTH=mechanism` 형식으로 나타나야 합니다.

*authobject*는 콜러블 객체여야 합니다:

```
data = authobject(response)
```

서버 계속(continuation) 응답을 처리하기 위해 호출됩니다; 전달된 *response* 인자는 `bytes`입니다. base64로 인코딩되어 서버로 전송되는 `bytes` *data*를 반환해야 합니다. 클라이언트 중단 응답 `*`를 대신 보내야 하면 `None`을 반환해야 합니다.

버전 3.5에서 변경: 문자열 사용자 이름과 비밀번호는 이제 ASCII로 제한되지 않고 `utf-8`로 인코딩됩니다.

`IMAP4.`**`check`**()

서버의 사서함을 검사합니다.

`IMAP4.`**`close`**()

현재 선택된 사서함을 닫습니다. 삭제된 메시지는 쓰기 가능한 사서함에서 제거됩니다. `LOGOUT` 이전의 권장 명령입니다.

`IMAP4.`**`copy`**(*message_set*, *new_mailbox*)

message_set 메시지를 *new_mailbox* 끝에 복사합니다.

`IMAP4.`**`create`**(*mailbox*)

*mailbox*라는 이름의 새 사서함을 만듭니다.

`IMAP4.`**`delete`**(*mailbox*)

*mailbox*라는 이름의 기존 사서함을 삭제합니다.

`IMAP4.`**`deleteacl`**(*mailbox*, *who*)

사서함(mailbox)의 사용자(who)에 대해 설정된 ACL을 삭제합니다 (모든 권한을 제거합니다).

`IMAP4.`**`enable`**(*capability*)

*capability*를 활성화합니다 (**RFC 5161**을 참조하십시오). 대부분의 기능은 활성화할 필요 없습니다. 현재 `UTF8=ACCEPT` 기능만 지원됩니다 (**RFC 6855**를 참조하십시오).

Added in version 3.5: *`enable()`* 메서드 자체와 **RFC 6855** 지원.

`IMAP4.`**`expunge`**()

선택한 사서함에서 삭제된 항목을 영구적으로 제거합니다. 삭제된 각 메시지에 대해 `EXPUNGE` 응답을 생성합니다. 반환된 데이터에는 수신된 순서의 `EXPUNGE` 메시지 번호 리스트가 포함됩니다.

`IMAP4.`**`fetch`**(*message_set*, *message_parts*)

메시지(일부)를 가져옵니다. *message_parts*는 괄호로 묶인 메시지 부분 이름의 문자열이어야 합니다, 예를 들어: `"(UID BODY[TEXT])"`). 반환된 데이터는 메시지 부분 봉투와 데이터의 튜플입니다.

IMAP4.**getacl**(*mailbox*)

*mailbox*에 대한 ACL을 가져옵니다. 이 메서드는 비표준이지만, Cyrus 서버에서 지원됩니다.

IMAP4.**getannotation**(*mailbox*, *entry*, *attribute*)

*mailbox*에 대해 지정된 ANNOTATION을 가져옵니다. 이 메서드는 비표준이지만, Cyrus 서버에서 지원됩니다.

IMAP4.**getquota**(*root*)

quota *root*의 자원 사용량과 제한을 가져옵니다. 이 메서드는 rfc2087에 정의된 IMAP4 QUOTA 확장의 일부입니다.

IMAP4.**getquotaroot**(*mailbox*)

명명된 *mailbox*에 대한 quota roots 리스트를 가져옵니다. 이 메서드는 rfc2087에 정의된 IMAP4 QUOTA 확장의 일부입니다.

IMAP4.**idle**(*duration=None*)

Return an Idler: an iterable context manager implementing the IMAP4 IDLE command as defined in **RFC 2177**.

The returned object sends the IDLE command when activated by the with statement, produces IMAP untagged responses via the *iterator* protocol, and sends DONE upon context exit.

All untagged responses that arrive after sending the IDLE command (including any that arrive before the server acknowledges the command) will be available via iteration. Any leftover responses (those not iterated in the with context) can be retrieved in the usual way after IDLE ends, using *IMAP4.response()*.

Responses are represented as (type, [data, ...]) tuples, as described in *IMAP4 Objects*.

The *duration* argument sets a maximum duration (in seconds) to keep idling, after which any ongoing iteration will stop. It can be an *int* or *float*, or None for no time limit. Callers wishing to avoid inactivity timeouts on servers that impose them should keep this at most 29 minutes (1740 seconds). Requires a socket connection; *duration* must be None on *IMAP4_stream* connections.

```
>>> with M.idle(duration=29 * 60) as idler:
...     for typ, data in idler:
...         print(typ, data)
...
EXISTS [b'1']
RECENT [b'1']
```

Idler.**burst**(*interval=0.1*)

Yield a burst of responses no more than *interval* seconds apart (expressed as an *int* or *float*).

This *generator* is an alternative to iterating one response at a time, intended to aid in efficient batch processing. It retrieves the next response along with any immediately available subsequent responses. (For example, a rapid series of EXPUNGE responses after a bulk delete.)

Requires a socket connection; does not work on *IMAP4_stream* connections.

```
>>> with M.idle() as idler:
...     # get a response and any others following by < 0.1 seconds
...     batch = list(idler.burst())
...     print(f'processing {len(batch)} responses...')
...     print(batch)
...
processing 3 responses...
[('EXPUNGE', [b'2']), ('EXPUNGE', [b'1']), ('RECENT', [b'0'])]
```

팁

The IDLE context' s maximum duration, as passed to *IMAP4.idle()*, is respected when waiting for the first response in a burst. Therefore, an expired Idler will cause this generator to return immediately without producing anything. Callers should consider this if using it in a loop.

> **참고**
>
> The iterator returned by *IMAP4.idle()* is usable only within a with statement. Before or after that context, unsolicited responses are collected internally whenever a command finishes, and can be retrieved with *IMAP4.response()*.

> **참고**
>
> The Idler class name and structure are internal interfaces, subject to change. Calling code can rely on its context management, iteration, and public method to remain stable, but should not subclass, instantiate, compare, or otherwise directly reference the class.

Added in version 3.14.

IMAP4.**list**([*directory*[, *pattern*]])

*directory*에 있는 *pattern*과 일치하는 사서함 이름을 나열합니다. *directory*는 기본적으로 최상위 메일 폴더이며, *pattern*은 기본적으로 모든 것과 일치합니다. 반환된 데이터는 LIST의 리스트를 포함합니다.

IMAP4.**login**(*user*, *password*)

평문 비밀번호를 사용하여 클라이언트를 식별합니다. *password*는 인용됩니다.

IMAP4.**login_cram_md5**(*user*, *password*)

암호를 보호하기 위해 클라이언트를 식별할 때 CRAM-MD5 인증의 사용을 강제합니다. 서버 CAPABILITY 응답에 문구 AUTH=CRAM-MD5가 포함된 경우에만 작동합니다.

버전 3.14에서 변경: An *IMAP4.error* is raised if MD5 support is not available.

IMAP4.**logout**()

서버에 대한 연결을 종료합니다. 서버 BYE 응답을 반환합니다.

버전 3.8에서 변경: 이 메서드는 더는 임의의 예외를 조용히 무시하지 않습니다.

IMAP4.**lsub**(*directory*='""', *pattern*='*')

directory에 있는 pattern과 일치하는 구독한(subscribed) 사서함 이름을 나열합니다. *directory*는 기본적으로 최상위 디렉터리이고 *pattern*은 기본적으로 모든 사서함과 일치합니다. 반환된 데이터는 메시지 부분 봉투와 데이터의 튜플입니다.

IMAP4.**myrights**(*mailbox*)

mailbox에 대한 현재 사용자의 ACL(즉, 사서함에 대해 가진 권한)을 표시합니다.

IMAP4.**namespace**()

RFC 2342에 정의된 대로 IMAP 이름 공간을 반환합니다.

IMAP4.**noop**()

서버에 NOOP을 보냅니다.

IMAP4.**open**(*host*, *port*, *timeout=None*)

*host*의 *port*로 소켓을 엽니다. 선택적 *timeout* 매개 변수는 연결 시도에 대한 시간제한을 초로 지정합니다. timeout이 제공되지 않거나 None이면, 전역 기본 소켓 시간제한이 사용됩니다. 또한 *timeout* 매개 변수가 0으로 설정되면, 비 블로킹 소켓을 만들지 못하도록 *ValueError*를 발생시킴에 유의하십시오. 이 메서드는 *IMAP4* 생성자에 의해 묵시적으로 호출됩니다. 이 메서드로 맺어진 연결 객체는 *IMAP4.read()*, *IMAP4.readline()*, *IMAP4.send()* 및 *IMAP4.shutdown()* 메서드에서 사용됩니다. 이 메서드를 재정의할 수 있습니다.

인자 self, host, port로 감사 이벤트 imaplib.open을 발생시킵니다.

버전 3.9에서 변경: *timeout* 매개 변수가 추가되었습니다.

IMAP4.**partial**(*message_num*, *message_part*, *start*, *length*)

메시지의 잘린 부분을 가져옵니다. 반환된 데이터는 메시지 부분 봉투와 데이터의 튜플입니다.

IMAP4.**proxyauth**(*user*)

인증을 *user*로 가정합니다. 권한 있는 관리자가 모든 사용자의 사서함으로 프락시 하도록 합니다.

IMAP4.**read**(*size*)

원격 서버에서 *size* 바이트를 읽습니다. 이 메서드를 재정의할 수 있습니다.

IMAP4.**readline**()

원격 서버에서 한 줄을 읽습니다. 이 메서드를 재정의할 수 있습니다.

IMAP4.**recent**()

업데이트를 서버에 요청합니다. 새 메시지가 없으면 반환된 데이터는 None이고, 그렇지 않으면 RECENT 응답의 값입니다.

IMAP4.**rename**(*oldmailbox*, *newmailbox*)

이름이 *oldmailbox*인 사서함의 이름을 *newmailbox*로 바꿉니다.

IMAP4.**response**(*code*)

수신되었다면 응답 *code*에 대한 데이터를 반환합니다, 또는 None을 반환합니다. 일반적인 유형 대신, 지정된 코드를 반환합니다.

IMAP4.**search**(*charset*, *criterion*[, ...])

일치하는 메시지가 있는지 사서함을 검색합니다. *charset*은 None일 수 있으며, 이 경우 서버에 대한 요청에 CHARSET이 지정되지 않습니다. IMAP 프로토콜은 적어도 하나의 기준을 지정하도록 요구합니다; 서버가 에러를 반환하면 예외가 발생합니다. *enable()* 명령을 사용하여 UTF8=ACCEPT 기능이 활성화되면 *charset*은 None이어야 합니다.

예:

```
# M은 연결된 IMAP4 인스턴스입니다...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# 또는:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.**select**(*mailbox='INBOX'*, *readonly=False*)

사서함을 선택합니다. 반환된 데이터는 *mailbox*에 있는 메시지 수입니다 (EXISTS 응답). 기본 *mailbox*는 'INBOX'입니다. *readonly* 플래그가 설정되면, 사서함을 수정할 수 없습니다.

IMAP4.**send**(*data*)

data를 원격 서버로 보냅니다. 이 메서드를 재정의할 수 있습니다.

인자 self, data로 감사 이벤트 imaplib.send를 발생시킵니다.

IMAP4.**setacl**(*mailbox*, *who*, *what*)

*mailbox*에 대한 ACL을 설정합니다. 이 메서드는 비표준이지만, Cyrus 서버에서 지원됩니다.

IMAP4.**setannotation**(*mailbox*, *entry*, *attribute*[, ...])

*mailbox*에 대한 ANNOTATION을 설정합니다. 이 메서드는 비표준이지만, Cyrus 서버에서 지원됩니다.

IMAP4.**setquota**(*root*, *limits*)

quota *root*의 자원 한도(*limits*)를 설정합니다. 이 메서드는 rfc2087에 정의된 IMAP4 QUOTA 확장의 일부입니다.

IMAP4.**shutdown**()

open에서 맺은 연결을 닫습니다. 이 메서드는 *IMAP4.logout()*에 의해 묵시적으로 호출됩니다. 이 메서드를 재정의할 수 있습니다.

IMAP4.**socket**()

서버에 연결하는 데 사용된 소켓 인스턴스를 반환합니다.

IMAP4.**sort**(*sort_criteria*, *charset*, *search_criterion*[, ...])

sort 명령은 결과에 대한 정렬이 추가된 search의 변형입니다. 반환된 데이터는 일치하는 메시지 번호의 공백으로 구분된 목록을 포함합니다.

sort에는 *search_criterion* 인자 앞에 두 개의 인자가 있습니다; *sort_criteria*의 괄호로 묶은 목록과 검색 *charset*. search와 달리, 검색 *charset* 인자는 필수임에 유의하십시오. uid search가 search에 해당하는 방식으로 sort에 해당하는 uid sort 명령도 있습니다. sort 명령은 먼저 검색 기준의 문자열 해석을 위해 charset 인자를 사용하여 주어진 검색 기준과 일치하는 메시지를 사서함에서 검색합니다. 그런 다음 일치하는 메시지 수를 반환합니다.

이것은 IMAP4rev1 확장 명령입니다.

IMAP4.**starttls**(*ssl_context=None*)

STARTTLS 명령을 보냅니다. *ssl_context* 인자는 선택적이며 *ssl.SSLContext* 객체여야 합니다. IMAP 연결의 암호화를 활성화합니다. 모범 사례는 보안 고려 사항을 읽으십시오.

Added in version 3.2.

버전 3.4에서 변경: 이 메서드는 이제 *ssl.SSLContext.check_hostname*과 서버 이름 표시(*Server Name Indication*)로 호스트명 확인을 지원합니다 (*ssl.HAS_SNI*를 참조하십시오).

IMAP4.**status**(*mailbox*, *names*)

*mailbox*에 대한 명명된 상태 조건(status conditions)을 요청합니다.

IMAP4.**store**(*message_set*, *command*, *flag_list*)

사서함의 메시지에 대한 플래그 속성을 변경합니다. *command*는 **RFC 2060**의 섹션 6.4.6에서 "FLAGS", "+ FLAGS" 또는 "-FLAGS" 중 하나로 지정되고, 선택적으로 ".SILENT" 접미사를 갖습니다.

예를 들어, 모든 메시지에 삭제 플래그를 설정하려면 다음을 수행합니다:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

> **참고**
>
> '] 을 포함하는 플래그를 만드는 것은 (예를 들어: "[test]") **RFC 3501**(IMAP 프로토콜)을 위반합니다. 그러나, imaplib는 역사적으로 이러한 태그를 만들 수 있도록 허락했고, Gmail과 같은 널리 사용되는 IMAP 서버는 이러한 플래그를 받아들이고 생성합니다. 역시 이러한 태그를 생성하는 비 파이썬 프로그램이 있습니다. RFC 위반이고 IMAP 클라이언트와 서버가 엄격해야 하기는 하지만, imaplib는 이전 버전과의 호환성을 위해 이러한 태그를 만들도록 계속 허락하며, 파이썬 3.6부터는 실제 호환성을 향상하기 때문에 서버에서 보낸다면 처리합니다.

IMAP4.**subscribe**(*mailbox*)

새 사서함을 구독합니다.

IMAP4.**thread**(*threading_algorithm*, *charset*, *search_criterion*[, ...])

thread 명령은 결과에 대한 스레딩이 추가된 search의 변형입니다. 반환된 데이터는 스레드 구성원의 스페이스로 구분된 목록을 포함합니다.

스레드 구성원은 연속된 부모와 자식을 나타내는 스페이스로 구분된 0개 이상의 메시지 번호로 구성됩니다.

thread에는 *search_criterion* 인자 앞에 두 개의 인자가 있습니다; *threading_algorithm*과 검색 *charset*. search와 달리, 검색 *charset* 인자는 필수임에 유의하십시오. uid search가 search에 해당하는 방식으로 thread에 해당하는 uid thread 명령도 있습니다. thread 명령은 먼저 검색 기준의 문자열

해석을 위해 *charset* 인자를 사용하여 주어진 검색 기준과 일치하는 메시지를 사서함에서 검색합니다. 그런 다음 지정된 스레딩 알고리즘에 따라 스레드 된 일치 메시지들을 반환합니다.

이것은 `IMAP4rev1` 확장 명령입니다.

`IMAP4.`**`uid`**(*command*, *arg*[, ...])

메시지 번호가 아닌 UID로 식별된 메시지들로 명령 인자를 실행합니다. 명령에 적합한 응답을 반환합니다. 최소한 하나의 인자가 제공되어야 합니다; 아무것도 제공하지 않으면, 서버는 에러를 반환하고 예외가 발생합니다.

`IMAP4.`**`unsubscribe`**(*mailbox*)

기존 사서함에서 구독 취소합니다.

`IMAP4.`**`unselect`**()

*imaplib.IMAP4.unselect()*는 선택한 사서함과 관련된 서버 자원을 해제하고 서버를 인증된 상태로 되돌립니다. 이 명령은 현재 선택된 사서함에서 메시지가 영구적으로 제거되지 않는다는 점을 제외하고, *imaplib.IMAP4.close()*와 같은 작업을 수행합니다.

Added in version 3.9.

`IMAP4.`**`xatom`**(*name*[, ...])

`CAPABILITY` 응답에서 서버가 통지한 간단한 확장 명령을 허용합니다.

IMAP4 인스턴스에는 다음과 같은 어트리뷰트가 정의되어 있습니다:

`IMAP4.`**`PROTOCOL_VERSION`**

서버의 `CAPABILITY` 응답에서 가장 최근에 지원되는 프로토콜.

`IMAP4.`**`debug`**

디버깅 출력을 제어하기 위한 정숫값. 초기화 값은 모듈 변수 `Debug`에서 취합니다. 3보다 큰 값은 각 명령을 추적합니다.

`IMAP4.`**`utf8_enabled`**

일반적으로 `False`이지만, `UTF8=ACCEPT` 기능에 대해 *enable()* 명령이 성공적으로 발행되면 `True`로 설정되는 불리언 값.

Added in version 3.5.

22.13.2 IMAP4 예

다음은 사서함을 열고 모든 메시지를 가져오고 인쇄하는 최소한의 예(에러 검사는 없습니다)입니다:

```
import getpass, imaplib

M = imaplib.IMAP4(host='example.org')
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print('Message %s\n%s\n' % (num, data[0][1]))
M.close()
M.logout()
```

22.14 smtplib — SMTP 프로토콜 클라이언트

소스 코드: Lib/smtplib.py

---

smtplib 모듈은 SMTP나 ESMTP 리스너 데몬을 사용하여 모든 인터넷 기계로 메일을 보내는 데 사용할 수 있는 SMTP 클라이언트 세션 객체를 정의합니다. SMTP와 ESMTP 연산에 대한 자세한 내용은 **RFC 821**(Simple Mail Transfer Protocol)과 **RFC 1869**(SMTP Service Extensions)를 참조하십시오.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

class smtplib.**SMTP**(*host=''*, *port=0*, *local_hostname=None*, [*timeout*,]*source_address=None*)

SMTP 인스턴스는 SMTP 연결을 캡슐화합니다. SMTP와 ESMTP 연산의 전체 레퍼토리를 지원하는 메서드가 있습니다. 선택적 *host*와 *port* 매개 변수가 제공되면, 초기화 중에 해당 매개 변수로 SMTP *connect()* 메서드가 호출됩니다. 지정되면, HELO/EHLO 명령에서 *local_hostname*이 로컬 호스트의 FQDN으로 사용됩니다. 그렇지 않으면 *socket.getfqdn()*을 사용하여 로컬 호스트 이름을 찾습니다. *connect()* 호출이 성공 코드 이외의 것을 반환하면 *SMTPConnectError*가 발생합니다. 선택적 *timeout* 매개 변수는 연결 시도와 같은 블로킹 연산을 위한 시간제한을 초로 지정합니다 (지정하지 않으면, 전역 기본 시간제한 설정이 사용됩니다). 시간제한이 만료되면 *TimeoutError*이 발생합니다. 선택적 *source_address* 매개 변수는 여러 네트워크 인터페이스가 있는 기계의 특정 소스 주소 그리고/또는 특정 소스 TCP 포트에 바인딩 할 수 있도록 합니다. 연결 전에 소켓이 소스 주소로 바인드 할 2-튜플 (host, port)를 취합니다. 생략되면 (또는 *host*나 *port*가 각각 '' 및/또는 0이면) OS 기본 동작이 사용됩니다.

일반적인 사용을 위해서는, 초기화/연결, *sendmail()* 및 *SMTP.quit()* 메서드 만 필요합니다. 아래에 예가 포함되어 있습니다.

SMTP 클래스는 with 문을 지원합니다. 이처럼 사용되면, with 문이 종료될 때 SMTP QUIT 명령이 자동으로 발행됩니다. 예를 들어:

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

모든 명령은 인자 self와 data로 감사 이벤트 smtplib.SMTP.send를 발생시킵니다. 여기서 data는 원격 호스트로 전송될 바이트열입니다.

버전 3.3에서 변경: with 문에 대한 지원이 추가되었습니다.

버전 3.3에서 변경: *source_address* 인자가 추가되었습니다.

Added in version 3.5: SMTPUTF8 확장(**RFC 6531**)이 이제 지원됩니다.

버전 3.9에서 변경: *timeout* 매개 변수가 0으로 설정되면, 비 블로킹 소켓을 만드는 것을 막기 위해 *ValueError*가 발생합니다.

class smtplib.**SMTP_SSL**(*host=''*, *port=0*, *local_hostname=None*, *, [*timeout*,]*context=None*, *source_address=None*)

SMTP_SSL 인스턴스는 *SMTP* 인스턴스와 정확히 같게 작동합니다. 연결 시작 시 SSL이 필요하고 starttls() 사용이 적합하지 않은 상황에서는 *SMTP_SSL*을 사용해야 합니다. *host*를 지정하지 않으면, 로컬 호스트가 사용됩니다. *port*가 0이면, 표준 SMTP-over-SSL 포트(465)가 사용됩니다. 선택적 인자 *local_hostname*, *timeout* 및 *source_address*는 *SMTP* 클래스에서와 같은 의미입니다. 역시 선택적인 *context*는 *SSLContext*를 포함할 수 있으며 보안 연결의 다양한 측면을 구성하도록 합니다. 모범 사례는 보안 고려 사항을 읽으십시오.

버전 3.3에서 변경: *context*가 추가되었습니다.

버전 3.3에서 변경: *source_address* 인자가 추가되었습니다.

버전 3.4에서 변경: 이 클래스는 이제 *ssl.SSLContext.check_hostname*과 서버 이름 표시(*Server Name Indication*)로 호스트 이름 확인을 지원합니다 (*ssl.HAS_SNI*를 참조하십시오).

버전 3.9에서 변경: *timeout* 매개 변수가 0으로 설정되면, 비 블로킹 소켓을 만드는 것을 막기 위해 *ValueError*가 발생합니다.

버전 3.12에서 변경: 폐지된 *keyfile* 과 *certfile* 매개 변수는 제거했습니다.

class smtplib.**LMTP**(*host=''*, *port=LMTP_PORT*, *local_hostname=None*, *source_address=None*[, *timeout*])

ESMTP와 매우 유사한 LMTP 프로토콜은 표준 SMTP 클라이언트를 기반으로 합니다. LMTP에 유닉스 소켓을 사용하는 것이 일반적이라서, `connect()` 메서드는 일반 host:port 서버뿐만 아니라 이를 지원해야 합니다. 선택적 인자 *local_hostname*과 *source_address*는 `SMTP` 클래스에서와 같은 의미입니다. 유닉스 소켓을 지정하려면, *host*에 '/' 로 시작하는 절대 경로를 사용해야 합니다.

일반 SMTP 메커니즘을 사용하여 인증이 지원됩니다. 유닉스 소켓을 사용할 때, LMTP는 일반적으로 인증을 지원하지 않거나 요구하지 않지만, 여러분의 상황에서는 다를 수 있습니다.

버전 3.9에서 변경: 선택적 *timeout* 매개 변수가 추가되었습니다.

훌륭한 예외 모음도 정의되어 있습니다:

exception smtplib.**SMTPException**

이 모듈에서 제공하는 다른 모든 예외에 대한 베이스 예외 클래스인 `OSError`의 서브 클래스.

버전 3.4에서 변경: SMTPException이 `OSError`의 서브 클래스가 되었습니다.

exception smtplib.**SMTPServerDisconnected**

이 예외는 예기치 않게 서버와의 연결이 끊어지거나, 서버에 연결하기 전에 `SMTP` 인스턴스를 사용하려고 할 때 발생합니다.

exception smtplib.**SMTPResponseException**

SMTP 에러 코드가 포함된 모든 예외의 베이스 클래스. 이러한 예외는 SMTP 서버가 에러 코드를 반환하는 일부 경우에 생성됩니다. 에러 코드는 에러의 `smtp_code` 어트리뷰트에 저장되며, `smtp_error` 어트리뷰트는 에러 메시지로 설정됩니다.

exception smtplib.**SMTPSenderRefused**

발신자 주소가 거부되었습니다. 모든 `SMTPResponseException` 예외에 의해 설정된 어트리뷰트 외에도, 이것은 'sender' 를 SMTP 서버가 거부한 문자열로 설정합니다.

exception smtplib.**SMTPRecipientsRefused**

모든 수신자 주소가 거부되었습니다. 각 수신자의 에러는 `SMTP.sendmail()`이 반환하는 것과 정확히 같은 종류의 딕셔너리인 `recipients` 어트리뷰트를 통해 액세스 할 수 있습니다.

exception smtplib.**SMTPDataError**

SMTP 서버가 메시지 데이터 수락을 거부했습니다.

exception smtplib.**SMTPConnectError**

서버와의 연결을 설정하는 동안 에러가 발생했습니다.

exception smtplib.**SMTPHeloError**

서버가 `HELO` 메시지를 거부했습니다.

exception smtplib.**SMTPNotSupportedError**

시도한 명령이나 옵션이 서버에서 지원되지 않습니다.

Added in version 3.5.

exception smtplib.**SMTPAuthenticationError**

SMTP 인증이 잘못되었습니다. 서버가 제공된 username/password 조합을 수락하지 않았을 가능성이 높습니다.

더 보기

RFC 821 - Simple Mail Transfer Protocol
SMTP의 프로토콜 정의. 이 문서는 SMTP의 모델, 운영 절차 및 프로토콜 세부 사항을 다룹니다.

RFC 1869 - SMTP Service Extensions
SMTP를 위한 ESMTP 확장의 정의. 이 문서는 새로운 명령으로 SMTP를 확장하고 서버에서 제공하는 명령의 동적 발견을 지원하는 프레임워크에 관해 설명하고, 몇 가지 추가 명령을 정의합니다.

22.14.1 SMTP 객체

SMTP 인스턴스에는 다음과 같은 메서드가 있습니다:

SMTP.**set_debuglevel**(*level*)

디버그 출력 수준을 설정합니다. *level*의 값이 1이나 `True`이면 연결과 서버와 주고받는 모든 메시지에 대한 디버그 메시지가 생성됩니다. *level*의 값이 2이면 이러한 메시지에 타임 스탬프가 추가됩니다.

버전 3.5에서 변경: 디버그 수준 2를 추가했습니다.

SMTP.**docmd**(*cmd*, *args=''*)

서버에 *cmd* 명령을 전송합니다. 선택적 인자 *args*는 단순히 공백으로 구분하여 명령에 이어붙입니다.

숫자 응답 코드와 실제 응답 줄로 구성된 2-튜플을 반환합니다 (여러 줄 응답은 하나의 긴 줄로 결합합니다).

일반적인 작업에서 이 메서드를 명시적으로 호출할 필요는 없습니다. 다른 메서드를 구현하는 데 사용되며 개인적인 확장을 테스트하는 데 유용할 수 있습니다.

응답을 기다리는 동안 서버와의 연결이 끊어지면, *SMTPServerDisconnected* 가 발생합니다.

SMTP.**connect**(*host='localhost'*, *port=0*)

지정된 포트(port)의 호스트(host)에 연결합니다. 기본값은 표준 SMTP 포트(25)로 로컬 호스트에 연결하는 것입니다. 호스트 이름이 콜론(`':'`)으로 끝나고 그 뒤에 숫자가 오면 해당 접미사가 제거되고 숫자는 사용할 포트 번호로 해석됩니다. 이 메서드는 인스턴스 화 중에 호스트가 지정되면 생성자에 의해 자동으로 호출됩니다. 연결 응답에서 서버가 보낸 응답 코드와 메시지의 2-튜플을 반환합니다.

인자 `self`, `host`, `port`로 감사 이벤트 `smtplib.connect`를 발생시킵니다.

SMTP.**helo**(*name=''*)

`HELO`를 사용하여 SMTP 서버에 자신을 식별합니다. hostname 인자의 기본값은 로컬 호스트의 완전히 정규화된 도메인 이름(fully qualified domain name) 입니다. 서버가 반환한 메시지는 객체의 `helo_resp` 어트리뷰트로 저장됩니다.

정상적인 작업에서는 이 메서드를 명시적으로 호출할 필요가 없습니다. 필요할 때 *sendmail()*에 의해 묵시적으로 호출됩니다.

SMTP.**ehlo**(*name=''*)

`EHLO`를 사용하여 ESMTP 서버에 자신을 식별합니다. hostname 인자의 기본값은 로컬 호스트의 완전히 정규화된 도메인 이름(fully qualified domain name) 입니다. ESMTP 옵션에 대한 응답을 검사하고 *has_extn()*에서 사용할 수 있도록 저장합니다. 또한 여러 정보 어트리뷰트를 설정합니다: 서버가 반환한 메시지는 `ehlo_resp` 어트리뷰트로 저장되고, 서버가 ESMTP를 지원하는지에 따라 `does_esmtp`가 `True`나 `False`로 설정되며, `esmtp_features`는 이 서버가 지원하는 SMTP 서비스 확장의 이름과 그 매개 변수(있다면)를 포함하는 딕셔너리가 됩니다.

메일을 보내기 전에 *has_extn()*을 사용하고 싶지 않은 한, 이 메서드를 명시적으로 호출할 필요는 없습니다. 필요할 때 *sendmail()*에 의해 묵시적으로 호출됩니다.

SMTP.**ehlo_or_helo_if_needed**()

이 세션에서 앞선 `EHLO`나 `HELO` 명령이 없으면, 이 메서드는 *ehlo()* 및/또는 *helo()*를 호출합니다. ESMTP `EHLO`를 먼저 시도합니다.

SMTPHeloError

서버가 `HELO` 인사에 올바르게 응답하지 않았습니다.

SMTP.**has_extn**(*name*)

*name*이 서버가 반환한 SMTP 서비스 확장 집합에 있으면 *True*를 반환하고, 그렇지 않으면 *False*를 반환합니다. 대소 문자는 무시됩니다.

SMTP.**verify**(*address*)

SMTP `VRFY`를 사용하여 이 서버에서 주소의 유효성을 확인합니다. 사용자 주소가 유효하면 코드 250과 전체 **RFC 822** 주소(사람 이름 포함)로 구성된 튜플을 반환합니다. 그렇지 않으면 400 이상의 SMTP 에러 코드와 에러 문자열을 반환합니다.

> **참고**
>
> 많은 사이트에서 스패머를 제거하기 위해 SMTP `VRFY`를 비활성화합니다.

`SMTP.login`(*user*, *password*, ***, *initial_response_ok=True*)

인증이 필요한 SMTP 서버에 로그인합니다. 인자는 인증할 사용자 이름과 비밀번호입니다. 이 세션에 앞선 `EHLO`나 `HELO` 명령이 없었으면, 이 메서드는 ESMTP `EHLO`를 먼저 시도합니다. 인증에 성공하면 이 메서드는 정상적으로 반환하고, 그렇지 않으면 다음 예외를 발생시킬 수 있습니다:

SMTPHeloError
서버가 `HELO` 인사에 올바르게 응답하지 않았습니다.

SMTPAuthenticationError
서버가 사용자 이름/비밀번호 조합을 수락하지 않았습니다.

SMTPNotSupportedError
서버가 `AUTH` 명령을 지원하지 않습니다.

SMTPException
적합한 인증 메서드가 없습니다.

*smtplib*가 지원하는 각 인증 메서드는 서버가 지원하는 것으로 광고되면 차례로 시도됩니다. 지원되는 인증 메서드 목록은 *auth()*를 참조하십시오. *initial_response_ok*는 *auth()*로 전달됩니다.

선택적 키워드 인자 *initial_response_ok*는, 이를 지원하는 인증 메서드의 경우, 챌린지/응답을 요구하지 않고 **RFC 4954**에 지정된 "초기 응답(initial response)"을 `AUTH` 명령과 함께 보낼 수 있는지를 지정합니다.

버전 3.5에서 변경: *SMTPNotSupportedError* 가 발생할 수 있고, *initial_response_ok* 매개 변수가 추가되었습니다.

`SMTP.auth`(*mechanism*, *authobject*, ***, *initial_response_ok=True*)

지정된 인증 메커니즘(*mechanism*)으로 `SMTP AUTH` 명령을 발행하고, *authobject*를 통해 챌린지 응답을 처리합니다.

*mechanism*은 `AUTH` 명령의 인자로 사용할 인증 메커니즘을 지정합니다; 유효한 값은 `esmtp_features`의 `auth` 요소에 나열된 값입니다.

*authobject*는 선택적 단일 인자를 취하는 콜러블 객체여야 합니다:

```
data = authobject(challenge=None)
```

선택적 키워드 인자 *initial_response_ok*가 참이면, 인자 없이 `authobject()` 가 먼저 호출됩니다. 이것은 **RFC 4954** "초기 응답(initial response)" ASCII `str`을 반환할 수 있으며, 이는 아래와 같이 `AUTH` 명령으로 인코딩되고 전송됩니다. `authobject()` 가 초기 응답을 지원하지 않으면 (예를 들어 챌린지가 필요하기 때문에), `challenge=None`으로 호출될 때 `None`을 반환해야 합니다. *initial_response_ok*가 거짓이면, `authobject()` 는 `None`으로 먼저 호출되지 않습니다.

초기 응답 확인이 `None`을 반환하거나, *initial_response_ok*가 거짓이면, 서버의 챌린지 응답을 처리하기 위해 `authobject()` 가 호출됩니다; 전달된 *challenge* 인자는 `bytes`입니다. base64로 인코딩되어 서버로 전송될 ASCII `str` *data*를 반환해야 합니다.

`SMTP` 클래스는 `CRAM-MD5`, `PLAIN` 및 `LOGIN` 메커니즘을 위한 `authobjects`를 제공합니다. 그것들은 각각 `SMTP.auth_cram_md5`, `SMTP.auth_plain` 및 `SMTP.auth_login`으로 명명됩니다. 모두 `SMTP` 인스턴스의 `user`와 `password` 프로퍼티가 적절한 값으로 설정되어 있도록 요구합니다.

사용자 코드는 일반적으로 `auth`를 직접 호출할 필요는 없지만, 대신 *login()* 메서드를 호출할 수는 있는데, 위의 각 메커니즘을 나열된 순서대로 시도합니다. `auth`는 *smtplib*가 직접 지원하지 않는 (또는 아직 지원하지 않는) 인증 메서드를 쉽게 구현할 수 있도록 노출되어 있습니다.

Added in version 3.5.

SMTP.**starttls**(*, *context=None*)

SMTP 연결을 TLS (Transport Layer Security) 모드로 전환합니다. 뒤따르는 모든 SMTP 명령은 암호화됩니다. 그런 다음 `ehlo()`를 다시 호출해야 합니다.

*keyfile*과 *certfile*이 제공되면, `ssl.SSLContext`를 만드는 데 사용됩니다.

선택적 *context* 매개 변수는 `ssl.SSLContext` 객체입니다; 이것은 keyfile과 certfile 대신 사용할 수 있으며 *keyfile*과 *certfile*이 모두 지정되면 `None`이어야 합니다.

이 세션에 앞선 `EHLO`나 `HELO` 명령이 없었으면, 이 메서드는 ESMTP `EHLO`를 먼저 시도합니다.

버전 3.12에서 변경: 폐지된 *keyfile* 과 *certfile* 매개 변수는 제거했습니다.

SMTPHeloError

서버가 `HELO` 인사에 올바르게 응답하지 않았습니다.

SMTPNotSupportedError

서버가 STARTTLS 확장을 지원하지 않습니다.

RuntimeError

파이썬 인터프리터가 SSL/TLS를 지원하지 않습니다.

버전 3.3에서 변경: *context*가 추가되었습니다.

버전 3.4에서 변경: 이 메서드는 이제 `SSLContext.check_hostname`과 서버 이름 표시(*Server Name Indicator*)로 호스트 이름 확인을 지원합니다 (`HAS_SNI`를 참조하십시오).

버전 3.5에서 변경: STARTTLS 지원 부족으로 발생한 에러는 이제 베이스 `SMTPException` 대신 `SMTPNotSupportedError` 서브 클래스입니다.

SMTP.**sendmail**(*from_addr*, *to_addrs*, *msg*, *mail_options=()*, *rcpt_options=()*)

메일을 보냅니다. 필수 인자는 **RFC 822** from-address 문자열, **RFC 822** to-address 문자열의 리스트 (단순 문자열은 주소가 한 개인 리스트로 처리됩니다) 및 메시지 문자열입니다. 호출자는 `MAIL FROM` 명령에 사용되는 ESMTP 옵션 (가령 `8bitmime`) 리스트를 *mail_options*로 전달할 수 있습니다. 모든 `RCPT` 명령과 함께 사용해야 하는 ESMTP 옵션(가령 `DSN` 명령)을 *rcpt_options*로 전달할 수 있습니다. (다른 수신자에게 다른 ESMTP 옵션을 사용해야 하면 메시지를 보내는 데 `mail()`, `rcpt()` 및 `data()`와 같은 저수준 메서드를 사용해야 합니다.)

> **참고**
>
> *from_addr*과 *to_addrs* 매개 변수는 전송 에이전트가 사용하는 메시지 봉투(envelope)를 구성하는 데 사용됩니다. `sendmail`은 어떤 방식으로든 메시지 헤더를 수정하지 않습니다.

*msg*는 ASCII 범위의 문자를 포함하는 문자열이거나, 바이트 문자열일 수 있습니다. 문자열은 ascii 코덱을 사용하여 바이트열로 인코딩되며, 독립된 `\r`과 `\n` 문자는 `\r\n` 문자로 변환됩니다. 바이트 문자열은 수정되지 않습니다.

이 세션에 앞선 `EHLO`나 `HELO` 명령이 없었으면, 이 메서드는 ESMTP `EHLO`를 먼저 시도합니다. 서버가 ESMTP를 지원하면, 메시지 크기와 지정된 각 옵션이 전달됩니다 (옵션이 서버가 광고한 기능 집합에 있다면). `EHLO`가 실패하면, `HELO`가 시도되고 ESMTP 옵션이 억제됩니다.

이 메서드는 적어도 한 명의 수신자에게 메일이 수락되면 정상적으로 반환됩니다. 그렇지 않으면 예외가 발생합니다. 즉, 이 메서드로 예외가 발생하지 않으면 누군가 메일을 받아야 합니다. 이 메서드에서 예외가 발생하지 않으면, 거부된 각 수신자에 대해 하나의 항목이 있는 딕셔너리를 반환합니다. 각 항목에는 서버에서 보낸 SMTP 에러 코드와 해당 에러 메시지의 튜플이 포함됩니다.

`SMTPUTF8`이 *mail_options*에 포함되어 있고, 서버가 이를 지원하면, *from_addr*과 *to_addrs*는 ASCII가 아닌 문자를 포함할 수 있습니다.

이 메서드는 다음과 같은 예외를 발생시킬 수 있습니다:

SMTPRecipientsRefused

모든 수신자가 거부되었습니다. 아무도 메일을 받지 못했습니다. 예외 객체의 `recipients` 어트리뷰트는 거부된 수신자에 대한 정보가 있는 딕셔너리입니다 (적어도 한 명의 수신자가 수락되었을 때 반환되는 것과 같습니다).

SMTPHeloError
서버가 HELO 인사에 올바르게 응답하지 않았습니다.

SMTPSenderRefused
서버가 *from_addr*을 수락하지 않았습니다.

SMTPDataError
서버가 예기치 않은 에러 코드(수신자 거부는 제외하고)로 응답했습니다.

SMTPNotSupportedError
SMTPUTF8이 *mail_options*에 제공되었지만, 서버에서 지원되지 않습니다.

달리 명시되지 않는 한, 예외가 발생한 후에도 연결은 열려있습니다.

버전 3.2에서 변경: *msg*는 바이트 문자열일 수 있습니다.

버전 3.5에서 변경: SMTPUTF8 지원이 추가되었으며, SMTPUTF8이 지정되었지만, 서버가 지원하지 않으면 *SMTPNotSupportedError* 가 발생할 수 있습니다.

SMTP.**send_message**(*msg*, *from_addr=None*, *to_addrs=None*, *mail_options=()*, *rcpt_options=()*)

이는 *email.message.Message* 객체로 표현되는 메시지로 *sendmail()*을 호출하는 편의 메서드입니다. 인자는 *msg*가 Message 객체라는 점을 제외하고 *sendmail()*과 같은 의미입니다.

*from_addr*이 None이거나 *to_addrs*가 None이면, send_message는 **RFC 5322** 에 지정된 대로 *msg*의 헤더에서 추출된 주소로 해당 인자를 채웁니다: *from_addr*은 *Sender* 필드가 있으면 이것으로 설정되고, 그렇지 않으면 *From* 필드로 설정됩니다. *to_addrs*는 (있다면) *msg*의 *To*, *Cc* 및 *Bcc* 필드 값을 결합합니다. 정확히 하나의 *Resent-** 헤더 집합이 메시지에 등장하면, 일반 헤더는 무시되고 *Resent-** 헤더가 대신 사용됩니다. 메시지에 둘 이상의 *Resent-** 헤더 집합이 포함되면, 가장 최근의 *Resent-* 헤더 집합을 모호하지 않게 감지할 방법이 없어서 *ValueError*가 발생합니다.

send_message는 \r\n을 *linesep*으로 사용하여 *BytesGenerator*를 통해 *msg*를 직렬화하고, *sendmail()*을 호출하여 결과 메시지를 전송합니다. *from_addr*과 *to_addrs*의 값과 관계없이 send_message는 *msg*에 나타날 수 있는 *Bcc*나 *Resent-Bcc* 헤더를 전송하지 않습니다. *from_addr*과 *to_addrs*의 주소 중 하나에 ASCII가 아닌 문자가 포함되어 있고 서버가 SMTPUTF8 지원을 광고하지 않으면, *SMTPNotSupportedError*가 발생합니다. 그렇지 않으면 Message는 *utf8* 어트리뷰트가 True로 설정된 자신의 *policy*의 복제본으로 직렬화되고, SMTPUTF8과 BODY=8BITMIME이 *mail_options*에 추가됩니다.

Added in version 3.2.

Added in version 3.5: 국제화된 주소 (SMTPUTF8) 지원.

SMTP.**quit**()

SMTP 세션을 종료하고 연결을 닫습니다. SMTP QUIT 명령의 결과를 반환합니다.

표준 SMTP/ESMTP 명령 HELP, RSET, NOOP, MAIL, RCPT 및 DATA에 해당하는 저수준 메서드도 지원됩니다. 일반적으로 이들은 직접 호출할 필요가 없어서, 여기에 설명되어 있지 않습니다. 자세한 내용은, 모듈 코드를 참조하십시오.

22.14.2 SMTP 예

이 예에서는 메시지 봉투(envelope)에 필요한 주소('To' 와 'From' 주소)와 전달할 메시지를 사용자에게 요구합니다. 메시지에 포함할 헤더는 입력한 대로 메시지에 포함됨에 유의하십시오; 이 예제는 **RFC 822** 헤더를 처리하지 않습니다. 특히, 'To' 와 'From' 주소는 메시지 헤더에 명시적으로 포함되어야 합니다:

```
import smtplib

def prompt(title):
    return input(title).strip()

from_addr = prompt("From: ")
to_addrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# From: 과 To: 헤더를 시작에 넣습니다!
lines = [f"From: {from_addr}", f"To: {', '.join(to_addrs)}", ""]
while True:
    try:
        line = input()
    except EOFError:
        break
    else:
        lines.append(line)

msg = "\r\n".join(lines)
print("Message length is", len(msg))

server = smtplib.SMTP("localhost")
server.set_debuglevel(1)
server.sendmail(from_addr, to_addrs, msg)
server.quit()
```

참고

일반적으로, `email` 패키지의 기능을 사용하여 전자 메일 메시지를 만들고자 할 것이고, 그런 다음 `send_message()`를 통해 보낼 수 있습니다; *email: 예제*를 참조하십시오.

22.15 `uuid` — UUID objects according to RFC 9562

소스 코드: Lib/uuid.py

This module provides immutable `UUID` objects (the `UUID` class) and *functions* for generating UUIDs corresponding to a specific UUID version as specified in **RFC 9562** (which supersedes **RFC 4122**), for example, `uuid1()` for UUID version 1, `uuid3()` for UUID version 3, and so on. Note that UUID version 2 is deliberately omitted as it is outside the scope of the RFC.

고유 ID만을 얻고자 한다면 `uuid1()` 또는 `uuid4()`를 호출하는 것이 좋습니다. `uuid1()` 함수는 컴퓨터의 네트워크 주소를 포함하여 UUID를 생성하므로 개인정보가 노출될 수 있음을 명심해야 합니다. `uuid4()`는 무작위 UUID를 생성합니다.

기본 플랫폼의 지원 여부에 따라, `uuid1()`은 "안전한" UUID를 반환하거나 그렇지 않을 수도 있습니다. 안전한 UUID는 두 프로세스가 같은 UUID를 얻지 못하게 하는 동기화 메서드에 의해 생성됩니다. `UUID`의 모든 인스턴스는 UUID의 안정성에 대한 정보를 중계하는 `is_safe` 어트리뷰트가 있고, 다음의 열거체를 사용합니다.

class uuid.**SafeUUID**

Added in version 3.7.

safe

플랫폼이 다중 프로세스에 안전한 방식으로 UUID를 생성합니다.

unsafe

다중 프로세스에 안전한 방식으로 생성되지 않습니다.

unknown

플랫폼이 UUID를 안전하게 생성하였는지에 대한 정보를 제공하지 않습니다.

class uuid.**UUID**(*hex=None, bytes=None, bytes_le=None, fields=None, int=None, version=None, *, is_safe=SafeUUID.unknown*)

32자리 16진수 문자열, *bytes* 인자에 빅 엔디안 순서 16바이트열, *bytes_le* 인자에 리틀 엔디안 순서 16바이트열, *fields* 인자에 여섯 개의 정수로 이루어진 튜플(32-bit *time_low*, 16-bit *time_mid*, 16-bit *time_hi_version*, 8-bit *clock_seq_hi_variant*, 8-bit *clock_seq_low*, 48-bit *node*), *int* 인자에 단일 128bit 정수 중 하나를 이용하여 UUID를 만듭니다. 16진수 문자열로 주어졌을 경우 중괄호, 붙임표(hyphen), URN 접두어는 모두 선택사항입니다. 예를 들어, 아래 표현들은 모두 같은 UUID를 산출합니다:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56' +
              b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

Exactly one of *hex*, *bytes*, *bytes_le*, *fields*, or *int* must be given. The *version* argument is optional; if given, the resulting UUID will have its variant and version number set according to **RFC 9562**, overriding bits in the given *hex*, *bytes*, *bytes_le*, *fields*, or *int*.

UUID 객체끼리 비교할 때 `UUID.int` 어트리뷰트를 비교하여 수행합니다. UUID가 아닌 객체와 비교하면 `TypeError`가 발생합니다.

`str(uuid)` 는 `12345678-1234-5678-1234-567812345678`과같이 32자리 16진수 UUID로 표현합니다.

`UUID` 객체는 다음과 같은 읽기 전용 어트리뷰트를 갖습니다.

`UUID.`**`bytes`**

16바이트 문자열(여섯 개의 빅 엔디안 순서 정수 필드 포함)인 UUID

`UUID.`**`bytes_le`**

16바이트 문자열(리틀 엔디안 순서 *time_low*, *time_mid*, *time_hi_version*)인 UUID

`UUID.`**`fields`**

UUID의 여섯 개의 정수 필드로 이루어진 튜플. 여섯 개의 개별 어트리뷰트와 두 개의 파생된 어트리뷰트도 사용 가능:

필드	의미
UUID.**time_low**	The first 32 bits of the UUID. Only relevant to version 1.
UUID.**time_mid**	The next 16 bits of the UUID. Only relevant to version 1.
UUID.**time_hi_version**	The next 16 bits of the UUID. Only relevant to version 1.
UUID.**clock_seq_hi_variant**	The next 8 bits of the UUID. Only relevant to versions 1 and 6.
UUID.**clock_seq_low**	The next 8 bits of the UUID. Only relevant to versions 1 and 6.
UUID.**node**	The last 48 bits of the UUID. Only relevant to version 1.
UUID.**time**	The 60-bit timestamp as a count of 100-nanosecond intervals since Gregorian epoch (1582-10-15 00:00:00) for versions 1 and 6, or the 48-bit timestamp in milliseconds since Unix epoch (1970-01-01 00:00:00) for version 7.
UUID.**clock_seq**	The 14-bit sequence number. Only relevant to versions 1 and 6.

UUID.**hex**

32자리 소문자 16진수 문자열 UUID.

UUID.**int**

128bit 정수 UUID

UUID.**urn**

The UUID as a URN as specified in **RFC 9562**.

UUID.**variant**

UUID 변종은 UUID의 내부 레이아웃을 결정합니다. 이는 상수 *RESERVED_NCS*, *RFC_4122*, *RESERVED_MICROSOFT*, 및 *RESERVED_FUTURE* 중 하나입니다.

UUID.**version**

The UUID version number (1 through 8, meaningful only when the variant is *RFC_4122*).

버전 3.14에서 변경: Added UUID versions 6, 7 and 8.

UUID.**is_safe**

플랫폼이 UUID를 다중 프로세스에 안전한 방식으로 생성했는지를 나타내는 *SafeUUID*의 열거체입니다.

Added in version 3.7.

uuid 모듈은 다음의 함수들을 정의합니다.

uuid.**getnode**()

하드웨어 주소를 48bit 양의 정수로 가져옵니다. 최초 실행 시 별도의 프로그램으로 실행될 수 있고, 매우 느려질 수 있습니다. 하드웨어 주소를 얻기 위한 시도가 모두 실패하면, **RFC 4122**가 권장하는 대로 멀티캐스트 비트(첫 옥텟의 최하위 비트)가 1로 설정된 무작위 48bit 숫자를 선택합니다. "하드웨어 주소"는 네트워크 인터페이스의 MAC 주소를 뜻합니다. 여러 네트워크 인터페이스를 가진

머신에서 보편적으로 관리하는 MAC 주소(즉, 첫 번째 옥텟의 두 번째 최하위 비트가 *unset*인 경우)는 로컬에서 관리하는 MAC 주소보다 우선하지만, 순서를 보장하지는 않습니다.

버전 3.7에서 변경: 보편적으로 관리하는 MAC 주소는 로컬로 관리하는 MAC 주소보다 우선합니다. 전자는 주소가 전 세계적으로 고유함을 보장하지만, 후자는 그렇지 않기 때문입니다.

uuid.**uuid1**(*node=None*, *clock_seq=None*)

Generate a UUID from a host ID, sequence number, and the current time according to **RFC 9562, §5.1**.

When *node* is not specified, `getnode()` is used to obtain the hardware address as a 48-bit positive integer. When a sequence number *clock_seq* is not specified, a pseudo-random 14-bit positive integer is generated.

If *node* or *clock_seq* exceed their expected bit count, only their least significant bits are kept.

uuid.**uuid3**(*namespace*, *name*)

Generate a UUID based on the MD5 hash of a namespace identifier (which is a UUID) and a name (which is a `bytes` object or a string that will be encoded using UTF-8) according to **RFC 9562, §5.3**.

uuid.**uuid4**()

Generate a random UUID in a cryptographically-secure method according to **RFC 9562, §5.4**.

uuid.**uuid5**(*namespace*, *name*)

Generate a UUID based on the SHA-1 hash of a namespace identifier (which is a UUID) and a name (which is a `bytes` object or a string that will be encoded using UTF-8) according to **RFC 9562, §5.5**.

uuid.**uuid6**(*node=None*, *clock_seq=None*)

Generate a UUID from a sequence number and the current time according to **RFC 9562, §5.6**.

This is an alternative to `uuid1()` to improve database locality.

When *node* is not specified, `getnode()` is used to obtain the hardware address as a 48-bit positive integer. When a sequence number *clock_seq* is not specified, a pseudo-random 14-bit positive integer is generated.

If *node* or *clock_seq* exceed their expected bit count, only their least significant bits are kept.

Added in version 3.14.

uuid.**uuid7**()

Generate a time-based UUID according to **RFC 9562, §5.7**.

For portability across platforms lacking sub-millisecond precision, UUIDs produced by this function embed a 48-bit timestamp and use a 42-bit counter to guarantee monotonicity within a millisecond.

Added in version 3.14.

uuid.**uuid8**(*a=None*, *b=None*, *c=None*)

Generate a pseudo-random UUID according to **RFC 9562, §5.8**.

When specified, the parameters *a*, *b* and *c* are expected to be positive integers of 48, 12 and 62 bits respectively. If they exceed their expected bit count, only their least significant bits are kept; non-specified arguments are substituted for a pseudo-random integer of appropriate size.

By default, *a*, *b* and *c* are not generated by a cryptographically secure pseudo-random number generator (CSPRNG). Use `uuid4()` when a UUID needs to be used in a security-sensitive context.

Added in version 3.14.

`uuid` 모듈은 `uuid3()`과 `uuid5()`를 위한 아래의 이름 공간 식별자를 정의합니다.

uuid.**NAMESPACE_DNS**

이 이름 공간이 지정되면 *name* 문자열은 전체 도메인 이름(FQDN)입니다.

uuid.**NAMESPACE_URL**

이 이름 공간이 지정되면 *name* 문자열은 URL입니다.

uuid.**NAMESPACE_OID**

이 이름 공간이 지정되면 *name* 문자열은 ISO OID입니다.

uuid.**NAMESPACE_X500**

이 이름 공간이 지정되면 *name* 문자열은 DER이나 텍스트 출력 형식의 X.500 DN입니다.

uuid 모듈은 *variant* 어트리뷰트로 사용할 수 있는 값으로 다음의 상수를 정의합니다:

uuid.**RESERVED_NCS**

NCS 호환성을 위해 예약됨.

uuid.**RFC_4122**

Specifies the UUID layout given in **RFC 4122**. This constant is kept for backward compatibility even though **RFC 4122** has been superseded by **RFC 9562**.

uuid.**RESERVED_MICROSOFT**

마이크로소프트 호환성을 위해 예약됨.

uuid.**RESERVED_FUTURE**

추후 정의를 위해 예약됨.

The *uuid* module defines the special Nil and Max UUID values:

uuid.**NIL**

A special form of UUID that is specified to have all 128 bits set to zero according to **RFC 9562, §5.9**.

Added in version 3.14.

uuid.**MAX**

A special form of UUID that is specified to have all 128 bits set to one according to **RFC 9562, §5.10**.

Added in version 3.14.

더 보기

RFC 9562 - A Universally Unique IDentifier (UUID) URN Namespace
이 명세는 UUID를 위한 통합 자원 식별자 이름 공간, UUID의 내부 형식 및 UUID 생성 방법을 정의합니다.

22.15.1 명령줄 사용법

Added in version 3.12.

The *uuid* module can be executed as a script from the command line.

```
python -m uuid [-h] [-u {uuid1,uuid3,uuid4,uuid5,uuid6,uuid7,uuid8}] [-n␣
↪NAMESPACE] [-N NAME]
```

The following options are accepted:

-h, --help

Show the help message and exit.

-u <uuid>

--uuid <uuid>

Specify the function name to use to generate the uuid. By default *uuid4()* is used.

버전 3.14에서 변경: Allow generating UUID versions 6, 7 and 8.

-n <namespace>

--namespace <namespace>

The namespace is a `UUID`, or `@ns` where `ns` is a well-known predefined UUID addressed by namespace name. Such as `@dns`, `@url`, `@oid`, and `@x500`. Only required for *uuid3()* / *uuid5()* functions.

-N <name>

--name <name>

The name used as part of generating the uuid. Only required for `uuid3()` / `uuid5()` functions.

-C <num>

--count <num>

Generate *num* fresh UUIDs.

Added in version 3.14.

22.15.2 예제

`uuid` 모듈을 사용하는 전형적인 몇 가지 예제입니다:

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')

>>> # get the Nil UUID
>>> uuid.NIL
UUID('00000000-0000-0000-0000-000000000000')

>>> # get the Max UUID
>>> uuid.MAX
UUID('ffffffff-ffff-ffff-ffff-ffffffffffff')

>>> # same as UUIDv1 but with fields reordered to improve DB locality
>>> uuid.uuid6()
UUID('1f0799c0-98b9-62db-92c6-a0d365b91053')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> # get UUIDv7 creation (local) time as a timestamp in milliseconds
>>> u = uuid.uuid7()
>>> u.time
1743936859822

>>> # get UUIDv7 creation (local) time as a datetime object
>>> import datetime as dt
>>> dt.datetime.fromtimestamp(u.time / 1000)
datetime.datetime(...)

>>> # make a UUID with custom blocks
>>> uuid.uuid8(0x12345678, 0x9abcdef0, 0x11223344)
UUID('00001234-5678-8ef0-8000-000011223344')
```

22.15.3 명령줄 예제

Here are some examples of typical usage of the `uuid` command-line interface:

```
# generate a random UUID - by default uuid4() is used
$ python -m uuid

# generate a UUID using uuid1()
$ python -m uuid -u uuid1

# generate a UUID using uuid5
$ python -m uuid -u uuid5 -n @url -N example.com

# generate 42 random UUIDs
$ python -m uuid -C 42
```

22.16 socketserver — 네트워크 서버를 위한 프레임워크

소스 코드: Lib/socketserver.py

`socketserver` 모듈은 네트워크 서버 작성 작업을 단순화합니다.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

다음과 같은 네 가지 기본 구상 서버 클래스가 있습니다:

class socketserver.**TCPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

클라이언트와 서버 간에 연속적인 데이터 스트림을 제공하는 인터넷 TCP 프로토콜을 사용합니다. *bind_and_activate*가 참이면, 생성자는 자동으로 `server_bind()`와 `server_activate()`를 호출하려고 시도합니다. 다른 매개 변수는 `BaseServer` 베이스 클래스로 전달됩니다.

class socketserver.**UDPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

순서가 잘못되거나 전송 중 손실될 수 있는 이산적 정보 패킷인 데이터 그램을 사용합니다. 매개 변수는 `TCPServer`와 같습니다.

class socketserver.**UnixStreamServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

class socketserver.**UnixDatagramServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

TCP와 UDP 클래스와 비슷하지만, 유닉스 도메인 소켓을 사용하는 자주 사용되지 않는 클래스입니다; 유닉스 이외의 플랫폼에서는 사용할 수 없습니다. 매개 변수는 `TCPServer`와 같습니다.

이 네 가지 클래스는 동기적으로 (*synchronously*) 요청을 처리합니다; 다음 요청을 시작하기 전에 각 요청을 완료해야 합니다. 계산이 많이 필요하거나 클라이언트가 처리하는 속도가 느리도록 데이터를 많이 반환하기 때문에 각 요청을 완료하는 데 시간이 오래 걸리면 적합하지 않습니다. 해결책은 각 요청을 처리하기 위해 별도의 프로세스나 스레드를 만드는 것입니다; *ForkingMixIn*과 *ThreadingMixIn* 믹스인 클래스를 사용하여 비동기 동작을 지원할 수 있습니다.

서버를 만들려면 몇 가지 단계가 필요합니다. 먼저, *BaseRequestHandler* 클래스를 서브 클래싱하고 *handle()* 메서드를 재정의하여 요청 처리기 클래스를 만들어야 합니다; 이 메서드는 들어오는 요청을 처리합니다. 둘째, 서버 주소와 요청 처리기 클래스를 전달하여 서버 클래스 중 하나를 인스턴스 화해야 합니다. `with` 문에서 서버를 사용하는 것이 좋습니다. 그런 다음 서버 객체의 *handle_request()*나 *serve_forever()* 메서드를 호출하여 하나 이상의 요청을 처리합니다. 마지막으로, (`with` 문을 사용하지 않았다면) *server_close()*를 호출하여 소켓을 닫습니다.

스레드 연결 동작을 위해 *ThreadingMixIn*에서 상속할 때, 갑작스러운 종료 시 스레드 작동 방식을 명시적으로 선언해야 합니다. *ThreadingMixIn* 클래스는 서버가 스레드 종료를 기다려야 하는지를 가리키는 *daemon_threads* 어트리뷰트를 정의합니다. 스레드가 자율적으로 동작하게 하려면 플래그를 명시적으로 설정해야 합니다; 기본값은 *False*인데, *ThreadingMixIn*으로 만들어진 모든 스레드가 종료될 때까지 파이썬이 종료되지 않음을 뜻합니다.

서버 클래스는 사용하는 네트워크 프로토콜과 관계없이 같은 외부 메서드와 어트리뷰트를 갖습니다.

22.16.1 서버 생성 노트

상속 다이어그램에는 5개의 클래스가 있으며, 그중 4개는 4가지 유형의 동기 서버를 나타냅니다:

```
+------------+
| BaseServer |
+------------+
      |
      v
+-----------+         +------------------+
| TCPServer |------->| UnixStreamServer |
+-----------+         +------------------+
      |
      v
+-----------+         +--------------------+
| UDPServer |------->| UnixDatagramServer |
+-----------+         +--------------------+
```

*UnixDatagramServer*는 *UnixStreamServer*가 아니라 *UDPServer*에서 파생됨에 유의하십시오 — IP와 유닉스 서버의 유일한 차이점은 주소 패밀리입니다.

class socketserver.**ForkingMixIn**

class socketserver.**ThreadingMixIn**

이러한 믹스인 클래스를 사용하여 각 서버 유형의 포킹(forking)과 스레딩(threading) 버전을 만들 수 있습니다. 예를 들어, *ThreadingUDPServer*는 다음과 같이 만듭니다:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

*UDPServer*에 정의된 메서드를 재정의하므로, 믹스인 클래스가 먼저 옵니다. 다양한 어트리뷰트를 설정하면 하부 서버 메커니즘의 동작도 변경됩니다.

아래 언급된 *ForkingMixIn*과 Forking 클래스들은 *fork()*를 지원하는 POSIX 플랫폼에서만 사용할 수 있습니다.

block_on_close

*ForkingMixIn.server_close*는 *block_on_close* 어트리뷰트가 `False`인 경우를 제외하고 모든 자식 프로세스가 완료될 때까지 대기합니다.

*ThreadingMixIn.server_close*는 *block_on_close* 어트리뷰트가 `False`인 경우를 제외하고 모든 비 데몬 스레드가 완료될 때까지 대기합니다.

max_children

Specify how many child processes will exist to handle requests at a time for *ForkingMixIn*. If the limit is reached, new requests will wait until one child process has finished.

daemon_threads

For *ThreadingMixIn* use daemonic threads by setting *ThreadingMixIn.daemon_threads* to `True` to not wait until threads complete.

버전 3.7에서 변경: *ForkingMixIn.server_close*와 *ThreadingMixIn.server_close*는 이제 모든 자식 프로세스와 비 데몬 스레드가 완료될 때까지 대기합니다. 3.7 이전의 동작을 옵트인 하기 위해 새 *ForkingMixIn.block_on_close* 클래스 어트리뷰트를 추가합니다.

class socketserver.**ForkingTCPServer**

class socketserver.**ForkingUDPServer**

class socketserver.**ThreadingTCPServer**

class socketserver.**ThreadingUDPServer**

class socketserver.**ForkingUnixStreamServer**

class socketserver.**ForkingUnixDatagramServer**

class socketserver.**ThreadingUnixStreamServer**

class socketserver.**ThreadingUnixDatagramServer**

이 클래스들이 믹스인 클래스를 사용하여 미리 정의됩니다.

Added in version 3.12: The `ForkingUnixStreamServer` and `ForkingUnixDatagramServer` classes were added.

서비스를 구현하려면, *BaseRequestHandler*에서 클래스를 파생시키고 *handle()* 메서드를 재정의해야 합니다. 그런 다음 서버 클래스 중 하나와 여러분의 요청 처리기 클래스를 결합하여 다양한 버전의 서비스를 실행할 수 있습니다. 요청 처리기 클래스는 데이터 그램과 스트림 서비스에서 달라야 합니다. 처리기 서브 클래스 *StreamRequestHandler* 나 *DatagramRequestHandler*를 사용하여 이 차이를 숨길 수 있습니다.

물론, 여전히 머리를 사용해야 합니다! 예를 들어, 서비스가 다른 요청으로 수정될 수 있는 메모리상의 상태를 포함할 때 포킹 서버를 사용하는 것은 의미가 없습니다. 자식 프로세스에의 수정은 부모 프로세스에 유지된 초기 상태에 도달하여 각 자식에 전달되지 못하기 때문입니다. 이 경우, 스레딩 서버를 사용할 수 있지만, 아마도 공유 데이터의 무결성을 보호하기 위해 록을 사용해야 합니다.

반면, 모든 데이터가 외부(예를 들어, 파일 시스템)에 저장되는 HTTP 서버를 구축한다면, 동기 클래스는 하나의 요청이 처리되는 동안 실질적으로 서비스가 "듣지 못하게" 만듭니다 – 클라이언트가 요청한 모든 데이터를 받는 속도가 느리다면 매우 오랜 시간이 걸릴 수 있습니다. 이럴 때는 스레딩이나 포킹 서버가 적합합니다.

때에 따라, 요청의 일부를 동기적으로 처리하는 것이 좋지만, 요청 데이터에 따라 포크 된 자식에서 처리를 완료하는 것이 적절할 수 있습니다. 이는 동기 서버를 사용하고 요청 처리기 클래스 *handle()* 메서드에서 명시적 포크를 수행하여 구현할 수 있습니다.

스레드도 *fork()*도 지원하지 않는 (또는 이것들이 서비스에 너무 비싸거나 부적절한) 환경에서 여러 동시 요청을 처리하는 또 다른 방법은 부분적으로 완료된 요청의 명시적인 테이블을 유지하고 *selectors*를 사용하여 다음에 작업할 요청을 (또는 새로 들어온 요청을 처리할지를) 결정하는 것입니다. 이는 (스레드나 서브 프로세스를 사용할 수 없다면) 각 클라이언트가 오랫동안 연결될 수 있는 스트림 서비스에 특히 중요합니다.

22.16.2 서버 객체

class socketserver.**BaseServer**(*server_address*, *RequestHandlerClass*)

이것은 모듈에 있는 모든 서버 객체의 슈퍼 클래스입니다. 아래에 주어진 인터페이스를 정의하지만, 대부분의 메서드를 구현하지 않고, 서브 클래스에서 구현됩니다. 두 개의 매개 변수는 각각 *server_address*와 *RequestHandlerClass* 어트리뷰트에 저장됩니다.

fileno()

서버가 리스닝 중인 소켓의 정수 파일 디스크립터를 반환합니다. 이 함수는 가장 일반적으로 같은 프로세스에서 여러 서버를 모니터링할 수 있도록 *selectors*에 전달됩니다.

handle_request()

단일 요청을 처리합니다. 이 함수는 다음 메서드들을 차례로 호출합니다: *get_request()*, *verify_request()* 및 *process_request()*. 처리기 클래스의 사용자 제공 *handle()* 메서드에서 예외가 발생하면, 서버의 *handle_error()* 메서드가 호출됩니다. *timeout* 초 내에 요청이 수신되지 않으면, *handle_timeout()*이 호출되고 *handle_request()*는 반환합니다.

serve_forever(*poll_interval=0.5*)

명시적인 *shutdown()* 요청이 있을 때까지 요청을 처리합니다. *poll_interval* 초마다 shutdown을 확인합니다. *timeout* 어트리뷰트를 무시합니다. 또한 *service_actions()*를 호출하는데, 서브 클래스나 믹스인이 주어진 서비스에 특정한 동작을 제공하기 위해 사용할 수 있습니다. 예를 들어, *ForkingMixIn* 클래스는 *service_actions()*를 사용하여 좀비 자식 프로세스를 정리합니다.

버전 3.3에서 변경: `serve_forever` 메서드에 `service_actions` 호출을 추가했습니다.

service_actions()

serve_forever() 루프에서 호출됩니다. 이 메서드는 서브 클래스나 믹스인 클래스에서 재정의되어 정리 조치와 같은 지정된 서비스에 특정한 조치를 수행할 수 있습니다.

Added in version 3.3.

shutdown()

serve_forever() 루프가 정지하도록 하고 정지할 때까지 기다립니다. *serve_forever()*가 다른 스레드에서 실행되는 동안 *shutdown()*을 호출해야 합니다. 그렇지 않으면 교착 상태가 됩니다.

server_close()

서버를 정리합니다. 재정의될 수 있습니다.

address_family

서버 소켓이 속한 프로토콜 패밀리. 일반적인 예는 *socket.AF_INET*, *socket.AF_INET6* 및 *socket.AF_UNIX*입니다. IPv6 서버 클래스를 원하면 클래스 어트리뷰트 `address_family = AF_INET6`을 설정하여 이 모듈의 TCP 나 UDP 서버 클래스의 서브클래스를 만드세요.

RequestHandlerClass

사용자 제공 요청 처리기 클래스; 요청마다 이 클래스의 인스턴스가 만들어집니다.

server_address

서버가 리스닝 중인 주소. 주소 형식은 프로토콜 패밀리에 따라 다릅니다; 자세한 내용은 *socket* 모듈 설명서를 참조하십시오. 인터넷 프로토콜의 경우, 주소를 제공하는 문자열과 정수 포트 번호를 포함하는 튜플입니다: `('127.0.0.1', 80)`, 예를 들어.

socket

서버가 들어오는 요청을 리스닝 할 소켓 객체.

서버 클래스는 다음 클래스 변수를 지원합니다:

allow_reuse_address

서버가 주소를 재사용하도록 허락하는지 여부. 기본값은 *False*이며, 정책을 변경하기 위해 서브 클래스에서 설정할 수 있습니다.

request_queue_size

요청 큐의 크기. 단일 요청을 처리하는 데 시간이 오래 걸리면, 서버가 바쁠 때 도착한 요청은 최대 *request_queue_size* 요청까지 큐에 배치됩니다. 큐가 가득 차면, 클라이언트의 추가 요청은 "연결 거부(Connection denied)" 에러를 받게 됩니다. 기본값은 일반적으로 5이지만, 서브 클래스가 재정의할 수 있습니다.

socket_type

서버가 사용하는 소켓의 유형. *socket.SOCK_STREAM*과 *socket.SOCK_DGRAM*은 두 가지 흔한 값입니다.

timeout

초 단위의 시간제한 기간, 또는 시간제한이 필요하지 않으면 `None`. `handle_request()`가 timeout 기간 내에 들어오는 요청을 받지 못하면, `handle_timeout()` 메서드가 호출됩니다.

`TCPServer`와 같은 베이스 서버 클래스의 서브 클래스가 재정의할 수 있는 다양한 서버 메서드가 있습니다; 이러한 메서드는 서버 객체의 외부 사용자에게는 유용하지 않습니다.

finish_request(*request*, *client_address*)

`RequestHandlerClass`를 인스턴스화하고 `handle()` 메서드를 호출하여 실제로 요청을 처리합니다.

get_request()

소켓으로부터의 요청을 받아들이고, 클라이언트와 통신하는 데 사용될 새 소켓 객체와 클라이언트 주소를 포함하는 2-튜플을 반환해야 합니다.

handle_error(*request*, *client_address*)

`RequestHandlerClass` 인스턴스의 `handle()` 메서드에서 예외가 발생하면 이 함수가 호출됩니다. 기본 액션은 표준 에러로 트레이스백을 인쇄하고 추가 요청을 계속 처리하는 것입니다.

버전 3.6에서 변경: 이제 `Exception` 클래스에서 파생된 예외에 대해서만 호출됩니다.

handle_timeout()

이 함수는 `timeout` 어트리뷰트가 `None` 이외의 값으로 설정되고 요청이 수신되지 않은 채로 시간제한 기간이 지나면 호출됩니다. 포킹 서버에서의 기본 액션은 종료한 모든 자식 프로세스의 상태를 수집하는 것이고, 반면에 스레딩 서버에서는 이 메서드가 아무 작업도 수행하지 않습니다.

process_request(*request*, *client_address*)

`finish_request()`를 호출하여 `RequestHandlerClass` 의 인스턴스를 만듭니다. 원한다면, 이 함수는 요청을 처리하기 위해 새 프로세스나 스레드를 만들 수 있습니다; `ForkingMixIn`과 `ThreadingMixIn` 클래스가 그렇게 합니다.

server_activate()

서버를 활성화하기 위해 서버의 생성자가 호출합니다. TCP 서버의 기본 동작은 단지 서버의 소켓에 대해 `listen()`을 호출합니다. 재정의될 수 있습니다.

server_bind()

소켓을 원하는 주소에 바인딩하기 위해 서버의 생성자가 호출합니다. 재정의될 수 있습니다.

verify_request(*request*, *client_address*)

불리언 값을 반환해야 합니다; 값이 `True`이면, 요청이 처리되고, `False`이면, 요청이 거부됩니다. 서버에 대한 액세스 제어를 구현하기 위해 이 함수를 재정의할 수 있습니다. 기본 구현은 항상 `True`를 반환합니다.

버전 3.6에서 변경: 컨텍스트 관리자 프로토콜에 대한 지원이 추가되었습니다. 컨텍스트 관리자를 벗어나는 것은 `server_close()`를 호출하는 것과 동등합니다.

22.16.3 요청 처리기 객체

class socketserver.**BaseRequestHandler**

이것은 모든 요청 처리기 객체의 슈퍼 클래스입니다. 아래에 주어진 인터페이스를 정의합니다. 구상 요청 처리기 서브 클래스는 새 `handle()` 메서드를 정의해야 하며, 다른 메서드를 재정의할 수 있습니다. 각 요청에 대해 서브 클래스의 새 인스턴스가 만들어집니다.

setup()

필요한 초기화 액션을 수행하기 위해 `handle()` 메서드 전에 호출됩니다. 기본 구현은 아무것도 수행하지 않습니다.

handle()

이 함수는 요청을 서비스하는 데 필요한 모든 작업을 수행해야 합니다. 기본 구현은 아무것도 수행하지 않습니다. 몇 가지 인스턴스 어트리뷰트를 사용할 수 있습니다; 요청은 `request`로

제공됩니다; 클라이언트 주소는 *client_address*로 제공됩니다; 서버별 정보에 액세스해야 하는 경우를 위해 서버 인스턴스는 *server*로 제공됩니다.

*request*의 형은 데이터 그램과 스트림 서비스에서 다릅니다. 스트림 서비스의 경우, *request*는 소켓 객체입니다; 데이터 그램 서비스의 경우, *request*는 문자열과 소켓 쌍입니다.

finish()

필요한 정리 액션을 수행하기 위해 *handle()* 메서드 이후에 호출됩니다. 기본 구현은 아무것도 수행하지 않습니다. *setup()*에서 예외가 발생하면, 이 함수가 호출되지 않습니다.

request

The *new* *socket.socket* object to be used to communicate with the client.

client_address

Client address returned by *BaseServer.get_request()*.

server

BaseServer object used for handling the request.

class socketserver.**StreamRequestHandler**

class socketserver.**DatagramRequestHandler**

이 *BaseRequestHandler* 서브 클래스는 *setup()*과 *finish()* 메서드를 재정의하고, *rfile*과 *wfile* 어트리뷰트를 제공합니다.

rfile

읽어서 요청을 수신할 파일 객체. *io.BufferedIOBase* 읽기 가능 인터페이스를 지원합니다.

wfile

응답을 기록할 파일 객체. *io.BufferedIOBase* 쓰기 가능 인터페이스를 지원합니다.

버전 3.6에서 변경: *wfile* 도 *io.BufferedIOBase* 쓰기 가능 인터페이스를 지원합니다.

22.16.4 예

socketserver.TCPServer 예

이것은 서버 쪽입니다:

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    서버의 요청 처리기 클래스.

    서버 연결마다 한 번 인스턴스화되며, 클라이언트와의
    통신을 구현하기 위해 handle() 메서드를 재정의해야
    합니다.
    """

    def handle(self):
        # self.request는 클라이언트에 연결된 TCP 소켓입니다
        pieces = [b'']
        total = 0
        while b'\n' not in pieces[-1] and total < 10_000:
            pieces.append(self.request.recv(2000))
            total += len(pieces[-1])
        self.data = b''.join(pieces)
        print(f"Received from {self.client_address[0]}:")
        print(self.data.decode("utf-8"))
        # 단지 같은 데이터를 다시 보내지만, 대문자로 변환합니다
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        self.request.sendall(self.data.upper())
        # 반환 후에, 소켓은 닫힙니다.

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # 포트 9999에서 localhost에 바인딩하여 서버를 만듭니다
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # 서버를 활성화합니다; Ctrl-C를 사용하여 프로그램을 중단할 때까지 계속
        # 실행됩니다.
        server.serve_forever()
```

스트림(표준 파일 인터페이스를 제공하여 통신을 단순화하는 파일류 객체)을 사용하는 대체 요청 처리기 클래스:

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
        # self.rfile은 처리기에 의해 만들어진 파일류 객체입니다.
        # 이제 예를 들어 원시 recv() 호출 대신 readline()을 사용할 수 있습니다
        # 발신자의 악용을 방지하기 위해 10000 바이트로 제한합니다.
        self.data = self.rfile.readline(10000).rstrip()
        print(f"{self.client_address[0]} wrote:")
        print(self.data.decode("utf-8"))
        # 마찬가지로, self.wfile은 클라이언트로 다시 쓰는 데
        # 사용되는 파일류 객체입니다
        self.wfile.write(self.data.upper())
```

The difference is that the `readline()` call in the second handler will call `recv()` multiple times until it encounters a newline character, while the first handler had to use a `recv()` loop to accumulate data until a newline itself. If it had just used a single `recv()` without the loop it would just have returned what has been received so far from the client. TCP is stream based: data arrives in the order it was sent, but there no correlation between client `send()` or `sendall()` calls and the number of `recv()` calls on the server required to receive it.

이것은 클라이언트 쪽입니다:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# 소켓을 만듭니다 (SOCK_STREAM은 TCP 소켓을 의미합니다)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # 서버에 연결하고 데이터를 전송합니다
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data, "utf-8"))
    sock.sendall(b"\n")

    # 서버에서 데이터를 수신하고 종료합니다
    received = str(sock.recv(1024), "utf-8")

print("Sent:     ", data)
print("Received:", received)
```

예제의 결과는 다음과 같아야 합니다:

서버:

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

클라이언트:

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:     python is nice
Received: PYTHON IS NICE
```

socketserver.UDPServer 예

이것은 서버 쪽입니다:

```
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    self.request가 데이터와 클라이언트 소켓의 쌍으로 구성되고 연결이
    없기 때문에 sendto()를 통해 데이터를 다시 보낼 때 클라이언트 주소를
    명시적으로 제공해야 한다는 점을 제외하면, 이 클래스는 TCP 처리기 클래스와
    유사하게 작동합니다.
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print(f"{self.client_address[0]} wrote:")
        print(data)
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()
```

이것은 클라이언트 쪽입니다:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM은 UDP 소켓으로 사용할 소켓 유형입니다
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# 보시다시피, connect() 호출은 없습니다; UDP는 연결이 없습니다.
# 대신, sendto()를 통해 데이터를 수신자에게 직접 보냅니다.
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
print("Sent:     ", data)
print("Received:", received)
```

예제의 출력은 TCP 서버 예제와 정확히 같아야 합니다.

비동기 믹스인

비동기 처리기를 구축하려면, *ThreadingMixIn* 과 *ForkingMixIn* 클래스를 사용하십시오.

ThreadingMixIn 클래스의 예:

```
import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data), 'ascii')
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":
    # 포트 0은 임의의 미사용 포트를 선택하는 것을 의미합니다
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    with server:
        ip, port = server.server_address

        # 서버로 스레드를 시작합니다 -- 그런 다음 그 스레드가 요청마다 스레드를 하나씩 더
        # 시작합니다
        server_thread = threading.Thread(target=server.serve_forever)
        # 메인 스레드가 종료할 때 서버 스레드를 종료합니다
        server_thread.daemon = True
        server_thread.start()
        print("Server loop running in thread:", server_thread.name)

        client(ip, port, "Hello World 1")
        client(ip, port, "Hello World 2")
        client(ip, port, "Hello World 3")

        server.shutdown()
```

예제의 결과는 다음과 같아야 합니다:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

ForkingMixIn 클래스는 서버가 요청마다 새 프로세스를 생성한다는 점을 제외하고 같은 방식으로 사용됩니다. *fork()*를 지원하는 POSIX 플랫폼에서만 사용 가능합니다.

22.17 http.server — HTTP 서버

소스 코드: Lib/http/server.py

이 모듈은 HTTP 서버를 구현하기 위한 클래스를 정의합니다.

> **경고**
>
> 프로덕션에는 *http.server*가 권장되지 않습니다. 기초적인 보안 검사만 구현합니다.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

HTTPServer 클래스는 *socketserver.TCPServer* 서브 클래스입니다. HTTP 소켓을 만들고 리스닝하면서 요청을 처리기로 디스패치 합니다. 서버를 만들고 실행하는 코드는 다음과 같습니다:

```
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class http.server.**HTTPServer**(*server_address*, *RequestHandlerClass*)

이 클래스는 *TCPServer* 클래스를 기반으로 하고, 서버 주소를 `server_name`과 `server_port`라는 인스턴스 변수로 저장합니다. 처리기는 일반적으로 처리기의 `server` 인스턴스 변수를 통해 서버에 액세스 할 수 있습니다.

class http.server.**ThreadingHTTPServer**(*server_address*, *RequestHandlerClass*)

이 클래스는 HTTPServer와 동일하지만 *ThreadingMixIn*을 사용하여 요청을 처리하는 데 스레드를 사용합니다. *HTTPServer*가 무기한 대기하도록 만드는 소켓을 미리 여는 웹 브라우저를 처리하는 데 유용합니다.

Added in version 3.7.

class http.server.**HTTPSServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*, **, certfile*, *keyfile=None*, *password=None*, *alpn_protocols=None*)

Subclass of *HTTPServer* with a wrapped socket using the *ssl* module. If the *ssl* module is not available, instantiating a `HTTPSServer` object fails with a *RuntimeError*.

The *certfile* argument is the path to the SSL certificate chain file, and the *keyfile* is the path to file containing the private key.

A *password* can be specified for files protected and wrapped with PKCS#8, but beware that this could possibly expose hardcoded passwords in clear.

> **더 보기**
>
> See `ssl.SSLContext.load_cert_chain()` for additional information on the accepted values for *certfile*, *keyfile* and *password*.

When specified, the *alpn_protocols* argument must be a sequence of strings specifying the "Application-Layer Protocol Negotiation" (ALPN) protocols supported by the server. ALPN allows the server and the client to negotiate the application protocol during the TLS handshake.

By default, it is set to `["http/1.1"]`, meaning the server supports HTTP/1.1.

Added in version 3.14.

class `http.server.`**`ThreadingHTTPSServer`**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*, **, certfile*, *keyfile=None*, *password=None*, *alpn_protocols=None*)

This class is identical to `HTTPSServer` but uses threads to handle requests by inheriting from `ThreadingMixIn`. This is analogous to `ThreadingHTTPServer` only using `HTTPSServer`.

Added in version 3.14.

The `HTTPServer`, `ThreadingHTTPServer`, `HTTPSServer` and `ThreadingHTTPSServer` must be given a *RequestHandlerClass* on instantiation, of which this module provides three different variants:

class `http.server.`**`BaseHTTPRequestHandler`**(*request*, *client_address*, *server*)

이 클래스는 서버에 도착하는 HTTP 요청을 처리하는 데 사용됩니다. 그 자체로는, 실제 HTTP 요청에 응답할 수 없습니다; 각 요청 메서드(예를 들어 GET이나 POST)를 처리하려면 서브 클래스를 만들어야 합니다. `BaseHTTPRequestHandler`는 많은 클래스 및 인스턴스 변수와 서브 클래스가 사용할 메서드를 제공합니다.

처리기는 요청과 헤더를 구문 분석한 다음, 요청 유형에 특정한 메서드를 호출합니다. 메서드 이름은 요청으로부터 구성됩니다. 예를 들어, 요청 메서드 `SPAM`의 경우, `do_SPAM()` 메서드가 인자 없이 호출됩니다. 모든 관련 정보는 처리기의 인스턴스 변수에 저장됩니다. 서브 클래스는 `__init__()` 메서드를 대체하거나 확장할 필요가 없습니다.

`BaseHTTPRequestHandler`에는 다음과 같은 인스턴스 변수가 있습니다:

client_address

클라이언트 주소를 나타내는 `(host, port)` 형식의 튜플을 포함합니다.

server

서버 인스턴스를 포함합니다.

close_connection

`handle_one_request()`가 반환되기 전에 설정해야 하는 불리언으로, 다른 요청이 기대되는지, 또는 연결을 종료해야 하는지를 나타냅니다.

requestline

HTTP 요청 줄의 문자열 표현을 포함합니다. 종료 CRLF가 제거됩니다. 이 어트리뷰트는 `handle_one_request()`에서 설정해야 합니다. 유효한 요청 줄이 처리되지 않았으면, 빈 문자열로 설정해야 합니다.

command

명령(요청 유형)을 포함합니다. 예를 들어, `'GET'`.

path

Contains the request path. If query component of the URL is present, then `path` includes the query. Using the terminology of **RFC 3986**, `path` here includes `hier-part` and the `query`.

request_version

요청의 버전 문자열을 포함합니다. 예를 들어, `'HTTP/1.0'`.

headers

MessageClass 클래스 변수로 지정된 클래스의 인스턴스를 보유합니다. 이 인스턴스는 HTTP 요청의 헤더를 구문 분석하고 관리합니다. *http.client*의 *parse_headers()* 함수가 헤더를 구문 분석하는 데 사용되며 HTTP 요청이 유효한 **RFC 2822** 스타일 헤더를 제공할 것을 요구합니다.

rfile

선택적 입력 데이터의 시작부터 읽을 준비가 된 *io.BufferedIOBase* 입력 스트림.

wfile

클라이언트로 돌려줄 응답을 쓰기 위한 출력 스트림을 포함합니다. HTTP 클라이언트와의 성공적인 상호 운용을 위해서 이 스트림에 쓸 때 HTTP 프로토콜을 올바르게 준수해야 합니다.

버전 3.6에서 변경: 이것은 *io.BufferedIOBase* 스트림입니다.

*BaseHTTPRequestHandler*에는 다음과 같은 어트리뷰트가 있습니다:

server_version

서버 소프트웨어 버전을 지정합니다. 이것을 재정의하고 싶을 수 있습니다. 형식은 여러 공백으로 구분된 문자열이며, 각 문자열은 name[/version] 형식입니다. 예를 들어, `'BaseHTTP/0.2'`.

sys_version

version_string 메서드와 *server_version* 클래스 변수에서 사용할 수 있는 형식으로 파이썬 시스템 버전을 포함합니다. 예를 들어, `'Python/1.4'`.

error_message_format

클라이언트에 대한 에러 응답을 빌드하기 위해 *send_error()* 메서드에서 사용해야 하는 포맷 문자열을 지정합니다. 문자열은 기본적으로 *send_error()*에 전달된 상태 코드에 따라 *responses*의 변수로 채워집니다.

error_content_type

클라이언트로 전송되는 에러 응답의 Content-Type HTTP 헤더를 지정합니다. 기본값은 `'text/html'`입니다.

protocol_version

서버가 준수하는 HTTP 버전을 지정합니다. 향후 요청에 대한 서버의 통신 기능을 클라이언트에게 알리기 위해 응답으로 전송됩니다. `'HTTP/1.1'`로 설정되면, 서버는 HTTP 지속적 연결(persistent connections)을 허용합니다; 그러나, 이때 서버는 반드시 클라이언트에 대한 모든 응답에 (*send_header()*를 사용해서) 정확한 `Content-Length` 헤더를 포함해야 합니다. 이전 버전과의 호환성을 위해, 기본 설정은 `'HTTP/1.0'`입니다.

MessageClass

HTTP 헤더를 구문 분석할 *email.message.Message*와 유사한 클래스를 지정합니다. 일반적으로, 이는 재정의되지 않으며, 기본값은 *http.client.HTTPMessage*입니다.

responses

이 어트리뷰트에는 에러 코드 정수에서 짧고 긴 메시지를 포함하는 두 요소 튜플로의 매핑이 포함됩니다. 예를 들어, `{code: (shortmessage, longmessage)}`. *shortmessage*는 일반적으로 에러 응답에서 *message* 키로 사용되고, *longmessage*는 *explain* 키로 사용됩니다. *send_response_only()*와 *send_error()* 메서드에서 사용됩니다.

BaseHTTPRequestHandler 인스턴스에는 다음과 같은 메서드가 있습니다:

handle()

들어오는 HTTP 요청을 처리하기 위해 *handle_one_request()*를 한 번 (또는, 지속적 연결이 활성화되었으면, 여러 번) 호출합니다. 재정의할 필요는 없습니다; 대신 적절한 `do_*()` 메서드를 구현하십시오.

handle_one_request()

이 메서드는 요청을 구문 분석하여 적절한 `do_*()` 메서드로 디스패치 합니다. 재정의할 필요는 없습니다.

handle_expect_100()

HTTP/1.1 호환 서버가 `Expect: 100-continue` 요청 헤더를 수신하면 `100 Continue`로 응답한 후 `200 OK` 헤더가 뒤따릅니다. 클라이언트가 계속하는 것을 서버가 원하지 않으면 에러를 발생시키기 위해 이 메서드를 재정의할 수 있습니다. 예를 들어 서버는 `417 Expectation Failed`를 응답 헤더로 보내고 `return False` 할 수 있습니다.

Added in version 3.2.

send_error(*code*, *message=None*, *explain=None*)

클라이언트에게 완전한 에러 응답을 보내고 로깅 합니다. 숫자 *code*는 HTTP 에러 코드를 지정하며, *message*는 선택적인 사람이 읽을 수 있는 에러에 대한 간단한 설명입니다. *explain* 인자는 에러에 대한 자세한 정보를 제공하는 데 사용될 수 있습니다; `error_message_format` 어트리뷰트를 사용하여 포맷되고 전체 헤더 집합 뒤에 응답 바디로 보냅니다. `responses` 어트리뷰트는 값이 제공되지 않을 때 사용될 *message*와 *explain*의 기본값을 담고 있습니다; 알 수 없는 코드의 경우 둘 다 기본값은 문자열 `???`입니다. 메서드가 HEAD이거나 응답 코드가 `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified` 중 하나면 바디는 비어 있게 됩니다.

버전 3.4에서 변경: 에러 응답에는 Content-Length 헤더가 포함됩니다. *explain* 인자를 추가했습니다.

send_response(*code*, *message=None*)

헤더 버퍼에 응답 헤더를 추가하고 받아들인 요청을 로깅 합니다. HTTP 응답 줄이 내부 버퍼에 기록되고, *Server*와 *Date* 헤더가 뒤따릅니다. 이 두 헤더의 값은 각각 `version_string()`과 `date_time_string()` 메서드에서 취합니다. 서버가 `send_header()` 메서드를 사용하여 다른 헤더를 보내려고 하지 않는다면, `send_response()` 다음에 `end_headers()` 호출이 있어야 합니다.

버전 3.3에서 변경: 헤더는 내부 버퍼에 저장되며 `end_headers()`를 명시적으로 호출해야 합니다.

send_header(*keyword*, *value*)

`end_headers()`나 `flush_headers()`가 호출될 때 출력 스트림에 기록될 내부 버퍼에 HTTP 헤더를 추가합니다. *keyword*는 헤더 키워드를 지정하고, *value*는 값을 지정해야 합니다. send_header 호출이 완료된 후, 작업을 완료하려면 반드시 `end_headers()`를 호출해야 함에 유의하십시오.

버전 3.2에서 변경: 헤더는 내부 버퍼에 저장됩니다.

send_response_only(*code*, *message=None*)

응답 헤더만 보내는데, 서버가 `100 Continue` 응답을 클라이언트로 전송할 목적으로 사용됩니다. 헤더는 버퍼링 되지 않고 출력 스트림으로 직접 전송합니다. *message*를 지정하지 않으면, 응답 *code*에 해당하는 HTTP 메시지가 전송됩니다.

Added in version 3.2.

end_headers()

(응답에서 HTTP 헤더의 끝을 나타내는) 빈 줄을 헤더 버퍼에 추가하고 `flush_headers()`를 호출합니다.

버전 3.2에서 변경: 버퍼링 된 헤더는 출력 스트림에 기록됩니다.

flush_headers()

마지막으로 헤더를 출력 스트림으로 보내고 내부 헤더 버퍼를 플러시 합니다.

Added in version 3.3.

log_request(*code='-'*, *size='-'*)

받아들인 (성공적인) 요청을 로깅 합니다. *code*는 응답과 관련된 숫자 HTTP 코드를 지정해야 합니다. 응답의 크기가 있으면, *size* 매개 변수로 전달되어야 합니다.

log_error(...)

요청을 이행할 수 없을 때 에러를 로깅 합니다. 기본적으로, 메시지를 `log_message()`에 전달하므로, 같은 인자(*format*과 추가 값)를 취합니다.

log_message(*format*, ...)

sys.stderr에 임의의 메시지를 로깅 합니다. 이것은 일반적으로 사용자 지정 에러 로깅 메커니즘을 만들기 위해 재정의됩니다. *format* 인자는 표준 printf 스타일 포맷 문자열이며, *log_message()*에 대한 추가 인자는 포매팅의 입력으로 적용됩니다. 클라이언트 ip 주소와 현재 날짜 및 시간은 로깅 되는 모든 메시지 앞에 붙습니다.

version_string()

서버 소프트웨어의 버전 문자열을 반환합니다. 이것은 *server_version*과 *sys_version* 어트리뷰트의 조합입니다.

date_time_string(*timestamp=None*)

timestamp(None이거나 *time.time()*이 반환한 형식이어야 합니다)로 지정된 날짜와 시간을 메시지 헤더용으로 포맷하여 반환합니다. *timestamp*를 생략하면, 현재 날짜와 시간이 사용됩니다.

결과는 'Sun, 06 Nov 1994 08:49:37 GMT'와 같은 모습입니다.

log_date_time_string()

로깅용으로 포맷한 현재 날짜와 시간을 반환합니다.

address_string()

클라이언트 주소를 반환합니다.

버전 3.3에서 변경: 이전에는, 이름 조회가 수행되었습니다. 이름 결정(name resolution) 지연을 피하고자, 이제 항상 IP 주소를 반환합니다.

class http.server.**SimpleHTTPRequestHandler**(*request*, *client_address*, *server*, *directory=None*)

이 클래스는 디렉터리 구조를 HTTP 요청에 직접 매핑하여 디렉터리 *directory*와 그 이하의, 또는 *directory*가 제공되지 않으면 현재 디렉터리의 파일을 제공합니다.

버전 3.7에서 변경: Added the *directory* parameter.

버전 3.9에서 변경: *directory* 매개 변수는 경로류 객체를 받아들입니다.

요청 구문 분석과 같은 많은 작업이 베이스 클래스 *BaseHTTPRequestHandler*에 의해 수행됩니다. 이 클래스는 *do_GET()*과 *do_HEAD()* 함수를 구현합니다.

다음은 *SimpleHTTPRequestHandler*의 클래스 수준 어트리뷰트로 정의됩니다:

server_version

이것은 "SimpleHTTP/" + __version__이며, 여기서 __version__은 모듈 수준에서 정의됩니다.

extensions_map

접미사를 MIME 형식으로 매핑하는 딕셔너리. 기본 시스템 매핑에 대한 사용자 정의 재정의를 포함합니다. 매핑은 대소 문자를 구분 없이 사용되므로, 소문자 키만 포함해야 합니다.

버전 3.9에서 변경: 이 딕셔너리는 더는 기본 시스템 매핑으로 채워지지 않고, 재정의 만 포함합니다.

SimpleHTTPRequestHandler 클래스는 다음 메서드를 정의합니다:

do_HEAD()

이 메서드는 'HEAD' 요청 유형을 제공합니다: 동등한 GET 요청에 대해 전송할 헤더를 전송합니다. 가능한 헤더에 대한 더 완전한 설명은 *do_GET()* 메서드를 참조하십시오.

do_GET()

요청을 현재 작업 디렉터리에 상대적인 경로로 해석하여 요청은 로컬 파일에 매핑됩니다.

요청이 디렉터리에 매핑되었으면, 디렉터리는 index.html이나 index.htm(이 순서대로) 파일을 검사합니다. 발견되면, 파일 내용이 반환됩니다; 그렇지 않으면 list_directory() 메서드를 호출하여 디렉터리 목록이 생성됩니다. 이 메서드는 *os.listdir()*을 사용하여 디렉터리를 스캔하고, *listdir()*이 실패하면 404 에러 응답을 반환합니다.

요청이 파일에 매핑되었으면, 파일을 엽니다. 요청된 파일을 열 때 발생하는 *OSError* 예외는 404, 'File not found' 에러로 매핑됩니다. 요청에 'If-Modified-Since' 헤더가 있고, 이

시간 이후 파일이 수정되지 않았으면, 304, 'Not Modified' 응답이 전송됩니다. 그렇지 않으면, 콘텐츠 유형은 guess_type() 메서드를 호출하여 추측되며, 이 메서드는 *extensions_map* 변수를 사용합니다. 그런 다음 파일 내용이 반환됩니다.

추측된 콘텐츠 유형의 'Content-type:' 헤더가 출력되고, 파일 크기가 담긴 'Content-Length:' 헤더와 파일 수정 시간이 담긴 'Last-Modified:' 헤더가 뒤따릅니다.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output.

사용 예로는, Lib/http/server.py 모듈에서 test 함수 구현을 참조하십시오.

버전 3.7에서 변경: 'If-Modified-Since' 헤더 지원.

SimpleHTTPRequestHandler 클래스는 현재 디렉터리를 기준으로 파일을 제공하는 매우 기본적인 웹 서버를 만들기 위해 다음과 같은 방식으로 사용될 수 있습니다:

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

SimpleHTTPRequestHandler can also be subclassed to enhance behavior, such as using different index file names by overriding the class attribute index_pages.

class http.server.**CGIHTTPRequestHandler**(*request*, *client_address*, *server*)

이 클래스는 현재 디렉터리와 그 아래에 있는 파일이나 CGI 스크립트의 출력을 제공하는 데 사용됩니다. HTTP 계층 구조를 로컬 디렉터리 구조에 매핑하는 것은 *SimpleHTTPRequestHandler*와 정확히 같음에 유의하십시오.

> **참고**
>
> *CGIHTTPRequestHandler* 클래스가 실행하는 CGI 스크립트는 리디렉션(HTTP 코드 302)을 실행할 수 없습니다, CGI 스크립트를 실행하기 전에 코드 200(스크립트 출력이 이어집니다)이 전송되기 때문입니다. 이것은 상태 코드를 선점합니다.

클래스는 CGI 스크립트라고 생각되면 파일로 제공하는 대신 CGI 스크립트를 실행합니다. 디렉터리 기반 CGI만 사용됩니다 — 다른 일반적인 서버 구성은 특수한 확장자를 CGI 스크립트를 나타내는 것으로 취급하는 것입니다.

요청이 cgi_directories 경로 아래로 이어지면, 파일을 제공하는 대신 CGI 스크립트를 실행하고 출력을 제공하도록 do_GET() 과 do_HEAD() 함수가 수정되었습니다.

*CGIHTTPRequestHandler*는 다음 데이터 멤버를 정의합니다:

cgi_directories

기본값은 ['/cgi-bin', '/htbin'] 이며 CGI 스크립트를 포함하는 것으로 취급할 디렉터리를 기술합니다.

*CGIHTTPRequestHandler*는 다음 메서드를 정의합니다:

do_POST()

이 메서드는 'POST' 요청 유형을 제공하며, CGI 스크립트에만 허용됩니다. CGI 이외의 url에 POST를 시도할 때 에러 501, "Can only POST to CGI scripts" 가 출력됩니다.

보안상의 이유로, CGI 스크립트는 nobody 사용자의 UID로 실행됨에 유의하십시오. CGI 스크립트 문제는 에러 403으로 변환됩니다.

Deprecated since version 3.13, will be removed in version 3.15: *CGIHTTPRequestHandler* is being removed in 3.15. CGI has not been considered a good way to do things for well over a decade. This code has been unmaintained for a while now and sees very little practical use. Retaining it could lead to further *security considerations*.

22.17.1 Command-line interface

*http.server*는 인터프리터의 `-m` 스위치를 사용하여 직접 호출할 수도 있습니다. 다음 예제는 현재 디렉터리를 기준으로 파일을 제공하는 방법을 보여줍니다:

```
python -m http.server [OPTIONS] [port]
```

The following options are accepted:

port

The server listens to port 8000 by default. The default can be overridden by passing the desired port number as an argument:

```
python -m http.server 9000
```

-b, --bind <address>

바인딩할 특정 주소를 지정합니다. IPv4와 IPv6 주소가 모두 지원됩니다. 기본적으로, 서버는 모든 인터페이스에 자신을 바인드 합니다. 예를 들어, 다음 명령은 서버가 localhost에만 바인드 하도록 합니다:

```
python -m http.server --bind 127.0.0.1
```

Added in version 3.4.

버전 3.8에서 변경: Support IPv6 in the `--bind` option.

-d, --directory <dir>

파일을 제공할 디렉터리를 지정합니다. 기본적으로 서버는 현재 디렉터리를 사용합니다. 예를 들어, 다음 명령은 특정 디렉터리를 사용합니다:

```
python -m http.server --directory /tmp/
```

Added in version 3.7.

-p, --protocol <version>

Specifies the HTTP version to which the server is conformant. By default, the server is conformant to HTTP/1.0. For example, the following command runs an HTTP/1.1 conformant server:

```
python -m http.server --protocol HTTP/1.1
```

Added in version 3.11.

--cgi

`--cgi` 옵션을 전달하여 명령 줄에서 *CGIHTTPRequestHandler*를 사용할 수 있습니다:

```
python -m http.server --cgi
```

Deprecated since version 3.13, will be removed in version 3.15: *http.server* command line `--cgi` support is being removed because *CGIHTTPRequestHandler* is being removed.

경고

CGIHTTPRequestHandler and the `--cgi` command-line option are not intended for use by untrusted clients and may be vulnerable to exploitation. Always use within a secure environment.

--tls-cert

Specifies a TLS certificate chain for HTTPS connections:

```
python -m http.server --tls-cert fullchain.pem
```

Added in version 3.14.

--tls-key

Specifies a private key file for HTTPS connections.

This option requires `--tls-cert` to be specified.

Added in version 3.14.

--tls-password-file

Specifies the password file for password-protected private keys:

```
python -m http.server \
       --tls-cert cert.pem \
       --tls-key key.pem \
       --tls-password-file password.txt
```

This option requires *–tls-cert*‘ to be specified.

Added in version 3.14.

22.17.2 Security considerations

`SimpleHTTPRequestHandler` will follow symbolic links when handling requests, this makes it possible for files outside of the specified directory to be served.

Earlier versions of Python did not scrub control characters from the log messages emitted to stderr from `python -m http.server` or the default `BaseHTTPRequestHandler.log_message` implementation. This could allow remote clients connecting to your server to send nefarious control codes to your terminal.

버전 3.12에서 변경: Control characters are scrubbed in stderr logs.

22.18 `http.cookies` — HTTP 상태 관리

소스 코드: Lib/http/cookies.py

`http.cookies` 모듈은 HTTP 상태 관리 메커니즘인 쿠키의 개념을 추상화하는 클래스를 정의합니다. 그것은 단순한 문자열 전용 쿠키를 지원하고, 동시에 직렬화 가능한 데이터형을 쿠키값으로 갖는 데 필요한 추상화를 제공합니다.

이 모듈은 예전에는 **RFC 2109**와 **RFC 2068** 명세에서 설명된 구문 분석 규칙을 엄격하게 적용했습니다. 그 이후로 MSIE 3.0x가 이 명세에 명시된 문자 규칙을 따르지 않았으며; 쿠키 처리와 관련하여 오늘날의 많은 브라우저와 서버 또한 구문 분석 규칙을 완화했다는 사실이 발견되었습니다. 결과적으로, 이제 이 모듈은 이전보다 약간 덜 엄격한 구문 분석 규칙을 사용합니다.

문자 집합, `string.ascii_letters`, `string.digits` 및 `!#$%&'*+-.^_`|~:`는 이 모듈이 쿠키 이름(`key`)에 허용한 유효한 문자 집합을 나타냅니다.

버전 3.3에서 변경: ‘:’ 를 유효한 쿠키 이름 문자로 허용합니다.

> **참고**
>
> 잘못된 쿠키가 발견되면, `CookieError`가 발생하므로, 쿠키 데이터가 브라우저에서 제공되면 항상 잘못된 데이터일 가능성에 대비하고 구문 분석할 때 `CookieError`를 잡아야 합니다.

exception http.cookies.**CookieError**

RFC 2109 위반으로 인해 실패하는 예외: 잘못된 어트리뷰트, 잘못된 *Set-Cookie* 헤더 등

class http.cookies.**BaseCookie**([*input*])

이 클래스는 키가 문자열이고 값이 *Morsel* 인스턴스인 딕셔너리 형 객체입니다. 키에 값을 설정하면, 값이 먼저 키와 값이 포함된 *Morsel*로 변환됩니다.

*input*이 주어지면, *load()* 메서드로 전달됩니다.

class http.cookies.**SimpleCookie**([*input*])

이 클래스는 *BaseCookie*에서 파생되며 *value_decode()*와 *value_encode()*를 재정의합니다. SimpleCookie는 문자열 쿠키값을 지원합니다. 값을 설정할 때, SimpleCookie는 내장 *str()*을 호출하여 값을 문자열로 변환합니다. HTTP에서 수신된 값은 문자열로 유지됩니다.

더 보기

모듈 ***http.cookiejar***

웹 클라이언트용 HTTP 쿠키 처리. *http.cookiejar*와 *http.cookies* 모듈 간의 의존성은 없습니다.

RFC 2109 - HTTP State Management Mechanism (HTTP 상태 관리 메커니즘)

이것은 이 모듈이 구현한 상태 관리 명세입니다.

22.18.1 쿠키 객체

BaseCookie.**value_decode**(*val*)

문자열 표현으로부터 튜플 (real_value, coded_value)를 반환합니다. real_value는 모든 형이 될 수 있습니다. 이 메서드는 *BaseCookie*에서는 아무런 디코딩을 하지 않습니다 — 재정의할 수 있도록 존재합니다.

BaseCookie.**value_encode**(*val*)

튜플 (real_value, coded_value)를 반환합니다. *val*은 모든 형이 될 수 있지만, coded_value는 항상 문자열로 변환됩니다. 이 메서드는 *BaseCookie*에서는 아무런 인코딩을 하지 않습니다 — 재정의할 수 있도록 존재합니다.

일반적으로, *value_encode()*와 *value_decode()*는 *value_decode* 범위에서 역연산입니다.

BaseCookie.**output**(*attrs=None, header='Set-Cookie:', sep='\r\n'*)

HTTP 헤더로서 송신하기 적절한 문자열 표현을 반환합니다. *attrs*와 *header*는 각 *Morsel*의 *output()* 메서드로 전송됩니다. *sep*는 헤더를 함께 결합하는 데 사용되며, 기본적으로 '\r\n' (CRLF) 조합입니다.

BaseCookie.**js_output**(*attrs=None*)

자바스크립트를 지원하는 브라우저에서 실행될 때 HTTP 헤더가 전송된 것처럼 작동하는, 삽입 가능한 자바스크립트 코드 조각을 반환합니다.

*attrs*의 의미는 *output()*과 같습니다.

BaseCookie.**load**(*rawdata*)

*rawdata*가 문자열이면, HTTP_COOKIE로 구문 분석하고 거기에 있는 값을 *Morsel*로 추가합니다. 딕셔너리면, 다음과 동등합니다:

```
for k, v in rawdata.items():
    cookie[k] = v
```

22.18.2 Morsel 객체

class http.cookies.**Morsel**

RFC 2109 어트리뷰트를 포함하는 키/값 쌍을 추상화합니다.

Morsel은 딕셔너리류 객체이며, 키의 집합은 상수입니다 — 다음과 같은 유효한 RFC 2109 어트리뷰트입니다:

expires
path
comment
domain
max-age
secure
version
httponly
samesite
partitioned

httponly 어트리뷰트는 쿠키가 HTTP 요청으로만 전송되고 자바스크립트를 통해 액세스할 수 없도록 지정합니다. 이것은 교차 사이트 스크립팅의 일부 형식을 방지하기 위한 것입니다.

The attribute *samesite* controls when the browser sends the cookie with cross-site requests. This helps to mitigate CSRF attacks. Valid values are "Strict" (only sent with same-site requests), "Lax" (sent with same-site requests and top-level navigations), and "None" (sent with same-site and cross-site requests). When using "None", the "secure" attribute must also be set, as required by modern browsers.

The attribute *partitioned* indicates to user agents that these cross-site cookies *should* only be available in the same top-level context that the cookie was first set in. For this to be accepted by the user agent, you **must** also set `Secure`.

In addition, it is recommended to use the `__Host` prefix when setting partitioned cookies to make them bound to the hostname and not the registrable domain. Read CHIPS (Cookies Having Independent Partitioned State) for full details and examples.

키는 대소 문자를 구분하지 않으며 기본값은 `''`입니다.

버전 3.5에서 변경: `__eq__()`는 이제 *key*와 *value*를 고려합니다.

버전 3.7에서 변경: 어트리뷰트 *key*, *value* 및 *coded_value*는 읽기 전용입니다. 설정하려면 *set()*을 사용하십시오.

버전 3.8에서 변경: *samesite* 어트리뷰트에 대한 지원이 추가되었습니다.

버전 3.14에서 변경: Added support for the *partitioned* attribute.

Morsel.**value**

쿠키의 값.

Morsel.**coded_value**

쿠키의 인코딩된 값 — 이것을 전송해야 합니다.

Morsel.**key**

쿠키의 이름.

Morsel.**set**(*key*, *value*, *coded_value*)

key, *value* 및 *coded_value* 어트리뷰트를 설정합니다.

Morsel.**isReservedKey**(*K*)

*K*가 *Morsel*의 키 집합의 구성원인지를 판단합니다.

Morsel.**output**(*attrs=None*, *header='Set-Cookie:'*)

HTTP 헤더로 보내기에 적합한, Morsel의 문자열 표현을 반환합니다. 기본적으로, *attrs*가 주어지지 않는 한, 모든 어트리뷰트가 포함됩니다. 주어지면 사용할 어트리뷰트 리스트여야 합니다. *header*는 기본적으로 `"Set-Cookie:"`입니다.

`Morsel.js_output(attrs=None)`

자바스크립트를 지원하는 브라우저에서 실행될 때, HTTP 헤더가 전송된 것처럼 작동하는, 삽입 가능한 자바스크립트 코드 조각을 반환합니다.

*attrs*의 의미는 `output()`과 같습니다.

`Morsel.OutputString(attrs=None)`

둘러싸는 HTTP나 자바스크립트 없이 Morsel을 표현하는 문자열을 반환합니다.

*attrs*의 의미는 `output()`과 같습니다.

`Morsel.update(values)`

Morsel 딕셔너리의 값을 딕셔너리 *values*의 값으로 갱신합니다. *values* 딕셔너리의 키 중 하나라도 유효한 **RFC 2109** 어트리뷰트가 아니면 에러를 발생시킵니다.

버전 3.5에서 변경: 유효하지 않은 키에 대해서 에러가 발생합니다.

`Morsel.copy(value)`

Morsel 객체의 얕은 복사본을 반환합니다.

버전 3.5에서 변경: 딕셔너리 대신 Morsel 객체를 반환합니다.

`Morsel.setdefault(key, value=None)`

key가 유효한 **RFC 2109** 어트리뷰트가 아니면 에러를 발생시킵니다. 그렇지 않으면, `dict.setdefault()`와 같이 동작합니다.

22.18.3 예제

다음 예제는 `http.cookies` 모듈을 사용하는 방법을 보여줍니다.

```
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C) # HTTP 헤더를 생성합니다
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output()) # 마찬가지입니다
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = cookies.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # 문자열에서 로드합니다 (HTTP 헤더)
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\\012;";')
>>> print(C)
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;"
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Set-Cookie: oreo=doublestuff; Path=/
>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = cookies.SimpleCookie()
>>> C["number"] = 7 # C["number"] = str(7) 과 동등합니다
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven
```

22.19 http.cookiejar — HTTP 클라이언트를 위한 쿠키 처리

소스 코드: Lib/http/cookiejar.py

---

http.cookiejar 모듈은 HTTP 쿠키의 자동 처리를 위한 클래스를 정의합니다. 웹 서버의 HTTP 응답으로 클라이언트 시스템에 설정된 후, 이후의 HTTP 요청에서 서버로 반환되는 작은 데이터 조각 – 쿠키(*cookies*) – 을 요구하는 웹 사이트에 액세스하는 데 유용합니다.

일반 Netscape 쿠키 프로토콜과 **RFC 2965**로 정의된 프로토콜이 모두 처리됩니다. RFC 2965 처리는 기본적으로 꺼져있습니다. **RFC 2109** 쿠키는 Netscape 쿠키로 구문 분석된 후 이후에 적용되는 '정책' 에 따라 Netscape나 RFC 2965 쿠키로 처리됩니다. 인터넷 쿠키의 대부분은 Netscape 쿠키임에 유의하십시오. *http.cookiejar*는 RFC 2965에 도입된 `max-age`와 `port` 쿠키 어트리뷰트를 처리하는 것을 포함하여, 시장 표준 Netscape 쿠키 프로토콜(원래 Netscape 명세에서 설정한 것과 꽤 다릅니다)을 따르려고 합니다.

> **참고**
>
> *Set-Cookie*와 *Set-Cookie2* 헤더에서 발견되는 다양한 이름 붙은 파라미터(예를 들어, `domain`과 `expires`)는 통상적으로 어트리뷰트(*attributes*)로 지칭됩니다. 파이썬 어트리뷰트와 구별하기 위해, 이 모듈의 설명서는 쿠키 어트리뷰트(*cookie-attribute*)라는 용어를 대신 사용합니다.

모듈은 다음 예외를 정의합니다:

exception http.cookiejar.**LoadError**

FileCookieJar 인스턴스는 파일에서 쿠키를 로드하지 못하면 이 예외를 발생시킵니다. *LoadError*는 *OSError*의 서브 클래스입니다.

버전 3.3에서 변경: 이전에는 *LoadError*가 *IOError*의 서브 형이었습니다. 이제는 *OSError*의 별칭입니다.

다음과 같은 클래스가 제공됩니다:

class http.cookiejar.**CookieJar**(*policy=None*)

*policy*는 *CookiePolicy* 인터페이스를 구현하는 객체입니다.

CookieJar 클래스는 HTTP 쿠키를 저장합니다. HTTP 요청에서 쿠키를 추출하고, HTTP 응답으로 반환합니다. 필요하면 *CookieJar* 인스턴스는 포함된 쿠키를 자동으로 만료합니다. 서브 클래스는 파일이나 데이터베이스에서 쿠키를 저장하고 검색하는 역할도 합니다.

class http.cookiejar.**FileCookieJar**(*filename=None, delayload=None, policy=None*)

*policy*는 `CookiePolicy` 인터페이스를 구현하는 객체입니다. 다른 인자에 대해서는, 해당 어트리뷰트의 설명서를 참조하십시오.

디스크의 파일에서 쿠키를 로드하고 아마도 파일에 쿠키를 저장할 수 있는 `CookieJar`. `load()`나 `revert()` 메서드가 호출될 때까지 이름이 지정된 파일에서 쿠키가 로드되지 **않습니다**. 이 클래스의 서브 클래스는 *FileCookieJar 서브 클래스와 웹 브라우저와의 협력* 섹션에 설명되어 있습니다.

This should not be initialized directly – use its subclasses below instead.

버전 3.8에서 변경: filename 매개 변수는 경로류 객체를 지원합니다.

class http.cookiejar.**CookiePolicy**

이 클래스는 각 쿠키가 서버에서 수락되고 서버로 반환되어야 하는지를 결정하는 역할을 맡습니다.

class http.cookiejar.**DefaultCookiePolicy**(*blocked_domains=None, allowed_domains=None, netscape=True, rfc2965=False, rfc2109_as_netscape=None, hide_cookie2=False, strict_domain=False, strict_rfc2965_unverifiable=True, strict_ns_unverifiable=False, strict_ns_domain=DefaultCookiePolicy.DomainLiberal, strict_ns_set_initial_dollar=False, strict_ns_set_path=False, secure_protocols=('https', 'wss')*)

생성자 인자는 키워드 인자로만 전달해야 합니다. *blocked_domains*는 쿠키를 수락하지도 쿠키를 반환하지도 않을 도메인 이름의 시퀀스입니다. *allowed_domains*가 `None`이 아니면, 이것이 쿠키를 수락하고 반환하는 유일한 도메인의 시퀀스입니다. *secure_protocols*는 보안 쿠키를 추가 할 수 있는 프로토콜의 시퀀스입니다. 기본적으로 *https*와 *wss*(보안 웹 소켓)는 보안 프로토콜로 간주합니다. 다른 모든 인자는, `CookiePolicy`와 `DefaultCookiePolicy` 객체에 대한 설명서를 참조하십시오.

`DefaultCookiePolicy`는 Netscape와 **RFC 2965** 쿠키를 위한 표준 수락/거부 규칙을 구현합니다. 기본적으로, **RFC 2109** 쿠키(즉 version 쿠키 어트리뷰트가 1인 `Set-Cookie` 헤더로 수신된 쿠키)는 RFC 2965 규칙에 따라 처리됩니다. 그러나, RFC 2965 처리가 꺼져 있거나 `rfc2109_as_netscape`가 `True`이면, `Cookie` 인스턴스의 `version` 어트리뷰트를 0으로 설정하여, `CookieJar` 인스턴스에서 RFC 2109 쿠키를 Netscape 쿠키로 '다운 그레이드' 합니다. `DefaultCookiePolicy`는 정책을 미세 조정할 수 있는 매개 변수도 제공합니다.

class http.cookiejar.**Cookie**

이 클래스는 Netscape, **RFC 2109** 및 **RFC 2965** 쿠키를 나타냅니다. `http.cookiejar` 사용자가 스스로 `Cookie` 인스턴스를 만들 것으로 기대하지 않습니다. 대신, 필요하면, `CookieJar` 인스턴스에서 `make_cookies()`를 호출하십시오.

더 보기

모듈 ***urllib.request***
자동 쿠키 처리가 지원되는 URL 열기.

모듈 ***http.cookies***
HTTP 쿠키 클래스, 주로 서버 측 코드에 유용합니다. `http.cookiejar`와 `http.cookies` 모듈은 서로 의존하지 않습니다.

https://curl.se/rfc/cookie_spec.html
원래 Netscape 쿠키 프로토콜의 명세. 이것이 여전히 지배적인 프로토콜이지만, 모든 주요 브라우저(및 `http.cookiejar`)에 의해 구현된 'Netscape 쿠키 프로토콜' 은 사라져가는 `cookie_spec.html`에서 스케치 된 것과의 유사성을 담고 있을 뿐입니다.

RFC 2109 - HTTP 상태 관리 메커니즘
RFC 2965로 개정되었습니다. version=1 인 `Set-Cookie`를 사용합니다.

RFC 2965 - HTTP 상태 관리 메커니즘
버그가 수정된 Netscape 프로토콜. `Set-Cookie` 대신 `Set-Cookie2`를 사용합니다. 널리 사용되지 않습니다.

https://kristol.org/cookie/errata.html
완료되지 않은 **RFC 2965**의 정오표.

RFC 2964 - HTTP 상태 관리 사용

22.19.1 CookieJar와 FileCookieJar 객체

CookieJar 객체는 포함된 *Cookie* 객체를 이터레이트 하기 위한 이터레이터 프로토콜을 지원합니다.

*CookieJar*에는 다음과 같은 메서드가 있습니다:

CookieJar.**add_cookie_header**(*request*)

*request*에 올바른 *Cookie* 헤더를 추가합니다.

정책이 허용하면 (즉, *CookieJar*의 *CookiePolicy* 인스턴스의 `rfc2965`와 `hide_cookie2` 어트리뷰트가 각각 참과 거짓이면), *Cookie2* 헤더도 적절한 경우 추가됩니다.

request 객체 (일반적으로 *urllib.request.Request* 인스턴스) 는 *urllib.request*에 설명되어 있듯이 `get_full_url()`, `has_header()`, `get_header()`, `header_items()`, `add_unredirected_header()` 및 어트리뷰트 `host`, `type`, `unverifiable`, `origin_req_host`를 지원해야 합니다.

버전 3.3에서 변경: *request* 객체에는 `origin_req_host` 어트리뷰트가 필요합니다. 폐지된 메서드 `get_origin_req_host()`에 대한 의존성이 제거되었습니다.

CookieJar.**extract_cookies**(*response*, *request*)

HTTP *response*에서 쿠키를 추출하여 정책이 허용하면 *CookieJar*에 저장합니다.

*CookieJar*는 *response* 인자에서 수락할 수 있는 *Set-Cookie*와 *Set-Cookie2* 헤더를 찾고, 쿠키를 적절하게 저장합니다 (*CookiePolicy.set_ok()* 메서드의 승인에 따라).

response 객체 (일반적으로 *urllib.request.urlopen()* 호출의 결과, 또는 유사한 것)는 *email.message.Message* 인스턴스를 반환하는 `info()` 메서드를 지원해야 합니다.

request 객체 (일반적으로 *urllib.request.Request* 인스턴스)는 *urllib.request*에 설명되어 있듯이 메서드 `get_full_url()` 과 어트리뷰트 `host`, `unverifiable` 및 `origin_req_host`를 지원해야 합니다. request는 쿠키를 설정할 수 있는지 확인하는 것뿐만 아니라 쿠키 어트리뷰트의 기본값을 설정하는 데 사용됩니다.

버전 3.3에서 변경: *request* 객체에는 `origin_req_host` 어트리뷰트가 필요합니다. 폐지된 메서드 `get_origin_req_host()`에 대한 의존성이 제거되었습니다.

CookieJar.**set_policy**(*policy*)

사용할 *CookiePolicy* 인스턴스를 설정합니다.

CookieJar.**make_cookies**(*response*, *request*)

response 객체에서 추출한 *Cookie* 객체의 시퀀스를 반환합니다.

*response*와 *request* 인자에 필요한 인터페이스는 *extract_cookies()* 설명서를 참조하십시오.

CookieJar.**set_cookie_if_ok**(*cookie*, *request*)

정책이 그래도 좋다고 한다면 *Cookie*를 설정합니다.

CookieJar.**set_cookie**(*cookie*)

설정할 수 있는지 정책을 확인하지 않고, *Cookie*를 설정합니다.

CookieJar.**clear**([*domain*[, *path*[, *name*]]])

일부 쿠키를 지웁니다.

인자 없이 호출되면, 모든 쿠키를 지웁니다. 단일 인자가 제공되면, 해당 *domain*에 속하는 쿠키만 제거됩니다. 두 개의 인자가 제공되면, 지정된 *domain*과 URL *path*에 속하는 쿠키가 제거됩니다. 세 개의 인자가 제공되면, 지정된 *domain*, *path* 및 *name*을 갖는 쿠키가 제거됩니다.

일치하는 쿠키가 없으면 *KeyError*를 발생시킵니다.

CookieJar.**clear_session_cookies**()

모든 세션 쿠키를 폐기합니다.

실제 discard 어트리뷰트를 갖는 모든 쿠키를 폐기합니다 (보통 max-age나 expires 쿠키 어트리뷰트가 없기 때문에, 혹은 명시적인 discard 쿠키 어트리뷰트). 대화식 브라우저의 경우, 세션의 끝은 일반적으로 브라우저 창을 닫는 것에 해당합니다.

save() 메서드는 참값의 *ignore_discard* 인자를 전달하여 다르게 요청하지 않는 한 세션 쿠키를 저장하지 않음에 유의하십시오.

*FileCookieJar*는 다음과 같은 추가 메서드를 구현합니다:

FileCookieJar.**save**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

쿠키를 파일에 저장합니다.

이 베이스 클래스는 *NotImplementedError*를 발생시킵니다. 서브 클래스는 이 메서드를 구현하지 않은 채로 둘 수 있습니다.

*filename*은 쿠키를 저장할 파일의 이름입니다. *filename*을 지정하지 않으면, self.filename이 사용됩니다 (이것의 기본값은 생성자에게 전달된 값입니다, 있다면); self.filename이 *None*이면 *ValueError*가 발생합니다.

ignore_discard: 폐기되는 것으로 설정된 쿠키도 저장합니다. *ignore_expires*: 만료된 쿠키도 저장합니다

파일이 이미 존재하면, 파일을 덮어써서, 파일에 포함된 모든 쿠키를 지웁니다. 저장된 쿠키는 나중에 *load()*나 *revert()* 메서드를 사용하여 복원할 수 있습니다.

FileCookieJar.**load**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

파일에서 쿠키를 로드합니다.

새로 로드한 쿠키가 덮어쓰지 않는 한 오래된 쿠키는 유지됩니다.

인자는 *save()*와 같습니다.

명명된 파일은 클래스가 이해하는 형식이어야 하고, 그렇지 않으면 *LoadError*가 발생합니다. 또한, 예를 들어 파일이 존재하지 않으면, *OSError*가 발생할 수 있습니다.

버전 3.3에서 변경: *IOError*가 발생했었지만, 이제는 *OSError*의 별칭입니다.

FileCookieJar.**revert**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)

모든 쿠키를 지우고 저장된 파일에서 쿠키를 다시 로드합니다.

*revert()*는 *load()*와 같은 예외를 발생시킬 수 있습니다. 실패가 일어나면, 객체 상태는 변경되지 않습니다.

FileCookieJar 인스턴스에는 다음과 같은 공용 어트리뷰트가 있습니다:

FileCookieJar.**filename**

쿠키를 보관할 기본 파일의 파일명. 이 어트리뷰트는 대입할 수 있습니다.

FileCookieJar.**delayload**

참이면, 디스크에서 쿠키를 천천히(lazily) 로드합니다. 이 어트리뷰트는 대입하지 않아야 합니다. (디스크의 쿠키가 변경되지 않는 한) 성능에만 영향을 미칠 뿐 동작을 바꾸지는 않기 때문에, 이것은 힌트일 뿐입니다. *CookieJar* 객체는 이를 무시할 수 있습니다. 표준 라이브러리에 포함된 *FileCookieJar* 클래스는 아무것도 쿠키를 천천히 로드하지 않습니다.

22.19.2 FileCookieJar 서브 클래스와 웹 브라우저와의 협력

다음 *CookieJar* 서브 클래스는 읽기와 쓰기를 위해 제공됩니다.

class http.cookiejar.**MozillaCookieJar**(*filename=None*, *delayload=None*, *policy=None*)

Mozilla cookies.txt 파일 형식(curl, Lynx 및 Netscape 브라우저에서도 사용됩니다)으로 디스크에서 쿠키를 로드하고 저장할 수 있는 *FileCookieJar*.

> **참고**
>
> RFC 2965 쿠키에 대한 정보와, `port` 같은 최신이나 비표준 쿠키 어트리뷰트에 대한 정보가 손실됩니다.

> **경고**
>
> 손실/손상이 불편한 쿠키가 있다면 저장하기 전에 쿠키를 백업하십시오 (로드/저장 왕복으로 인해 파일이 약간 변경될 수 있는 미묘한 부분이 있습니다).

또한 Mozilla가 실행되는 동안 저장된 쿠키는 Mozilla에 의해 방해를 받음에 유의하십시오.

class `http.cookiejar.`**`LWPCookieJar`**(*filename=None, delayload=None, policy=None*)

libwww-perl 라이브러리의 `Set-Cookie3` 파일 형식과 호환되는 형식으로 쿠키를 디스크에서 로드하고 저장할 수 있는 `FileCookieJar`. 사람이 읽을 수 있는 파일에 쿠키를 저장하려는 경우에 편리합니다.

버전 3.8에서 변경: filename 매개 변수는 경로류 객체를 지원합니다.

22.19.3 CookiePolicy 객체

`CookiePolicy` 인터페이스를 구현하는 객체에는 다음과 같은 메서드가 있습니다:

`CookiePolicy.`**`set_ok`**(*cookie, request*)

서버에서 쿠키를 수락해야 하는지를 나타내는 불리언 값을 반환합니다.

*cookie*는 `Cookie` 인스턴스입니다. *request*는 `CookieJar.extract_cookies()` 설명서에서 정의한 인터페이스를 구현하는 객체입니다.

`CookiePolicy.`**`return_ok`**(*cookie, request*)

쿠키를 서버로 반환해야 하는지를 나타내는 불리언 값을 반환합니다.

*cookie*는 `Cookie` 인스턴스입니다. *request*는 `CookieJar.add_cookie_header()` 설명서에서 정의한 인터페이스를 구현하는 객체입니다.

`CookiePolicy.`**`domain_return_ok`**(*domain, request*)

주어진 쿠키 도메인(domain)에서, 쿠키를 반환하지 않아야 하면 `False`를 반환합니다.

이 메서드는 최적화입니다. 특정 도메인을 가진 모든 쿠키를 검사할 필요(많은 파일을 읽는 것을 수반합니다)를 제거합니다. `domain_return_ok()`와 `path_return_ok()`에서 참을 반환하면 모든 작업이 `return_ok()`로 넘어갑니다.

쿠키 도메인에 대해 `domain_return_ok()`가 참을 반환하면, 쿠키 경로에 대해 `path_return_ok()`가 호출됩니다. 그렇지 않으면, 해당 쿠키 도메인에 대해 `path_return_ok()`와 `return_ok()`가 호출되지 않습니다. `path_return_ok()`가 참을 반환하면, `return_ok()`가 `Cookie` 객체 자체로 호출되어 전체 검사를 수행합니다. 그렇지 않으면, 해당 쿠키 경로에 대해 `return_ok()`가 호출되지 않습니다.

`domain_return_ok()`는 *request* 도메인뿐만 아니라 모든 *cookie* 도메인에 대해 호출됨에 유의하십시오. 예를 들어, 요청 도메인이 `"www.example.com"`이면 함수는 `".example.com"`과 `"www.example.com"` 모두에 대해 호출될 수 있습니다. `path_return_ok()`도 마찬가지입니다.

request 인자는 `return_ok()`에 대해 설명된 대로입니다.

`CookiePolicy.`**`path_return_ok`**(*path, request*)

주어진 쿠키 경로에 대해, 쿠키를 반환하지 않아야 하면 `False`를 반환합니다.

`domain_return_ok()` 설명서를 참조하십시오.

위의 메서드를 구현하는 것 외에도, `CookiePolicy` 인터페이스의 구현은 어떤 프로토콜을 어떻게 사용해야 하는지를 나타내는 다음 어트리뷰트도 제공해야 합니다. 이러한 모든 어트리뷰트는 대입할 수 있습니다.

CookiePolicy.**netscape**

Netscape 프로토콜을 구현합니다.

CookiePolicy.**rfc2965**

RFC 2965 프로토콜을 구현합니다.

CookiePolicy.**hide_cookie2**

요청에 *Cookie2* 헤더를 추가하지 않습니다 (이 헤더가 있으면 서버에게 우리가 RFC 2965 쿠키를 이해하고 있음을 알립니다).

CookiePolicy 클래스를 정의하는 가장 유용한 방법은 *DefaultCookiePolicy* 에서 서브 클래싱하고 위의 메서드 중 일부나 전부를 재정의하는 것입니다. *CookiePolicy* 자체는 모든 쿠키를 설정하고 수신하도록 허용하는 '널 정책' 으로 사용될 수 있습니다 (이 정책이 그리 유용하지는 않을 것입니다).

22.19.4 DefaultCookiePolicy 객체

쿠키 수락과 반환에 대한 표준 규칙을 구현합니다.

RFC 2965와 Netscape 쿠키를 모두 다룹니다. RFC 2965 처리는 기본적으로 꺼져있습니다.

자체 정책을 제공하는 가장 쉬운 방법은 이 클래스를 재정의하고 자체 검사를 추가하기 전에 재정의된 구현에서 해당 메서드를 호출하는 것입니다:

```
import http.cookiejar
class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

CookiePolicy 인터페이스를 구현하는 데 필요한 기능 외에도, 이 클래스를 사용하면 도메인이 쿠키를 설정하고 수신하는 것을 차단하고 허락할 수 있도록 합니다. 또한 다소 느슨한 Netscape 프로토콜 규칙을 약간 강화할 수 있는 몇 가지 엄격성 스위치가 있습니다 (일부 양성 쿠키를 차단하는 대가를 치르면서).

도메인 차단 목록과 수락 목록이 제공됩니다 (둘 다 기본적으로 꺼져있습니다). 차단 목록에 없고 수락 목록에 있는(수락 목록이 활성화되었다면) 도메인만 쿠키 설정과 반환에 참여합니다. *blocked_domains* 생성자 인자와 `blocked_domains()` 및 `set_blocked_domains()` 메서드 (그리고 *allowed_domains*를 위한 해당 인자와 메서드)를 사용하십시오. 수락 목록을 설정하면, *None*으로 설정하여 수락 목록을 다시 끌 수 있습니다.

점으로 시작하지 않는 차단 또는 수락 목록에 있는 도메인은 일치할 쿠키 도메인과 같아야 합니다. 예를 들어, `"example.com"`은 차단 목록 항목 `"example.com"`과 일치하지만, `"www.example.com"`은 일치하지 않습니다. 점으로 시작하는 도메인은 더 구체적인 도메인과도 일치합니다. 예를 들어, `"www.example.com"`과 `"www.coyote.example.com"`은 모두 `".example.com"`과 일치합니다 (하지만 `"example.com"` 자체는 일치하지 않습니다). IP 주소는 예외이며, 정확히 일치해야 합니다. 예를 들어, blocked_domains에 `"192.168.1.2"`와 `".168.1.2"`가 포함되어 있으면, 192.168.1.2는 차단되지만, 193.168.1.2는 차단되지 않습니다.

*DefaultCookiePolicy*는 다음과 같은 추가 메서드를 구현합니다:

DefaultCookiePolicy.**blocked_domains**()

차단된 도메인 시퀀스를 반환합니다 (튜플로).

DefaultCookiePolicy.**set_blocked_domains**(*blocked_domains*)

차단된 도메인 시퀀스를 설정합니다.

DefaultCookiePolicy.**is_blocked**(*domain*)

*domain*이 쿠키 설정이나 수신에 대한 차단 목록에 있으면 `True`를 반환합니다.

DefaultCookiePolicy.**allowed_domains**()

None, 또는 수락 도메인 시퀀스를 반환합니다 (튜플).

DefaultCookiePolicy.**set_allowed_domains**(*allowed_domains*)

수락 도메인 시퀀스, 또는 *None*을 설정합니다.

DefaultCookiePolicy.**is_not_allowed**(*domain*)

*domain*이 쿠키 설정이나 수신에 대한 수락 목록에 있으면 `True`를 반환합니다.

DefaultCookiePolicy 인스턴스에는 다음과 같은 어트리뷰트가 있습니다. 이 어트리뷰트들은 모두 같은 이름의 생성자 인자로 초기화되며, 모두 대입할 수 있습니다.

DefaultCookiePolicy.**rfc2109_as_netscape**

참이면, *CookieJar* 인스턴스가 *Cookie* 인스턴스의 version 쿠키 어트리뷰트를 0으로 설정하여 **RFC 2109** 쿠키 (즉, *Set-Cookie* 헤더에서 수신된 version 쿠키 어트리뷰트가 1인 쿠키)를 Netscape 쿠키로 다운 그레이드하도록 요청합니다. 기본값은 *None*입니다, 이 경우 **RFC 2965** 처리가 꺼졌을 때만 RFC 2109 쿠키가 다운 그레이드됩니다. 따라서, RFC 2109 쿠키는 기본적으로 다운 그레이드됩니다.

일반 엄격성 스위치:

DefaultCookiePolicy.**strict_domain**

사이트가 `.co.uk`, `.gov.uk`, `.co.nz` 등의 국가 코드 최상위 도메인으로 2개의 구성 요소 도메인을 설정하는 것을 수락하지 않습니다. 이것은 완벽하지는 않으며 작동하지 않을 수도 있습니다!

RFC 2965 프로토콜 엄격성 스위치:

DefaultCookiePolicy.**strict_rfc2965_unverifiable**

확인할 수 없는 트랜잭션(unverifiable transactions)에 대한 **RFC 2965** 규칙을 따릅니다 (일반적으로 확인할 수 없는 트랜잭션은 다른 사이트에서 호스팅 되는 이미지에 대한 리디렉션이나 요청으로 인한 결과입니다). 이것이 거짓이면, 확인 가능성에 기초하여 쿠키가 차단되지 않습니다.

넷스케이프 프로토콜 엄격성 스위치:

DefaultCookiePolicy.**strict_ns_unverifiable**

Netscape 쿠키에도 확인할 수 없는 트랜잭션에 대한 **RFC 2965** 규칙을 적용합니다.

DefaultCookiePolicy.**strict_ns_domain**

Netscape 쿠키에 대한 도메인 일치 규칙이 얼마나 엄격한지를 나타내는 플래그. 허용 가능한 값은 아래를 참조하십시오.

DefaultCookiePolicy.**strict_ns_set_initial_dollar**

이름이 `'$'`로 시작하는 Set-Cookie: 헤더의 쿠키를 무시합니다.

DefaultCookiePolicy.**strict_ns_set_path**

경로가 요청 URI와 경로 일치하지 않는 쿠키 설정을 수락하지 않습니다.

strict_ns_domain is a collection of flags. Its value is constructed by or-ing together (for example, `DomainStrictNoDots|DomainStrictNonDomain` means both flags are set).

DefaultCookiePolicy.**DomainStrictNoDots**

쿠키를 설정할 때, '호스트 접두사' 에는 점이 없어야 합니다 (예를 들어 `www.foo`에 점이 있어서, `www.foo.bar.com`이 `.bar.com`에 대한 쿠키를 설정할 수 없습니다).

DefaultCookiePolicy.**DomainStrictNonDomain**

`domain` 쿠키 어트리뷰트를 명시적으로 지정하지 않은 쿠키는 쿠키를 설정한 도메인과 같은 도메인으로만 반환될 수 있습니다 (예를 들어 `spam.example.com`으로는 `domain` 쿠키 어트리뷰트가 없는 `example.com`의 쿠키를 반환하지 않습니다).

DefaultCookiePolicy.**DomainRFC2965Match**

쿠키를 설정할 때, 전체 **RFC 2965** 도메인 일치가 필요합니다.

편의를 위해 다음 어트리뷰트가 제공되며, 위 플래그의 가장 유용한 조합입니다:

DefaultCookiePolicy.**DomainLiberal**

0과 동등합니다 (즉, 위의 모든 Netscape 도메인 엄격성 플래그가 꺼집니다).

DefaultCookiePolicy.**DomainStrict**

DomainStrictNoDots|DomainStrictNonDomain 과 동등합니다.

22.19.5 Cookie 객체

Cookie 인스턴스는 다양한 쿠키 표준에 지정된 표준 쿠키 어트리뷰트에 대략 상응하는 파이썬 어트리뷰트를 갖습니다. 기본값을 지정하기 위한 복잡한 규칙이 있고, max-age와 expires 쿠키 어트리뷰트가 동등한 정보를 포함하고, **RFC 2109** 쿠키가 버전 1에서 버전 0 (Netscape) 쿠키로 *http.cookiejar*에 의해 '다운그레이드' 될 수 있기 때문에 대응은 일대일이 아닙니다.

CookiePolicy 메서드에서의 드문 경우를 제외하고 이러한 어트리뷰트에 대입할 필요는 없습니다. 이 클래스는 내부 일관성을 강요하지 않기 때문에, 그렇게 한다면 무엇을 하고 있는지 알아야 합니다.

Cookie.**version**

정수나 *None*. Netscape 쿠키는 *version* 0을 갖습니다. **RFC 2965**와 **RFC 2109** 쿠키는 1의 version 쿠키 어트리뷰트를 갖습니다. 그러나, *http.cookiejar*는 RFC 2109 쿠키를 Netscape 쿠키로 '다운그레이드' 할 수 있으며, 이 경우 *version*은 0입니다.

Cookie.**name**

쿠키 이름 (문자열).

Cookie.**value**

쿠키값 (문자열), 또는 *None*.

Cookie.**port**

포트나 포트 집합을 나타내는 문자열 (예를 들어 '80' 이나 '80,8080'), 또는 *None*.

Cookie.**domain**

쿠키 도메인 (문자열).

Cookie.**path**

쿠키 경로 (문자열, 예를 들어 '/acme/rocket_launchers').

Cookie.**secure**

쿠키가 보안 연결을 통해서만 반환되어야 하면 True.

Cookie.**expires**

에포크 이후의 초 단위 정수 만료 날짜, 또는 *None*. *is_expired()* 메서드도 참조하십시오.

Cookie.**discard**

세션 쿠키이면 True.

Cookie.**comment**

이 쿠키의 기능을 설명하는 서버의 문자열 주석, 또는 *None*.

Cookie.**comment_url**

이 쿠키의 기능을 설명하는 서버의 주석에 대한 URL 링크, 또는 *None*.

Cookie.**rfc2109**

이 쿠키가 **RFC 2109** 쿠키로 수신되었으면 (즉 쿠키가 *Set-Cookie* 헤더로 도착하고, 해당 헤더의 Version 쿠키 어트리뷰트 값이 1인 경우) True. *http.cookiejar*가 RFC 2109 쿠키를 Netscape 쿠키로 '다운그레이드' 할 수 있기 때문에 이 어트리뷰트가 제공됩니다. 이 경우 *version*은 0입니다.

Cookie.**port_specified**

서버가 (*Set-Cookie* / *Set-Cookie2* 헤더에서) 포트나 포트 집합을 명시적으로 지정했으면 True.

Cookie.**domain_specified**

서버가 도메인을 명시적으로 지정했으면 True.

Cookie.**domain_initial_dot**

서버에서 명시적으로 지정한 도메인이 점('.')으로 시작하면 True.

쿠키에는 비표준 쿠키 어트리뷰트가 추가로 있을 수 있습니다. 다음 메서드를 사용하여 액세스 할 수 있습니다:

Cookie.**has_nonstandard_attr**(*name*)

쿠키에 지정된 이름의 쿠키 어트리뷰트가 있으면 True를 반환합니다.

Cookie.**get_nonstandard_attr**(*name*, *default=None*)

쿠키에 지정된 이름의 쿠키 어트리뷰트가 있으면, 그 값을 반환합니다. 그렇지 않으면 *default*를 반환합니다.

Cookie.**set_nonstandard_attr**(*name*, *value*)

지정된 이름의 쿠키 어트리뷰트의 값을 설정합니다.

Cookie 클래스는 다음 메서드도 정의합니다:

Cookie.**is_expired**(*now=None*)

쿠키에 대해 서버가 만료해야 한다고 요청한 시간이 지났으면 True. *now*가 제공되면 (에포크 이후의 초로), 쿠키가 지정된 시간에 만료되는지를 반환합니다.

22.19.6 예

첫 번째 예는 *http.cookiejar*의 가장 일반적인 사용법을 보여줍니다:

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

이 예는 Netscape, Mozilla 또는 Lynx 쿠키를 사용하여 URL을 여는 방법을 보여줍니다 (쿠키 파일의 위치에 대해서는 유닉스/Netscape 규칙을 가정합니다):

```
import os, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

다음 예는 *DefaultCookiePolicy* 의 사용법을 보여줍니다. **RFC 2965** 쿠키를 켜고, Netscape 쿠키를 설정하고 반환할 때 도메인에 대해 더욱 엄격하며, 일부 도메인이 쿠키를 설정하거나 쿠키를 반환하지 못하도록 차단합니다:

```
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

22.20 xmlrpc — XMLRPC 서버와 클라이언트 모듈

XML-RPC는 HTTP를 트랜스포트로 사용해서 전달되는 XML을 사용하는 원격 프로시저 호출 방법입니다. 이를 통해, 클라이언트는 원격 서버(서버는 URI로 지정됩니다)의 매개 변수가 있는 메서드를 호출하고 구조화된 데이터를 받을 수 있습니다.

xmlrpc는 XML-RPC를 구현하는 서버와 클라이언트 모듈을 모아둔 패키지입니다. 모듈은 다음과 같습니다:

- *xmlrpc.client*
- *xmlrpc.server*

22.21 xmlrpc.client — XML-RPC 클라이언트 액세스

소스 코드: Lib/xmlrpc/client.py

XML-RPC는 HTTP(S)를 통해 전달된 XML을 트랜스포트로 사용하는 원격 프로시저 호출(Remote Procedure Call) 방법입니다. 이를 통해, 클라이언트는 원격 서버에서 매개 변수를 사용하여 메서드를 호출하고 (서버는 URI로 이름이 지정됩니다) 구조화된 데이터를 돌려받을 수 있습니다. 이 모듈은 XML-RPC 클라이언트 코드 작성을 지원합니다; 적합한 파이썬 객체와 전송 회선 상의 XML 간 변환의 모든 세부 사항을 처리합니다.

> **경고**
>
> The *xmlrpc.client* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data, see *XML security*.

버전 3.5에서 변경: HTTPS URI의 경우, *xmlrpc.client*는 이제 기본적으로 필요한 모든 인증서와 호스트명 확인을 수행합니다.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

class xmlrpc.client.**ServerProxy**(*uri, transport=None, encoding=None, verbose=False, allow_none=False, use_datetime=False, use_builtin_types=False, *, headers=(), context=None*)

ServerProxy 인스턴스는 원격 XML-RPC 서버와의 통신을 관리하는 객체입니다. 필수적인 첫 번째 인자는 URI(Uniform Resource Indicator)이며 일반적으로 서버의 URL입니다. 선택적인 두 번째 인자는 트랜스포트 팩토리 인스턴스입니다; 기본적으로 https: URL의 경우는 내부 SafeTransport 인스턴스이고 그렇지 않으면 내부 HTTP Transport 인스턴스입니다. 선택적 세 번째 인자는 인코딩이며, 기본적으로 UTF-8입니다. 선택적 네 번째 인자는 디버깅 플래그입니다.

그 뒤에 오는 매개 변수들은 반환된 프락시 인스턴스 사용을 제어합니다. *allow_none*이 참이면, 파이썬 상수 None이 XML로 변환됩니다; 기본 동작은 None이 *TypeError*를 발생시키는 것입니다. 이것은 XML-RPC 명세에 일반적으로 사용되는 확장이지만, 모든 클라이언트와 서버에서 지원되는 것은 아닙니다; 설명은 http://ontosys.com/xml-rpc/extensions.php를 참조하십시오. *use_builtin_types* 플래그를 사용하여 날짜/시간 값을 *datetime.datetime* 객체로 표현하고 바이너리 데이터를 *bytes* 객체로 표현할 수 있습니다; 이 플래그는 기본적으로 거짓입니다. *datetime.datetime*, *bytes* 및 *bytearray* 객체는 호출로 전달될 수 있습니다. *headers* 매개 변수는 각 요청과 함께 보낼 선택적 HTTP 헤더의 시퀀스이며, 헤더 이름과 값을 나타내는 2-튜플의 시퀀스로 표현됩니다. (예를 들어 [('Header-Name', 'value')]). HTTPS URL이 제공되면, *context*는 *ssl.SSLContext*일 수 있으며, 하부 HTTPS 연결의 SSL 설정을 구성합니다. 사용되지 않는 *use_datetime* 플래그는 *use_builtin_types*와 유사하지만, 날짜/시간 값에만 적용됩니다.

버전 3.3에서 변경: *use_builtin_types* 플래그가 추가되었습니다.

버전 3.8에서 변경: *headers* 매개 변수가 추가되었습니다.

HTTP와 HTTPS 트랜스포트는 모두 HTTP 기본 인증(Basic Authentication)을 위한 URL 구문 확장을 지원합니다: http://user:pass@host:port/path. user:pass 부분은 HTTP 'Authorization' 헤더로 base64 인코딩되고, XML-RPC 메서드를 호출할 때 연결 프로세스의 일부로 원격 서버로 전송됩니다. 원격 서버가 기본 인증 사용자와 비밀번호를 요구할 때만 이를 사용해야 합니다.

반환된 인스턴스는 원격 서버에서 해당 RPC 호출을 호출하는 데 사용할 수 있는 메서드가 있는 프락시 객체입니다. 원격 서버가 인트로스펙션(introspection) API를 지원하면, 프락시를 사용하여 원격 서버에서 지원하는 메서드를 조회하고 (서비스 검색, service discovery) 다른 서버 관련 메타 데이터를 가져올 수 있습니다.

적합한 형(예를 들어 XML을 통해 마샬링 할 수 있는 형)에는 다음이 포함됩니다 (별도로 표시된 경우를 제외하고는 같은 파이썬 형으로 역마샬링 됩니다):

XML-RPC 형	파이썬 형
`boolean`	*`bool`*
`int`, `i1`, `i2`, `i4`, `i8` 또는 `biginteger`	-2147483648에서 2147483647 범위의 *`int`*. 값은 `<int>` 태그를 얻습니다.
`double`이나 `float`	*`float`*. 값은 `<double>` 태그를 얻습니다.
`string`	*`str`*
`array`	적합한 요소를 포함하는 *`list`*나 *`tuple`*. 배열은 리스트로 반환됩니다.
`struct`	*`dict`*. 키는 문자열이어야 하며, 값은 적합한 형일 수 있습니다. 사용자 정의 클래스의 객체를 전달할 수 있습니다; `__dict__` 어트리뷰트만 전송됩니다.
`dateTime.iso8601`	*`DateTime`*이나 *`datetime.datetime`*. 반환되는 형은 *use_builtin_types*와 *use_datetime* 플래그 값에 따라 다릅니다.
`base64`	*`Binary`*, *`bytes`* 또는 *`bytearray`*. 반환되는 형은 *use_builtin_types* 플래그의 값에 따라 다릅니다.
`nil`	`None` 상수. *allow_none*이 참일 때만 전달이 허용됩니다.
`bigdecimal`	*`decimal.Decimal`*. 반환되는 형 전용.

이것이 XML-RPC가 지원하는 데이터형의 전체 집합입니다. 메서드 호출은 XML-RPC 서버 에러를 알리는 데 사용되는 특수 *`Fault`* 인스턴스나 HTTP/HTTPS 전송 계층의 에러를 알리는 데 사용되는 *`ProtocolError`*를 발생시킬 수도 있습니다. *`Fault`*와 *`ProtocolError`*는 모두 `Error`라는 베이스 클래스에서 파생됩니다. xmlrpc 클라이언트 모듈은 현재 내장형의 서브 클래스 인스턴스를 마샬링하지 않음에 유의하십시오.

문자열을 전달할 때, `<`, `>` 및 `&` 와 같은 XML에 특수한 문자는 자동으로 이스케이프 됩니다. 그러나, 0에서 31 사이의 ASCII 값을 가진 제어 문자(물론 탭, 줄 넘김 및 캐리지 리턴은 제외하고)와 같이 문자열에 XML에서 허용되지 않는 문자가 없도록 확인하는 것은 호출자의 책임입니다; 이렇게 하지 않으면 XML 형식이 잘못된 XML-RPC 요청이 발생합니다. XML-RPC를 통해 임의의 바이트열을 전달해야 하면, *`bytes`*나 *`bytearray`* 클래스 또는 아래 설명된 *`Binary`* 래퍼 클래스를 사용하십시오.

`Server`는 이전 버전과의 호환성을 위해 *`ServerProxy`*의 별칭으로 유지됩니다. 새 코드는 *`ServerProxy`*를 사용해야 합니다.

버전 3.5에서 변경: *context* 인자를 추가했습니다.

버전 3.6에서 변경: 접두사가 있는 형 태그 지원이 추가되었습니다 (예를 들어 `ex:nil`). 숫자를 위해 Apache XML-RPC 구현이 사용하는 추가 형의 역마샬링 지원이 추가되었습니다: `i1`, `i2`, `i8`, `biginteger`, `float` 및 `bigdecimal`. 설명은 https://ws.apache.org/xmlrpc/types.html 을 참조하십시오.

더 보기

XML-RPC HOWTO
여러 언어로 된 XML-RPC 연산과 클라이언트 소프트웨어에 대한 훌륭한 설명. XML-RPC 클라이언트 개발자가 알아야 할 거의 모든 것이 포함되어 있습니다.

XML-RPC Introspection
인트로스펙션을 위한 XML-RPC 프로토콜 확장을 설명합니다.

XML-RPC Specification
공식 명세.

22.21.1 ServerProxy 객체

ServerProxy 인스턴스에는 XML-RPC 서버가 받아들이는 각 원격 프로시저 호출에 해당하는 메서드가 있습니다. 메서드를 호출하면 RPC를 수행하는데, 이름과 인자 서명 모두로 디스패치 됩니다 (예를 들어 같은 메서드 이름이 여러 인자 서명으로 오버로드 될 수 있습니다). RPC는 값을 반환하여 완료되는데, 값은 적합한 형으로 반환된 데이터이거나 에러를 나타내는 *Fault* 나 *ProtocolError* 객체일 수 있습니다.

XML 인트로스팩션 API를 지원하는 서버는 예약된 `system` 어트리뷰트 밑에 그룹화된 몇 가지 공통 메서드를 지원합니다:

`ServerProxy.system.`**`listMethods`**`()`

이 메서드는 문자열 리스트를 반환하는데, XML-RPC 서버가 지원하는 각 (system이 아닌) 메서드마다 하나씩 제공됩니다.

`ServerProxy.system.`**`methodSignature`**`(`*name*`)`

이 메서드는 하나의 매개 변수를 취하는데, XML-RPC 서버에 의해 구현된 메서드의 이름입니다. 이 메서드에 대해 가능한 서명의 배열을 반환합니다. 서명은 형의 배열입니다. 이 형 중 첫 번째는 메서드의 반환형이고 나머지는 매개 변수입니다.

다중 서명(즉 오버로딩)이 허용되므로, 이 메서드는 하나가 아닌 서명의 리스트를 반환합니다.

서명 자체는 메서드가 기대하는 최상위 매개 변수로 제한됩니다. 예를 들어, 메서드가 구조체 배열 하나를 매개 변수로 기대하고, 문자열을 반환하면, 서명은 단순히 "string, array" 입니다. 세 개의 정수를 기대하고 문자열을 반환하면, 서명은 "string, int, int, int" 입니다.

메서드에 서명이 정의되지 않으면, 배열이 아닌 값이 반환됩니다. 파이썬에서 이것은 반환된 값의 형이 리스트 이외의 어떤 것이 됨을 의미합니다.

`ServerProxy.system.`**`methodHelp`**`(`*name*`)`

이 메서드는 하나의 매개 변수를 취하는데, XML-RPC 서버에 의해 구현된 메서드의 이름입니다. 해당 메서드의 사용법을 기술하는 설명서 문자열을 반환합니다. 이러한 문자열이 없으면, 빈 문자열이 반환됩니다. 설명서 문자열에 HTML 마크업이 포함될 수 있습니다.

버전 3.5에서 변경: *ServerProxy* 인스턴스는 하부 트랜스포트를 닫기 위한 컨텍스트 관리자 프로토콜을 지원합니다.

다음은 실제 예입니다. 서버 코드:

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

이전 서버에 대한 클라이언트 코드:

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```

22.21.2 DateTime 객체

class xmlrpc.client.**DateTime**

이 클래스는 에포크(epoch) 이후의 초, 시간 튜플, ISO 8601 시간/날짜 문자열 또는 *datetime.datetime*으로 초기화될 수 있습니다. 주로 마샬링/역마샬링 코드 내부에서 사용하기 위해 지원되는, 다음과 같은 메서드가 있습니다:

decode(*string*)

인스턴스의 새 시간 값으로 문자열을 받아들입니다.

encode(*out*)

이 *DateTime* 항목의 XML-RPC 인코딩을 *out* 스트림 객체에 씁니다.

풍부한 비교와 __repr__() 메서드를 통해 특정 파이썬 내장 연산자도 지원합니다.

다음은 실제 예입니다. 서버 코드:

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

이전 서버에 대한 클라이언트 코드:

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# ISO8601 문자열을 datetime 객체로 변환합니다
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

22.21.3 Binary 객체

class xmlrpc.client.**Binary**

이 클래스는 바이트열 데이터(NUL을 포함할 수 있습니다)로 초기화될 수 있습니다. *Binary* 객체의 내용에 대한 기본 액세스는 어트리뷰트에 의해 제공됩니다:

data

Binary 인스턴스로 캡슐화된 바이너리 데이터. 데이터는 *bytes* 객체로 제공됩니다.

Binary 객체에는 주로 마샬링/역마샬링 코드 내부에서 사용하기 위해 지원되는 다음과 같은 메서드가 있습니다:

decode(*bytes*)

base64 *bytes* 객체를 받아들이고 인스턴스의 새 데이터로 디코딩합니다.

encode(*out*)

이 바이너리 항목의 XML-RPC base64 인코딩을 *out* 스트림 객체에 씁니다.

인코딩된 데이터에는 **RFC 2045 섹션 6.8**에 따라 76문자마다 줄 바꿈이 있는데, 이는 XML-RPC 명세가 작성될 때 사실상 표준 base64 명세였습니다.

`__eq__()`와 `__ne__()` 메서드를 통해 특정 파이썬 내장 연산자도 지원합니다.

바이너리 객체의 사용 예. XMLRPC를 통해 이미지를 전송할 것입니다:

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

클라이언트는 이미지를 가져와서 파일에 저장합니다:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

22.21.4 Fault 객체

class xmlrpc.client.**Fault**

Fault 객체는 XML-RPC 결함 태그의 내용을 캡슐화합니다. Fault 객체에는 다음과 같은 어트리뷰트가 있습니다:

faultCode

결함 형을 나타내는 int.

faultString

결함과 연관된 진단 메시지가 포함된 문자열.

다음 예제에서는 복소수 형의 객체를 반환하여 의도적으로 *Fault*를 발생시킵니다. 서버 코드:

```
from xmlrpc.server import SimpleXMLRPCServer

# 복소수를 반환하기 때문에 마샬링 에러가 발생합니다
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

server.serve_forever()
```

이전 서버에 대한 클라이언트 코드:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

22.21.5 ProtocolError 객체

class xmlrpc.client.**ProtocolError**

ProtocolError 객체는 하부 전송 계층에서의 프로토콜 에러를 기술합니다 (가령 URI로 명명된 서버가 없을 때 404 'not found' 에러). 다음과 같은 어트리뷰트가 있습니다:

url

에러를 일으킨 URI나 URL.

errcode

에러 코드.

errmsg

에러 메시지나 진단 문자열.

headers

에러를 일으킨 HTTP/HTTPS 요청의 헤더를 포함하는 딕셔너리.

다음 예에서는 잘못된 URI를 제공하여 의도적으로 *ProtocolError*를 발생시킵니다:

```
import xmlrpc.client

# XMLRPC 요청에 응답하지 않는 URI로 ServerProxy를 만듭니다
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
    print("A protocol error occurred")
    print("URL: %s" % err.url)
    print("HTTP/HTTPS headers: %s" % err.headers)
    print("Error code: %d" % err.errcode)
    print("Error message: %s" % err.errmsg)
```

22.21.6 MultiCall 객체

MultiCall 객체는 원격 서버에 대한 여러 호출을 단일 요청으로 캡슐화하는 방법을 제공합니다[1].

class xmlrpc.client.**MultiCall**(*server*)

boxcar 메서드 호출에 사용되는 객체를 만듭니다. *server*는 최종 호출 대상입니다. 결과 객체를 호출할 수 있지만, 즉시 None을 반환하고, 호출 이름과 매개 변수를 *MultiCall* 객체에 저장하기만 합니다. 객체 자체를 호출하면 저장된 모든 호출이 단일 system.multicall 요청으로 전송됩니다. 이 호출의 결과는 제너레이터입니다; 이 제너레이터를 이터레이트 하면 개별 결과를 산출합니다.

이 클래스의 사용 예는 다음과 같습니다. 서버 코드:

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
```

(다음 페이지에 계속)

[1] 이 접근법은 xmlrpc.com에서의 토론에서 처음 제시되었습니다.

(이전 페이지에서 계속)

```
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# 간단한 산술 함수들이 있는 간단한 서버
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

이전 서버에 대한 클라이언트 코드:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

22.21.7 편의 함수

xmlrpc.client.**dumps**(*params*, *methodname=None*, *methodresponse=None*, *encoding=None*, *allow_none=False*)

*params*를 XML-RPC 요청으로, 또는 *methodresponse*가 참이면 응답으로 변환합니다. *params*는 인자의 튜플이거나 `Fault` 예외 클래스의 인스턴스일 수 있습니다. *methodresponse*가 참이면, 단일 값만 반환될 수 있는데, *params*의 길이가 1이어야 한다는 뜻입니다. 제공되면, *encoding*은 생성된 XML에서 사용할 인코딩입니다; 기본값은 UTF-8입니다. 파이썬의 `None` 값은 표준 XML-RPC에서 사용할 수 없습니다; 확장을 통해 이를 사용하려면 *allow_none*에 참값을 제공하십시오.

xmlrpc.client.**loads**(*data*, *use_datetime=False*, *use_builtin_types=False*)

XML-RPC 요청이나 응답을 파이썬 객체 `(params, methodname)`로 변환합니다. *params*는 인자의 튜플입니다; *methodname*은 문자열이거나 패킷에 메서드 이름이 없으면 `None`입니다. XML-RPC 패킷이 결함 조건을 나타내면, 이 함수는 `Fault` 예외를 발생시킵니다. *use_builtin_types* 플래그를 사용하여 날짜/시간 값을 `datetime.datetime` 객체로 표현하고 바이너리 데이터를 `bytes` 객체로 표현할 수 있습니다; 이 플래그는 기본적으로 거짓입니다.

사용되지 않는 *use_datetime* 플래그는 *use_builtin_types*와 유사하지만, 날짜/시간 값에만 적용됩니다.

버전 3.3에서 변경: *use_builtin_types* 플래그가 추가되었습니다.

22.21.8 클라이언트 사용 예

```
# 간단한 테스트 프로그램 (XML-RPC 명세에서 인용)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # 로컬 서버
with ServerProxy("http://betty.userland.com") as proxy:

    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

HTTP 프락시를 통해 XML-RPC 서버에 액세스하려면, 사용자 정의 트랜스포트를 정의해야 합니다. 다음 예제는 방법을 보여줍니다:

```
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):

    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
server = xmlrpc.client.ServerProxy('http://betty.userland.com',␣
↪transport=transport)
print(server.examples.getStateName(41))
```

22.21.9 클라이언트와 서버 사용 예

SimpleXMLRPCServer 예제를 참조하십시오.

22.22 `xmlrpc.server` — 기본 XML-RPC 서버

소스 코드: Lib/xmlrpc/server.py

`xmlrpc.server` 모듈은 파이썬으로 작성된 XML-RPC 서버를 위한 기본 서버 프레임워크를 제공합니다. 서버는 `SimpleXMLRPCServer`를 사용하여 독립적이거나, `CGIXMLRPCRequestHandler`를 사용하여 CGI 환경에 내장될 수 있습니다.

> **경고**
>
> The `xmlrpc.server` module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data, see *XML security*.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

class xmlrpc.server.**SimpleXMLRPCServer**(*addr*, *requestHandler=SimpleXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=False*)

새 서버 인스턴스를 만듭니다. 이 클래스는 XML-RPC 프로토콜에 의해 호출될 수 있는 함수를 등록하는 메서드를 제공합니다. *requestHandler* 매개 변수는 요청 처리기 인스턴스의 팩토리여야 합니다; 기본값은 `SimpleXMLRPCRequestHandler` 입니다. *addr*과 *requestHandler* 매개 변수는 `socketserver.TCPServer` 생성자에 전달됩니다. *logRequests*가 참(기본값)이면, 요청이 로그 됩니다; 이 매개 변수를 거짓으로 설정하면 로깅이 해제됩니다. *allow_none*과 *encoding* 매개 변수는 `xmlrpc.client`로 전달되고 서버에서 반환될 XML-RPC 응답을 제어합니다. *bind_and_activate* 매개 변수는 생성자가 `server_bind()`와 `server_activate()`를 즉시 호출하는지를 제어합니다; 기본값은 참입니다. 이를 거짓으로 설정하면 코드가 주소가 바인드되기 전에 *allow_reuse_address* 클래스 변수를 조작할 수 있습니다. *use_builtin_types* 매개 변수는 `loads()` 함수로 전달되며 날짜/시간 값이나 바이너리 데이터가 수신될 때 처리되는 형을 제어합니다; 기본값은 거짓입니다.

버전 3.3에서 변경: *use_builtin_types* 플래그가 추가되었습니다.

class xmlrpc.server.**CGIXMLRPCRequestHandler**(*allow_none=False*, *encoding=None*, *use_builtin_types=False*)

CGI 환경에서 XML-RPC 요청을 처리할 새 인스턴스를 만듭니다. *allow_none*과 *encoding* 매개 변수는 `xmlrpc.client`로 전달되고 서버에서 반환될 XML-RPC 응답을 제어합니다. *use_builtin_types* 매개 변수는 `loads()` 함수로 전달되며 날짜/시간 값이나 바이너리 데이터가 수신될 때 처리되는 형을 제어합니다; 기본값은 거짓입니다.

버전 3.3에서 변경: *use_builtin_types* 플래그가 추가되었습니다.

class xmlrpc.server.**SimpleXMLRPCRequestHandler**

새 요청 처리기 인스턴스를 만듭니다. 이 요청 처리기는 `POST` 요청을 지원하고 `SimpleXMLRPCServer` 생성자 매개 변수에 대한 *logRequests* 매개 변수가 적용되도록 로깅을 수정합니다.

22.22.1 SimpleXMLRPCServer 객체

`SimpleXMLRPCServer` 클래스는 `socketserver.TCPServer`를 기반으로 하며 간단한 독립형 XML-RPC 서버를 작성하는 수단을 제공합니다.

SimpleXMLRPCServer.**register_function**(*function=None*, *name=None*)

XML-RPC 요청에 응답할 수 있는 함수를 등록합니다. *name*이 제공되면, *function*과 연결되는 메서드 이름이 되고, 그렇지 않으면 `function.__name__`이 사용됩니다. *name*은 문자열이며 마침표 문자를 포함하여 파이썬 식별자에서 유효하지 않은 문자를 포함할 수 있습니다.

이 메서드는 데코레이터로도 사용할 수 있습니다. 데코레이터로 사용될 때, *name*은 *function*을 *name*으로 등록하기 위해 키워드 인자로만 제공될 수 있습니다. *name*을 제공하지 않으면, `function.__name__`이 사용됩니다.

버전 3.7에서 변경: `register_function()`은 데코레이터로 사용할 수 있습니다.

SimpleXMLRPCServer.**register_instance**(*instance*, *allow_dotted_names=False*)

`register_function()`을 사용하여 등록되지 않은 메서드 이름을 노출하는데 사용되는 객체를 등록합니다. *instance*가 `_dispatch()` 메서드를 포함하면, 요청된 메서드 이름과 요청의 매개 변수로 호출됩니다. API는 `def _dispatch(self, method, params)` 입니다 (*params*가 가변 인자 목록을 나타내지 않음에 유의하십시오). 이것이 작업을 수행하기 위해 하부 함수를 호출하면, 해당 함수를 매개 변수 리스트를 확장하여 `func(*params)`로 호출합니다. `_dispatch()`의 반환 값이 클라이언트에 결과로 반환됩니다. *instance*에 `_dispatch()` 메서드가 없으면, 요청된 메서드의 이름과 일치하는 어트리뷰트를 검색합니다.

선택적 *allow_dotted_names* 인자가 참이고 인스턴스에 `_dispatch()` 메서드가 없으면, 요청된 메서드 이름에 마침표가 포함될 때, 메서드 이름의 각 구성 요소가 개별적으로 검색되어, 간단한 계층 구조

검색이 수행되는 효과를 줍니다. 이 검색에서 찾은 값은 요청의 매개 변수로 호출되며 반환 값은 클라이언트로 다시 전달됩니다.

> **경고**
>
> *allow_dotted_names* 옵션을 활성화하면 침입자가 모듈의 전역 변수에 액세스할 수 있으며 침입자가 여러분의 기계에서 임의의 코드를 실행할 수 있습니다. 안전한 폐쇄 네트워크에서만 이 옵션을 사용하십시오.

`SimpleXMLRPCServer.register_introspection_functions()`

XML-RPC 내부 검사 함수 `system.listMethods`, `system.methodHelp` 및 `system.methodSignature`를 등록합니다.

`SimpleXMLRPCServer.register_multicall_functions()`

XML-RPC 다중 호출(multicall) 함수 system.multicall을 등록합니다.

`SimpleXMLRPCRequestHandler.rpc_paths`

XML-RPC 요청을 수신하기 위한 URL의 유효한 경로 부분을 나열하는 튜플이어야 하는 어트리뷰트 값. 다른 경로로 들어오는 요청은 404 "no such page" HTTP 에러를 발생시킵니다. 이 튜플이 비어 있으면, 모든 경로를 유효한 것으로 간주합니다. 기본값은 `('/', '/RPC2')` 입니다.

SimpleXMLRPCServer 예제

서버 코드:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# 특정 경로로 제한합니다.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# 서버를 만듭니다
with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # pow() 함수를 등록합니다; pow.__name__ 값을 이름으로 사용하는데,
    # 그냥 'pow' 입니다.
    server.register_function(pow)

    # 함수를 다른 이름으로 등록합니다
    def adder_function(x, y):
        return x + y
    server.register_function(adder_function, 'add')

    # 인스턴스를 등록합니다; 인스턴스의 모든 메서드가 XML-RPC 메서드로 노출됩니다
    # (이 경우는 'mul' 뿐입니다).
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

    # 서버의 메인 루프를 실행합니다
    server.serve_forever()
```

다음 클라이언트 코드는 앞의 서버가 제공하는 메서드를 호출합니다:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # 2**3 = 8 을 반환합니다
print(s.add(2,3))  # 5 를 반환합니다
print(s.mul(5,2))  # 5*2 = 10 을 반환합니다

# 사용할 수 있는 메서드의 목록을 인쇄합니다
print(s.system.listMethods())
```

register_function() 은 데코레이터로도 사용할 수 있습니다. 앞의 서버 예제에서 데코레이터 방식으로 함수를 등록할 수 있습니다:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # pow() 함수를 등록합니다; pow.__name__ 값을 이름으로 사용하는데,
    # 그냥 'pow' 입니다.
    server.register_function(pow)

    # register_function을 데코레이터로 사용해서, 함수를 다른 이름으로 등록합니다.
    # *name* 은 키워드 인자로만 줄 수 있습니다.
    @server.register_function(name='add')
    def adder_function(x, y):
        return x + y

    # 함수를 function.__name__ 으로 등록합니다.
    @server.register_function
    def mul(x, y):
        return x * y

    server.serve_forever()
```

Lib/xmlrpc/server.py 모듈에 포함된 다음 예는 점으로 구분된 이름을 허용하고 다중 호출 함수를 등록하는 서버를 보여줍니다.

> **경고**
>
> *allow_dotted_names* 옵션을 활성화하면 침입자가 모듈의 전역 변수에 액세스할 수 있으며 침입자가 여러분의 기계에서 임의의 코드를 실행할 수 있습니다. 이 예제는 안전한 폐쇄 네트워크 내에서만 사용하십시오.

```
import datetime

class ExampleService:
    def getData(self):
        return '42'

    class currentTime:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        @staticmethod
        def getCurrentTime():
            return datetime.datetime.now()

with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
    server.register_instance(ExampleService(), allow_dotted_names=True)
    server.register_multicall_functions()
    print('Serving XML-RPC on localhost port 8000')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nKeyboard interrupt received, exiting.")
        sys.exit(0)
```

이 ExampleService 데모는 명령 줄에서 호출할 수 있습니다:

```
python -m xmlrpc.server
```

위 서버와 상호 작용하는 클라이언트는 `Lib/xmlrpc/client.py`에 포함되어 있습니다:

```
server = ServerProxy("http://localhost:8000")

try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2,9)
multi.add(1,2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

데모 XMLRPC 서버와 상호 작용하는 이 클라이언트는 다음과 같이 호출할 수 있습니다:

```
python -m xmlrpc.client
```

22.22.2 CGIXMLRPCRequestHandler

`CGIXMLRPCRequestHandler` 클래스는 파이썬 CGI 스크립트로 전송된 XML-RPC 요청을 처리하는 데 사용할 수 있습니다.

`CGIXMLRPCRequestHandler.`**`register_function`**(*function=None*, *name=None*)

XML-RPC 요청에 응답할 수 있는 함수를 등록합니다. *name*이 제공되면, *function*과 연결되는 메서드 이름이 되고, 그렇지 않으면 `function.__name__`이 사용됩니다. *name*은 문자열이며 마침표 문자를 포함하여 파이썬 식별자에서 유효하지 않은 문자를 포함할 수 있습니다.

이 메서드는 데코레이터로도 사용할 수 있습니다. 데코레이터로 사용될 때, *name*은 *function*을 *name*으로 등록하기 위해 키워드 인자로만 제공될 수 있습니다. *name*을 제공하지 않으면, `function.__name__`이 사용됩니다.

버전 3.7에서 변경: *`register_function()`*은 데코레이터로 사용할 수 있습니다.

CGIXMLRPCRequestHandler.**register_instance**(*instance*)

*register_function()*을 사용하여 등록되지 않은 메서드 이름을 노출하는데 사용되는 객체를 등록합니다. instance가 `_dispatch()` 메서드를 포함하면, 요청된 메서드 이름과 요청의 매개 변수로 호출됩니다; 반환 값이 클라이언트에 결과로 반환됩니다. instance에 `_dispatch()` 메서드가 없으면, 요청된 메서드의 이름과 일치하는 어트리뷰트를 검색합니다; 요청된 메서드 이름에 마침표가 포함될 때, 메서드 이름의 각 구성 요소가 개별적으로 검색되어, 간단한 계층 구조 검색이 수행되는 효과를 줍니다. 이 검색에서 찾은 값은 요청의 매개 변수로 호출되며 반환 값은 클라이언트로 다시 전달됩니다.

CGIXMLRPCRequestHandler.**register_introspection_functions**()

XML-RPC 내부 검사 함수 `system.listMethods`, `system.methodHelp` 및 `system.methodSignature`를 등록합니다.

CGIXMLRPCRequestHandler.**register_multicall_functions**()

XML-RPC 다중 호출(multicall) 함수 `system.multicall`을 등록합니다.

CGIXMLRPCRequestHandler.**handle_request**(*request_text=None*)

XML-RPC 요청을 처리합니다. *request_text*가 제공되면, HTTP 서버가 제공한 POST 데이터여야 합니다, 그렇지 않으면 stdin의 내용이 사용됩니다.

예:

```
class MyFuncs:
    def mul(self, x, y):
        return x * y


handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

22.22.3 XMLRPC 서버 문서화

이 클래스들은 HTTP GET 요청에 대한 응답으로 HTML 설명서를 제공하기 위해 위의 클래스를 확장합니다. 서버는 *DocXMLRPCServer*를 사용하여 독립적이거나, *DocCGIXMLRPCRequestHandler*를 사용하여 CGI 환경에 내장될 수 있습니다.

class xmlrpc.server.**DocXMLRPCServer**(*addr, requestHandler=DocXMLRPCRequestHandler, logRequests=True, allow_none=False, encoding=None, bind_and_activate=True, use_builtin_types=True*)

새 서버 인스턴스를 만듭니다. 모든 매개 변수는 *SimpleXMLRPCServer*와 같은 의미입니다; *requestHandler*의 기본값은 *DocXMLRPCRequestHandler* 입니다.

버전 3.3에서 변경: *use_builtin_types* 플래그가 추가되었습니다.

class xmlrpc.server.**DocCGIXMLRPCRequestHandler**

CGI 환경에서 XML-RPC 요청을 처리할 새 인스턴스를 만듭니다.

class xmlrpc.server.**DocXMLRPCRequestHandler**

새 요청 처리기 인스턴스를 만듭니다. 이 요청 처리기는 XML-RPC POST 요청과 설명서 GET 요청을 지원하고, *DocXMLRPCServer* 생성자 매개 변수에 대한 *logRequests* 매개 변수가 적용되도록 로깅을 수정합니다.

22.22.4 DocXMLRPCServer 객체

DocXMLRPCServer 클래스는 *SimpleXMLRPCServer* 에서 파생되며 스스로 설명하는 독립형 XML-RPC 서버를 만드는 수단을 제공합니다. HTTP POST 요청은 XML-RPC 메서드 호출로 처리됩니다. HTTP GET 요청은 pydoc 스타일 HTML 문서를 생성하는 것으로 처리합니다. 이를 통해 서버는 자체 웹 기반 설명서를 제공할 수 있습니다.

DocXMLRPCServer.**set_server_title**(*server_title*)

생성된 HTML 설명서에 사용되는 제목을 설정합니다. 이 제목은 HTML "title" 요소 안에서 사용됩니다.

DocXMLRPCServer.**set_server_name**(*server_name*)

생성된 HTML 설명서에 사용되는 이름을 설정합니다. 이 이름은 설명서의 최상단의 "h1" 요소 안에 나타납니다.

DocXMLRPCServer.**set_server_documentation**(*server_documentation*)

생성된 HTML 설명서에 사용되는 설명을 설정합니다. 이 설명은 설명서에서 서버 이름 아래 단락으로 나타납니다.

22.22.5 DocCGIXMLRPCRequestHandler

DocCGIXMLRPCRequestHandler 클래스는 *CGIXMLRPCRequestHandler* 에서 파생되며 스스로 설명하는 XML-RPC CGI 스크립트를 만드는 수단을 제공합니다. HTTP POST 요청은 XML-RPC 메서드 호출로 처리됩니다. HTTP GET 요청은 pydoc 스타일 HTML 문서를 생성하는 것으로 처리합니다. 이를 통해 서버는 자체 웹 기반 설명서를 제공할 수 있습니다.

DocCGIXMLRPCRequestHandler.**set_server_title**(*server_title*)

생성된 HTML 설명서에 사용되는 제목을 설정합니다. 이 제목은 HTML "title" 요소 안에서 사용됩니다.

DocCGIXMLRPCRequestHandler.**set_server_name**(*server_name*)

생성된 HTML 설명서에 사용되는 이름을 설정합니다. 이 이름은 설명서의 최상단의 "h1" 요소 안에 나타납니다.

DocCGIXMLRPCRequestHandler.**set_server_documentation**(*server_documentation*)

생성된 HTML 설명서에 사용되는 설명을 설정합니다. 이 설명은 설명서에서 서버 이름 아래 단락으로 나타납니다.

22.23 ipaddress — IPv4/IPv6 조작 라이브러리

소스 코드: Lib/ipaddress.py

---

*ipaddress*는 IPv4와 IPv6 주소와 네트워크를 만들고, 조작하고, 연산하는 기능을 제공합니다.

이 모듈의 함수와 클래스를 사용하면 두 호스트가 같은 서브 네트에 있는지 확인하기, 특정 서브 네트의 모든 호스트를 이터레이트 하기, 문자열이 유효한 IP 주소나 네트워크를 나타내는지 검사하기 등 IP 주소와 관련된 다양한 작업을 간단하게 처리할 수 있습니다.

이것은 전체 모듈 API 레퍼런스입니다 - 개요와 소개는 ipaddress-howto를 참조하십시오.

Added in version 3.3.

22.23.1 편의 팩토리 함수

ipaddress 모듈은 IP 주소, 네트워크 및 인터페이스를 편리하게 만드는 팩토리 함수를 제공합니다:

ipaddress.**ip_address**(*address*)

인자로 전달된 IP 주소에 따라 *IPv4Address* 나 *IPv6Address* 객체를 반환합니다. IPv4나 IPv6 주소가 제공될 수 있습니다; `2**32` 보다 작은 정수는 기본적으로 IPv4로 간주합니다. *address*가 유효한 IPv4나 IPv6 주소를 나타내지 않으면 *ValueError*가 발생합니다.

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

ipaddress.**ip_network**(*address, strict=True*)

인자로 전달된 IP 주소에 따라 *IPv4Network* 나 *IPv6Network* 객체를 반환합니다. *address*는 IP 네트워크를 나타내는 문자열이나 정수입니다. IPv4나 IPv6 네트워크가 제공될 수 있습니다; `2**32` 보다 작은 정수는 기본적으로 IPv4로 간주합니다. *strict*는 *IPv4Network* 나 *IPv6Network* 생성자로 전달됩니다. *address*가 유효한 IPv4나 IPv6 주소를 나타내지 않거나, 네트워크에 호스트 비트가 설정되어있으면 *ValueError*가 발생합니다.

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

ipaddress.**ip_interface**(*address*)

인자로 전달된 IP 주소에 따라 *IPv4Interface* 나 *IPv6Interface* 객체를 반환합니다. *address*는 IP 주소를 나타내는 문자열이나 정수입니다. IPv4나 IPv6 주소가 제공될 수 있습니다. `2**32` 보다 작은 정수는 기본적으로 IPv4로 간주합니다. *address*가 유효한 IPv4나 IPv6 주소를 나타내지 않으면 *ValueError*가 발생합니다.

이러한 편의 함수들의 한 가지 단점은 IPv4와 IPv6 형식을 모두 처리해야 한다는 것이, 함수가 IPv4나 IPv6 형식 중 어느 것을 의도하는지 알 수 없기 때문에, 에러 메시지가 정확한 에러에 대한 정보를 최소한으로만 제공한다는 것을 의미한다는 것입니다. 적절한 버전 별 클래스 생성자를 직접 호출하여 더 자세한 에러 보고를 얻을 수 있습니다.

22.23.2 IP 주소

주소 객체

*IPv4Address*와 *IPv6Address* 객체는 많은 공통 어트리뷰트를 공유합니다. IPv6 주소에만 의미가 있는 일부 어트리뷰트는 두 IP 버전을 모두 올바르게 처리하는 코드를 더 쉽게 작성할 수 있도록 *IPv4Address* 객체에 의해 구현됩니다. 주소 객체는 해시 가능해서, 딕셔너리에서 키로 사용할 수 있습니다.

class ipaddress.**IPv4Address**(*address*)

IPv4 주소를 구성합니다. *address*가 유효한 IPv4 주소가 아니면 *AddressValueError* 가 발생합니다.

다음은 유효한 IPv4 주소를 구성합니다:

1. 점으로 구분된, 0–255 범위의 4개의 십진 정수로 구성되는 십진 점 표기법의 문자열 (예를 들어 `192.168.0.1`). 각 정수는 주소의 옥텟(바이트)을 나타냅니다. 8진수 표기법과의 혼동을 방지하기 위해 선행 0은 허용되지 않습니다.
2. 32비트에 맞는 정수.
3. 길이가 4인 *bytes* 객체에 채워진 정수 (가장 유효한 옥텟이 먼저 옵니다).

```
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xC0\xA8\x00\x01')
IPv4Address('192.168.0.1')
```

버전 3.8에서 변경: Leading zeros are tolerated, even in ambiguous cases that look like octal notation.

버전 3.9.5에서 변경: Leading zeros are no longer tolerated and are treated as an error. IPv4 address strings are now parsed as strict as glibc *inet_pton()*.

version

적절한 버전 번호: IPv4의 경우 `4`, IPv6의 경우 `6`.

버전 3.14에서 변경: Made available on the class.

max_prefixlen

이 버전에 대한 주소 표현의 총 비트 수: IPv4의 경우 `32`, IPv6의 경우 `128`.

접두사는 주소가 네트워크의 일부인지를 판별하기 위해 비교되는 주소의 선행 비트 수를 정의합니다.

버전 3.14에서 변경: Made available on the class.

compressed

exploded

점으로 구분된 십진 표기법의 문자열 표현. 선행 0은 표현에 포함되지 않습니다.

IPv4는 옥텟이 0으로 설정된 주소에 대한 약식 표기법을 정의하지 않기 때문에, 이 두 어트리뷰트는 IPv4 주소의 경우 항상 `str(addr)` 과 같습니다. 이러한 어트리뷰트를 노출하면 IPv4와 IPv6 주소를 모두 처리할 수 있는 디스플레이 코드를 더 쉽게 작성할 수 있습니다.

packed

이 주소의 바이너리 표현 - 적절한 길이의 *bytes* 객체 (가장 유효한 옥텟이 먼저 옵니다). IPv4의 경우 4바이트, IPv6의 경우 16바이트입니다.

reverse_pointer

IP 주소에 대한 역 DNS PTR 레코드의 이름, 예를 들어:

```
>>> ipaddress.ip_address("127.0.0.1").reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> ipaddress.ip_address("2001:db8::1").reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa'
```

이 이름은 결정된(resolved) 호스트명 자체가 아니라 PTR 조회를 수행하는 데 사용할 수 있는 이름입니다.

Added in version 3.5.

is_multicast

주소가 멀티캐스트용으로 예약되었으면 `True`. **RFC 3171**(IPv4의 경우)이나 **RFC 2373**(IPv6의 경우)을 참조하십시오.

is_private

다음을 제외하고 iana-ipv4-special-registry(IPv4의 경우)나 iana-ipv6-special-registry(IPv6의 경우)에 의해 주소가 전역적으로 도달할 수 없도록 정의되었으면 `True`:

- `is_private` is `False` for the shared address space (`100.64.0.0/10`)
- For IPv4-mapped IPv6-addresses the `is_private` value is determined by the semantics of the underlying IPv4 addresses and the following condition holds (see *IPv6Address.ipv4_mapped*):

  ```
  address.is_private == address.ipv4_mapped.is_private
  ```

`is_private` has value opposite to *is_global*, except for the shared address space (`100.64.0.0/10` range) where they are both `False`.

버전 3.13에서 변경: Fixed some false positives and false negatives.

- `192.0.0.0/24` is considered private with the exception of `192.0.0.9/32` and `192.0.0.10/32` (previously: only the `192.0.0.0/29` sub-range was considered private).
- `64:ff9b:1::/48` is considered private.
- `2002::/16` is considered private.

- There are exceptions within `2001::/23` (otherwise considered private): `2001:1::1/128`, `2001:1::2/128`, `2001:3::/32`, `2001:4:112::/48`, `2001:20::/28`, `2001:30::/28`. The exceptions are not considered private.

is_global

다음을 제외하고 iana-ipv4-special-registry(IPv4의 경우)나 iana-ipv6-special-registry(IPv6의 경우)에 의해 주소가 전역적으로 도달할 수 있도록 정의되었으면 `True`:

For IPv4-mapped IPv6-addresses the `is_private` value is determined by the semantics of the underlying IPv4 addresses and the following condition holds (see *`IPv6Address.ipv4_mapped`*):

```
address.is_global == address.ipv4_mapped.is_global
```

`is_global` has value opposite to *`is_private`*, except for the shared address space (`100.64.0.0/10` range) where they are both `False`.

Added in version 3.4.

버전 3.13에서 변경: Fixed some false positives and false negatives, see *`is_private`* for details.

is_unspecified

주소가 지정되지 않았으면 `True`. **RFC 5735**(IPv4의 경우)나 **RFC 2373**(IPv6의 경우)을 참조하십시오.

is_reserved

`True` if the address is noted as reserved by the IETF. For IPv4, this is only `240.0.0.0/4`, the `Reserved` address block. For IPv6, this is all addresses allocated as `Reserved by IETF` for future use.

> **참고**
>
> For IPv4, `is_reserved` is not related to the address block value of the `Reserved-by-Protocol` column in iana-ipv4-special-registry.

> **조심**
>
> For IPv6, `fec0::/10` a former Site-Local scoped address prefix is currently excluded from that list (see *`is_site_local`* & **RFC 3879**).

is_loopback

이것이 루프 백 주소이면 `True`. **RFC 3330**(IPv4의 경우)이나 **RFC 2373**(IPv6의 경우)을 참조하십시오.

is_link_local

주소가 링크 로컬 사용을 위해 예약되었으면 `True`. **RFC 3927**을 참조하십시오.

ipv6_mapped

`IPv4Address` object representing the IPv4-mapped IPv6 address. See **RFC 4291**.

Added in version 3.13.

`IPv4Address.`**`__format__`**(*fmt*)

명시적 포맷 문자열로 제어되는 IP 주소의 문자열 표현을 반환합니다. *fmt*는 다음 중 하나일 수 있습니다: *`str()`*과 동등한 기본 옵션인 `'s'`, 0으로 채워진 이진수 문자열을 위한 `'b'`, 대문자나 소문자 16진수 표현을 위한 `'X'`나 `'x'`, 또는 IPv4 주소의 경우 `'b'`와 IPv6의 경우 `'x'`와 동등한 `'n'`. 이진수와 16진수 표현의 경우, 형식 지정자 `'#'`과 그룹화 옵션 `'_'`를 사용할 수 있습니다. `__format__`은 `format`, `str.format` 및 f-문자열에서 사용됩니다.

```
>>> format(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> '{:#b}'.format(ipaddress.IPv4Address('192.168.0.1'))
'0b11000000101010000000000000000001'
>>> f'{ipaddress.IPv6Address("2001:db8::1000"):s}'
'2001:db8::1000'
>>> format(ipaddress.IPv6Address('2001:db8::1000'), '_X')
'2001_0DB8_0000_0000_0000_0000_0000_1000'
>>> '{:#_n}'.format(ipaddress.IPv6Address('2001:db8::1000'))
'0x2001_0db8_0000_0000_0000_0000_0000_1000'
```

Added in version 3.9.

class ipaddress.**IPv6Address**(*address*)

IPv6 주소를 구성합니다. *address*가 유효한 IPv6 주소가 아니면 `AddressValueError` 가 발생합니다.

다음은 유효한 IPv6 주소를 구성합니다:

1. 4개의 16진수로 구성된 그룹 8개로 구성된 문자열, 각 그룹은 16비트를 나타냅니다. 그룹은 콜론으로 구분됩니다. 이것은 펼쳐진(*exploded*) (longhand) 표기법을 기술합니다. 문자열은 다양한 방법으로 압축될(*compressed*) (약식 표기법) 수도 있습니다. 자세한 내용은 **RFC 4291**을 참조하십시오. 예를 들어, `"0000:0000:0000:0000:0000:0abc:0007:0def"`는 `"::abc:7:def"`로 압축될 수 있습니다.

   선택적으로, 문자열은 접미사 `%scope_id`로 표시되는 스코프 존(scope zone) ID를 가질 수도 있습니다. 존재하면, 스코프 ID는 비어 있지 않아야 하며, `%`를 포함할 수 없습니다. 자세한 내용은 **RFC 4007**을 참조하십시오. 예를 들어, `fe80::1234%1`는 노드의 첫 번째 링크에서 주소 `fe80::1234`를 식별할 수 있습니다.

2. 128비트에 맞는 정수.

3. 길이가 16인 `bytes` 객체에 채워진 정수, 빅 엔디안.

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
>>> ipaddress.IPv6Address('ff02::5678%1')
IPv6Address('ff02::5678%1')
```

compressed

주소 표현의 짧은 형식, 그룹에서 선행 0을 생략하고 0으로만 구성된 그룹의 가장 긴 시퀀스를 하나의 빈 그룹으로 축소됩니다.

이것은 IPv6 주소에 대해 `str(addr)` 가 반환하는 값이기도 합니다.

exploded

주소 표현의 긴 형식, 모든 선행 0과 완전히 0으로 구성된 그룹이 포함됩니다.

다음 어트리뷰트와 메서드에 대해서는, `IPv4Address` 클래스의 해당 설명서를 참조하십시오:

packed

reverse_pointer

version

max_prefixlen

is_multicast

is_private

is_global

Added in version 3.4.

is_unspecified

is_reserved

is_loopback

is_link_local

is_site_local

주소가 사이트 로컬 사용을 위해 예약되었으면 `True`. 사이트 로컬 주소 공간은 **RFC 3879**에서 폐지되었음에 유의하십시오. *is_private*를 사용하여 이 주소가 **RFC 4193**에 의해 정의된 고유한 로컬 주소 공간에 있는지 검사하십시오.

ipv4_mapped

IPv4에 매핑된 주소(`::FFFF/96`으로 시작합니다)로 표시되는 주소의 경우, 이 프로퍼티는 내장 IPv4 주소를 보고합니다. 다른 주소의 경우, 이 프로퍼티는 `None`이 됩니다.

scope_id

RFC 4007에서 정의된 스코프 지정된 주소의 경우, 이 프로퍼티는 주소가 속한 주소 스코프의 특정 존(zone)을 문자열로 식별합니다. 스코프 존이 지정되지 않았으면, 이 프로퍼티는 `None` 입니다.

sixtofour

RFC 3056에서 정의한 6to4 주소(`2002::/16`으로 시작합니다)인 것으로 보이는 주소의 경우, 이 프로퍼티는 내장 IPv4 주소를 보고합니다. 다른 주소의 경우, 이 프로퍼티는 `None`이 됩니다.

teredo

RFC 4380에서 정의한 Teredo 주소(`2001::/32`로 시작합니다)인 것으로 보이는 주소의 경우, 이 프로퍼티는 내장 `(server, client)` IP 주소 쌍을 보고합니다. 다른 주소의 경우, 이 프로퍼티는 `None`이 됩니다.

`IPv6Address.__format__`(*fmt*)

*IPv4Address*의 해당 메서드 설명서를 참조하십시오.

Added in version 3.9.

문자열과 정수로의 변환

socket 모듈과 같은 네트워킹 인터페이스와 상호 운용하려면, 주소를 문자열이나 정수로 변환해야 합니다. 이것은 *str()*과 *int()* 내장 함수를 사용하여 처리됩니다:

```
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

IPv6 스코프 지정된 주소는 스코프 존 ID 없이 정수로 변환됨에 유의하십시오.

연산자

주소 객체는 일부 연산자를 지원합니다. 달리 명시하지 않는 한, 연산자는 호환 가능한 객체 간에만 적용할 수 있습니다 (즉 IPv4와 IPv4, IPv6와 IPv6).

비교 연산자

주소 객체는 일반적인 비교 연산자 집합으로 비교할 수 있습니다. 다른 스코프 존(scope zone) ID를 가진 같은 IPv6 주소는 같지 않습니다. 몇 가지 예:

```
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
>>> IPv6Address('fe80::1234') == IPv6Address('fe80::1234%1')
False
>>> IPv6Address('fe80::1234%1') != IPv6Address('fe80::1234%2')
True
```

산술 연산자

주소 객체에 정수를 더하거나 뺄 수 있습니다. 몇 가지 예:

```
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not permitted as an IPv4␣
↪address
```

22.23.3 IP 네트워크 정의

`IPv4Network` 와 `IPv6Network` 객체는 IP 네트워크 정의를 정의하고 검사하기 위한 메커니즘을 제공합니다. 네트워크 정의는 마스크(*mask*)와 네트워크 주소(*network address*)로 구성되며, 마스크로 마스크 (바이너리 AND) 될 때 네트워크 주소와 같은 IP 주소 범위를 정의합니다. 예를 들어□ 마스크 `255.255.255.0` 과 네트워크 주소 `192.168.1.0`으로 구성된 네트워크 정의는 `192.168.1.0`에서 `192.168.1.255`까지의 IP 주소로 구성됩니다.

접두사, 네트 마스크 및 호스트 마스크

IP 네트워크 마스크를 지정하는 몇 가지 동등한 방법이 있습니다. 접두사(*prefix*) `/<nbits>`는 네트워크 마스크에 설정된 상위 비트 수를 나타내는 표기법입니다. 네트 마스크(*net mask*)는 몇 개의 상위 비트가 설정된 IP 주소입니다. 따라서 접두사 `/24`는 IPv4에서 네트 마스크 `255.255.255.0`, 또는 IPv6에서 `ffff:ff00::` 와 동등합니다. 또한 호스트 마스크(*host mask*)는 네트 마스크(*net mask*)의 논리적 역전이며, 때로 네트워크 마스크를 나타내기 위해 (예를 들어 Cisco 접근 제어 목록에서) 사용됩니다. IPv4에서 `/24`에 해당하는 호스트 마스크는 `0.0.0.255`입니다.

네트워크 객체

주소 객체가 구현하는 모든 어트리뷰트는 네트워크 객체도 구현합니다. 또한, 네트워크 객체는 추가 어트리뷰트를 구현합니다. 이 모든 것은 `IPv4Network` 와 `IPv6Network` 사이에서 공통이라서, 중복을 피하고자 `IPv4Network` 에 대해서만 설명됩니다. 네트워크 객체는 해시 가능해서 딕셔너리에서 키로 사용할 수 있습니다.

class ipaddress.**IPv4Network**(*address*, *strict=True*)

IPv4 네트워크 정의를 구성합니다. *address*는 다음 중 하나일 수 있습니다:

1. IP 주소와 선택적 마스크로 구성되고, 슬래시(/)로 구분된 문자열. IP 주소는 네트워크 주소이며, 마스크는 접두사를 뜻하는 단일 숫자이거나 IPv4 주소의 문자열 표현일 수 있습니다. 후자의

경우, 마스크는 0이 아닌 필드로 시작하면 네트 마스크로, 또는 0으로 시작하면 호스트 마스크로 해석되는데, 네트 마스크로 취급되는 모든 0인 마스크만 유일한 예외입니다. 마스크가 제공되지 않으면, `/32`로 간주합니다.

예를 들어, 다음 *address* 명세는 동등합니다: `192.168.1.0/24`, `192.168.1.0/255.255.255.0` 및 `192.168.1.0/0.0.0.255`.

2. 32비트에 맞는 정수. 이는 네트워크 주소가 *address*이고 마스크가 `/32`인 단일 주소 네트워크와 동등합니다.
3. 길이가 4인 `bytes` 객체에 채워진 정수, 빅 엔디안. 해석은 정수 *address*와 유사합니다.
4. A two-tuple of an address description and a netmask, where the address description is either a string, a 32-bits integer, a 4-bytes packed integer, or an existing `IPv4Address` object; and the netmask is either an integer representing the prefix length (e.g. `24`) or a string representing the prefix mask (e.g. `255.255.255.0`).

*address*가 유효한 IPv4 주소가 아니면 `AddressValueError` 가 발생합니다. 마스크가 IPv4 주소에 유효하지 않으면 `NetmaskValueError` 가 발생합니다.

*strict*가 `True`이고 제공된 address에 호스트 비트가 설정되면, `ValueError`가 발생합니다. 그렇지 않으면, 적절한 네트워크 주소를 결정하기 위해 호스트 비트가 마스크 되어 제거됩니다.

달리 명시되지 않는 한, 다른 네트워크/주소 객체를 받아들이는 모든 네트워크 메서드는 인자의 IP 버전이 `self`와 호환되지 않으면 `TypeError`를 발생시킵니다.

버전 3.5에서 변경: *address* 생성자 매개 변수에 대해 2-튜플 형식을 추가했습니다.

version

max_prefixlen

`IPv4Address` 의 해당 어트리뷰트 설명서를 참조하십시오.

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

이 어트리뷰트들은 네트워크 주소와 브로드 캐스트 주소 모두에 대해 참이면 네트워크 전체에 대해 참입니다.

network_address

네트워크의 네트워크 주소. 네트워크 주소와 접두사 길이는 네트워크를 고유하게 정의합니다.

broadcast_address

네트워크의 브로드 캐스트 주소. 브로드 캐스트 주소로 전송된 패킷은 네트워크의 모든 호스트가 수신해야 합니다.

hostmask

`IPv4Address` 객체로 제공되는 호스트 마스크.

netmask

`IPv4Address` 객체로 제공되는 네트 마스크.

with_prefixlen

compressed

exploded

접두사 표기법의 마스크를 갖는 네트워크의 문자열 표현.

`with_prefixlen`과 `compressed`는 항상 `str(network)`와 같습니다. `exploded`는 펼쳐진 형식의 네트워크 주소를 사용합니다.

with_netmask

네트 마스크 표기법의 마스크를 갖는 네트워크의 문자열 표현.

with_hostmask

호스트 마스크 표기법의 마스크를 갖는 네트워크의 문자열 표현.

num_addresses

네트워크의 총 주소 수.

prefixlen

네트워크 접두사 길이, 비트 단위.

hosts()

네트워크에서 사용 가능한 호스트에 대한 이터레이터를 반환합니다. 사용 가능한 호스트는 네트워크 주소 자체와 네트워크 브로드 캐스트 주소를 제외하고, 네트워크에 속하는 IP 주소입니다. 마스크 길이가 31인 네트워크의 경우, 네트워크 주소와 네트워크 브로드 캐스트 주소도 결과에 포함됩니다. 마스크가 32인 네트워크는 단일 호스트 주소가 포함된 리스트를 반환합니다.

```
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
>>> list(ip_network('192.0.2.0/31').hosts())
[IPv4Address('192.0.2.0'), IPv4Address('192.0.2.1')]
>>> list(ip_network('192.0.2.1/32').hosts())
[IPv4Address('192.0.2.1')]
```

overlaps(*other*)

이 네트워크가 *other*에 부분적으로나 전체적으로 포함되어 있거나 *other*가 이 네트워크에 완전히 포함되면 `True`.

address_exclude(*network*)

지정된 *network*를 이 네트워크에서 제거하여 만들어지는 네트워크 정의를 계산합니다. 네트워크 객체의 이터레이터를 반환합니다. 이 네트워크에 *network*가 완전히 포함되지 않으면 `ValueError`가 발생합니다.

```
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))
[IPv4Network('192.0.2.8/29'), IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'), IPv4Network('192.0.2.0/32')]
```

subnets(*prefixlen_diff=1*, *new_prefix=None*)

인자 값에 따라, 현재 네트워크 정의를 만들기 위해 참여하는 서브 네트들. *prefixlen_diff*는 우리의 접두사 길이를 늘여야 하는 양입니다. *new_prefix*는 서브 네트의 원하는 새 접두사입니다; 우리의 접두사보다 커야 합니다. *prefixlen_diff*와 *new_prefix* 중 하나만 설정해야 합니다. 네트워크 객체의 이터레이터를 반환합니다.

```
>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
>>> list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=26))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError('new prefix must be longer')
ValueError: new prefix must be longer
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
```

supernet(*prefixlen_diff=1*, *new_prefix=None*)

인자 값에 따라, 이 네트워크 정의를 포함하는 슈퍼 네트. *prefixlen_diff*는 우리의 접두사 길이를 줄여야 하는 양입니다. *new_prefix*는 슈퍼 네트의 원하는 새 접두사입니다; 우리의 접두사보다 작아야 합니다. *prefixlen_diff*와 *new_prefix* 중 하나만 설정해야 합니다. 단일 네트워크 객체를 반환합니다.

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')
```

subnet_of(*other*)

이 네트워크가 *other*의 서브 네트이면 `True`를 반환합니다.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> b.subnet_of(a)
True
```

Added in version 3.7.

supernet_of(*other*)

이 네트워크가 *other*의 슈퍼 네트이면 `True`를 반환합니다.

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> a.supernet_of(b)
True
```

Added in version 3.7.

compare_networks(*other*)

이 네트워크를 *other*와 비교합니다. 이 비교에서는 네트워크 주소 만 고려됩니다; 호스트 비트는 고려하지 않습니다. `-1`, `0` 또는 `1`을 반환합니다.

```
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.2/32'))
-1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.0/32'))
1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.1/32'))
0
```

버전 3.7부터 폐지됨: "<", "==" 및 ">"와 같은 순서와 비교 알고리즘을 사용합니다.

class ipaddress.**IPv6Network**(*address*, *strict=True*)

IPv6 네트워크 정의를 구성합니다. *address*는 다음 중 하나일 수 있습니다:

1. IP 주소와 선택적 접두사 길이로 구성되고 슬래시(/)로 구분된 문자열. IP 주소는 네트워크 주소이며, 접두사 길이는 단일 숫자인 접두사여야 합니다. 접두사 길이가 제공되지 않으면, `/128`로 간주합니다.

   현재 확장된 네트 마스크는 지원되지 않음에 유의하십시오. `2001:db00::0/24`는 유효한 인자이지만 `2001:db00::0/ffff:ff00::`는 아니라는 뜻입니다.

2. 128비트에 맞는 정수. 이는 네트워크 주소가 *address*이고 마스크가 `/128`인 단일 주소 네트워크와 동등합니다.

3. 길이가 16인 `bytes` 객체에 채워진 정수, 빅 엔디안. 해석은 정수 *address*와 유사합니다.

4. A two-tuple of an address description and a netmask, where the address description is either a string, a 128-bits integer, a 16-bytes packed integer, or an existing `IPv6Address` object; and the netmask is an integer representing the prefix length.

*address*가 유효한 IPv6 주소가 아니면 `AddressValueError` 가 발생합니다. 마스크가 IPv6 주소에 유효하지 않으면 `NetmaskValueError` 가 발생합니다.

*strict*가 `True`이고 제공된 address에 호스트 비트가 설정되면, `ValueError`가 발생합니다. 그렇지 않으면, 적절한 네트워크 주소를 결정하기 위해 호스트 비트가 마스크 되어 제거됩니다.

버전 3.5에서 변경: *address* 생성자 매개 변수에 대해 2-튜플 형식을 추가했습니다.

version

max_prefixlen

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

network_address

broadcast_address

hostmask

netmask

with_prefixlen

compressed

exploded

with_netmask

with_hostmask

num_addresses

prefixlen

hosts()

네트워크에서 사용 가능한 호스트에 대한 이터레이터를 반환합니다. 사용 가능한 호스트는 서브 네트 라우터 애니 캐스트 주소를 제외하고 네트워크에 속하는 모든 IP 주소입니다. 마스크 길이가 127인 네트워크의 경우, 서브 네트 라우터 애니 캐스트 주소도 결과에 포함됩니다. 마스크가 128인 네트워크는 단일 호스트 주소가 포함된 리스트를 반환합니다.

overlaps(*other*)

address_exclude(*network*)

subnets(*prefixlen_diff=1*, *new_prefix=None*)

supernet(*prefixlen_diff=1*, *new_prefix=None*)

subnet_of(*other*)

supernet_of(*other*)

compare_networks(*other*)

`IPv4Network` 의 해당 어트리뷰트 설명서를 참조하십시오.

is_site_local

This attribute is true for the network as a whole if it is true for both the network address and the broadcast address.

연산자

네트워크 객체는 일부 연산자를 지원합니다. 달리 명시하지 않는 한, 연산자는 호환 가능한 객체 간에만 적용할 수 있습니다 (즉 IPv4와 IPv4, IPv6와 IPv6).

논리 연산자

네트워크 객체는 일반적인 논리 연산자 집합으로 비교할 수 있습니다. 네트워크 객체는 먼저 네트워크 주소로 정렬된 다음, 네트 마스크로 정렬됩니다.

이터레이션

네트워크에 속하는 모든 주소를 나열하기 위해 네트워크 객체를 이터레이트 할 수 있습니다. 이터레이션의 경우, 사용 불가능한 호스트를 포함하여 모든 호스트가 반환됩니다 (사용 가능한 호스트의 경우는 `hosts()` 메서드를 사용하십시오). 예:

```
>>> for addr in IPv4Network('192.0.2.0/28'):
...     addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

주소 컨테이너로서의 네트워크

네트워크 객체는 주소의 컨테이너 역할을 할 수 있습니다. 몇 가지 예:

```
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

22.23.4 인터페이스 객체

인터페이스 객체는 해시 가능해서 딕셔너리에서 키로 사용할 수 있습니다.

class ipaddress.**IPv4Interface**(*address*)

IPv4 인터페이스를 구성합니다. *address*의 의미는 임의의 호스트 주소가 항상 허용된다는 점을 제외하고는 *IPv4Network* 의 생성자와 같습니다.

*IPv4Interface*는 *IPv4Address* 의 서브 클래스이기 때문에, 그 클래스의 모든 어트리뷰트를 상속합니다. 또한, 다음과 같은 어트리뷰트를 사용할 수 있습니다:

ip

네트워크 정보가 없는 주소 (*IPv4Address*).

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network

이 인터페이스가 속한 네트워크 (*IPv4Network*).

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen

접두사 표기법의 마스크를 갖는 인터페이스의 문자열 표현.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask

네트워크를 네트 마스크로 사용하는 인터페이스의 문자열 표현.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask

네트워크를 호스트 마스크로 사용하는 인터페이스의 문자열 표현.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class ipaddress.**IPv6Interface**(*address*)

IPv6 인터페이스를 구성합니다. *address*의 의미는 임의의 호스트 주소가 항상 허용된다는 점을 제외하고는 `IPv6Network` 의 생성자와 같습니다.

`IPv6Interface`는 `IPv6Address` 의 서브 클래스이기 때문에, 그 클래스의 모든 어트리뷰트를 상속합니다. 또한, 다음과 같은 어트리뷰트를 사용할 수 있습니다:

ip

network

with_prefixlen

with_netmask

with_hostmask

`IPv4Interface` 의 해당 어트리뷰트 설명서를 참조하십시오.

연산자

인터페이스 객체는 일부 연산자를 지원합니다. 달리 명시하지 않는 한, 연산자는 호환 가능한 객체 간에만 적용할 수 있습니다 (즉 IPv4와 IPv4, IPv6와 IPv6).

논리 연산자

인터페이스 객체는 일반적인 논리 연산자 집합으로 비교할 수 있습니다.

동등 비교(==과 !=)의 경우, 같다고 비교되려면 IP 주소와 네트워크가 같아야 합니다. 인터페이스는 주소나 네트워크 객체와 같다고 비교되지 않습니다.

순서(<, > 등)의 경우 규칙이 다릅니다. 같은 IP 버전의 인터페이스와 주소 객체를 비교할 수 있으며, 주소 객체는 항상 인터페이스 객체보다 앞에 정렬됩니다. 두 개의 인터페이스 객체는 먼저 네트워크로 비교되고, 같으면 IP 주소로 비교됩니다.

22.23.5 다른 모듈 수준 함수

이 모듈은 다음과 같은 모듈 수준 함수도 제공합니다:

ipaddress.**v4_int_to_packed**(*address*)

네트워크 (빅 엔디안) 순서로 채워진 길이 4인 바이트열로 주소를 표현합니다. *address*는 IPv4 IP 주소의 정수 표현입니다. 정수가 음수이거나 너무 커서 IPv4 IP 주소가 될 수 없으면 `ValueError`가 발생합니다.

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

ipaddress.**v6_int_to_packed**(*address*)

네트워크 (빅 엔디안) 순서로 채워진 길이 16인 바이트열로 주소를 나타냅니다. *address*는 IPv6 IP 주소의 정수 표현입니다. 정수가 음수이거나 너무 커서 IPv6 IP 주소가 될 수 없으면 `ValueError`가 발생합니다.

ipaddress.**summarize_address_range**(*first*, *last*)

첫 번째와 마지막 IP 주소가 주어진 요약된 네트워크 범위의 이터레이터를 반환합니다. *first*는 범위의 첫 번째 `IPv4Address` 나 `IPv6Address` 이고 *last*는 범위의 마지막 `IPv4Address` 나 `IPv6Address`입니다. *first*나 *last*가 IP 주소가 아니거나, 같은 버전이 아니면 `TypeError`가 발생합니다. *last*가 *first*보다 크지 않거나 *first* 주소 버전이 4나 6이 아니면 `ValueError`가 발생합니다.

```
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(
...     ipaddress.IPv4Address('192.0.2.0'),
...     ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.
↪0.2.130/32')]
```

ipaddress.**collapse_addresses**(*addresses*)

축약된 *IPv4Network* 나 *IPv6Network* 객체의 이터레이터를 반환합니다. *addresses*는 *IPv4Network* 나 *IPv6Network* 객체의 이터러블입니다. *addresses*에 혼합 버전 객체가 포함되어 있으면 *TypeError*가 발생합니다.

```
>>> [ipaddr for ipaddr in
... ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'),
... ipaddress.IPv4Network('192.0.2.128/25')])]
[IPv4Network('192.0.2.0/24')]
```

ipaddress.**get_mixed_type_key**(*obj*)

네트워크와 주소를 정렬하기에 적합한 키를 반환합니다. 주소와 네트워크 객체는 기본적으로 정렬할 수 없습니다; 이들은 근본적으로 달라서, 다음과 같은 표현은:

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

의미가 없습니다. 그러나 *ipaddress*가 어쨌든 정렬하도록 할 때가 있습니다. 이렇게 해야 하면, 이 함수를 *sorted()*의 *key* 인자로 사용할 수 있습니다.

*obj*는 네트워크나 주소 객체입니다.

22.23.6 맞춤 예외

클래스 생성자의 더욱 구체적인 에러 보고를 지원하기 위해, 모듈은 다음 예외를 정의합니다:

exception ipaddress.**AddressValueError**(*ValueError*)

주소와 관련된 모든 값 에러.

exception ipaddress.**NetmaskValueError**(*ValueError*)

네트 마스크와 관련된 모든 값 에러.

CHAPTER 23

멀티미디어 서비스

이 장에서 설명하는 모듈은 주로 멀티미디어 응용 프로그램에 유용한 다양한 알고리즘 또는 인터페이스를 구현합니다. 가용성은 설치에 달려있습니다. 다음은 개요입니다:

23.1 wave — WAV 파일 읽고 쓰기

소스 코드: Lib/wave.py

---

wave 모듈은 파형 오디오 "WAVE" (또는 "WAV") 파일 형식에 편리한 인터페이스를 제공합니다. 압축되지 않은 PCM 인코딩 웨이브 파일만 지원합니다.

버전 3.12에서 변경: Support for `WAVE_FORMAT_EXTENSIBLE` headers was added, provided that the extended format is `KSDATAFORMAT_SUBTYPE_PCM`.

wave 모듈은 다음 함수와 예외를 정의합니다:

`wave.`**`open`**(*file*, *mode=None*)

*file*이 문자열이면, 그 이름의 파일을 엽니다, 그렇지 않으면 파일류 객체로 처리합니다. *mode*는 다음 중 하나일 수 있습니다:

'rb'
읽기 전용 모드.

'wb'
쓰기 전용 모드.

WAV 파일의 읽기와 쓰기를 동시에 허락하지 않음에 유의하십시오.

`'rb'` *mode*는 *Wave_read* 객체를 반환하고, `'wb'` *mode*는 *Wave_write* 객체를 반환합니다. *mode*가 생략되고, 파일류 객체가 *file*로 전달되면, `file.mode`가 *mode*의 기본값으로 사용됩니다.

파일류 객체를 전달하면, `close()` 메서드가 호출될 때 wave 객체는 그 파일을 닫지 않습니다. 파일 객체를 닫는 것은 호출자의 책임입니다.

open() 함수는 `with` 문과 함께 사용할 수 있습니다. `with` 블록이 완료될 때, *Wave_read.close()*나 *Wave_write.close()* 메서드가 호출됩니다.

버전 3.4에서 변경: 위치 변경할 수 없는(unseekable) 파일에 대한 지원이 추가되었습니다.

exception `wave.`**`Error`**

WAV 명세를 위반하거나 구현 결함으로 인해 무언가가 불가능할 때 발생하는 에러.

23.1.1 Wave_read 객체

class wave.**Wave_read**

WAV 파일을 읽습니다.

open() 이 반환하는, Wave_read 객체는 다음과 같은 메서드를 가지고 있습니다:

close()

스트림이 *wave*에 의해 열렸다면 스트림을 닫고, 인스턴스를 사용할 수 없게 만듭니다. 이것은 객체가 가비지 수집될 때 자동으로 호출됩니다.

getnchannels()

오디오 채널 수를 반환합니다 (모노는 `1`, 스테레오는 `2`).

getsampwidth()

샘플 폭을 바이트 단위로 반환합니다.

getframerate()

샘플링 빈도를 반환합니다.

getnframes()

오디오 프레임의 수를 반환합니다.

getcomptype()

압축 유형을 반환합니다 (지원되는 유형은 `'NONE'` 뿐입니다).

getcompname()

*getcomptype()*의 사람이 읽을 수 있는 버전. 보통 `'not compressed'`이 `'NONE'`에 해당합니다.

getparams()

`get*()` 메서드의 결과와 동등한, *namedtuple()* `(nchannels, sampwidth, framerate, nframes, comptype, compname)`를 반환합니다.

readframes(*n*)

최대 *n* 프레임의 오디오를 *bytes* 객체로 읽고 반환합니다.

rewind()

파일 포인터를 오디오 스트림의 시작 부분으로 되감습니다.

다음의 두 메서드는 이전 `aifc` 모듈과의 호환성을 위해 정의되었으며, 흥미로운 작업을 수행하지 않습니다.

getmarkers()

`None`을 반환합니다.

Deprecated since version 3.13, will be removed in version 3.15: 이 메서드는 파이썬 3.13에서 게거된 `aifc` 모듈과의 호환성을 위해서만 존재했습니다.

getmark(*id*)

에러를 발생시킵니다.

Deprecated since version 3.13, will be removed in version 3.15: 이 메서드는 파이썬 3.13에서 게거된 `aifc` 모듈과의 호환성을 위해서만 존재했습니다.

다음의 두 메서드는 이들 사이에서 호환 가능한 용어 "위치(position)"를 정의하며, 그 외에는 구현에 따라 다릅니다.

setpos(*pos*)

파일 포인터를 지정된 위치로 설정합니다.

tell()

현재 파일 포인터 위치를 반환합니다.

23.1.2 Wave_write 객체

class wave.**Wave_write**

WAV 파일을 기록합니다.

*open()*에 의해 반환된 Wave_write 객체에.

위치 변경할 수 있는(seekable) 출력 스트림의 경우, 실제로 기록된 프레임 수를 반영하도록 wave 헤더가 자동으로 갱신됩니다. 위치 변경할 수 없는(unseekable) 스트림의 경우, 첫 번째 프레임 데이터를 쓸 때, *nframes* 값이 정확해야 합니다. 정확한 *nframes* 값을 실현하려면 *close()*가 호출되기 전에 기록될 프레임 수로 *setnframes()*나 *setparams()*를 호출한 다음, *writeframesraw()*를 사용하여 프레임 데이터를 쓰거나, 기록할 모든 프레임 데이터로 *writeframes()*를 호출할 수 있습니다. 후자의 경우 *writeframes()*는 데이터의 프레임 수를 계산하고 프레임 데이터를 기록하기 전에 적절한 *nframes*를 설정합니다.

버전 3.4에서 변경: 위치 변경할 수 없는(unseekable) 파일에 대한 지원이 추가되었습니다.

Wave_write 객체에는 다음과 같은 메서드가 있습니다:

close()

*nframes*를 올바르게 만들고, 파일이 *wave*로 열렸으면 파일을 닫습니다. 이 메서드는 객체가 가비지 수집될 때 호출됩니다. 출력 스트림이 위치 변경할 수 없고 *nframes*가 실제로 기록된 프레임 수와 일치하지 않으면 예외를 일으킵니다.

setnchannels(*n*)

채널 수를 설정합니다.

setsampwidth(*n*)

샘플 폭을 *n* 바이트로 설정합니다.

setframerate(*n*)

프레임 속도를 *n*으로 설정합니다.

버전 3.2에서 변경: 이 메서드에 대한 비 정수 입력은 가장 가까운 정수로 자리 올림 됩니다.

setnframes(*n*)

프레임 수를 *n*으로 설정합니다. 실제로 쓴 프레임 수가 다르면 나중에 변경됩니다 (이 변경 시도는 출력 스트림이 위치 변경할 수 없으면 에러를 발생시킵니다).

setcomptype(*type*, *name*)

압축 유형과 설명을 설정합니다. 현재, 압축 유형 NONE 만 지원됩니다. 즉, 압축하지 않습니다.

setparams(*tuple*)

*tuple*은 (nchannels, sampwidth, framerate, nframes, comptype, compname)이어야 하며, set*() 메서드에 유효한 값이어야 합니다. 모든 파라미터를 설정합니다.

tell()

파일의 현재 위치를 반환하는데, *Wave_read.tell()*과 *Wave_read.setpos()* 메서드와 같은 면책 조항이 적용됩니다.

writeframesraw(*data*)

*nframes*를 수정하지 않고 오디오 프레임을 씁니다.

버전 3.4에서 변경: 이제 모든 바이트열류 객체가 허락됩니다.

writeframes(*data*)

오디오 프레임을 기록하고 *nframes*를 올바르게 만듭니다. 출력 스트림이 위치 변경할 수 없고 *data*를 기록한 후에 기록된 총 프레임 수가 *nframes*에 대해 이전에 설정된 값과 일치하지 않으면 에러가 발생시킵니다.

버전 3.4에서 변경: 이제 모든 바이트열류 객체가 허락됩니다.

*writeframes()*나 *writeframesraw()*를 호출한 후 파라미터를 설정하는 것이 유효하지 않고, 이를 시도하면 *wave.Error*가 발생함에 유의하십시오.

23.2 colorsys — 색 체계 간의 변환

소스 코드: Lib/colorsys.py

---

colorsys 모듈은 컴퓨터 모니터에서 사용되는 RGB (Red Green Blue, 적 녹 청) 색 공간과 세 가지 다른 좌표계 YIQ, HLS (Hue Lightness Saturation, 색상 명도 채도), HSV(Hue Saturation Value, 색상 채도 명도)로 표현된 색 간 양방향 색 변환을 정의합니다. 이러한 모든 색 공간의 좌표는 부동 소수점 값입니다. YIQ 공간에서, Y 좌표는 0과 1사이지만, I 와 Q 좌표는 양수나 음수가 될 수 있습니다. 다른 모든 공간에서, 좌표는 모두 0과 1 사이입니다.

> **더 보기**
>
> 색 공간에 대한 자세한 내용은 https://poynton.ca/ColorFAQ.html 와 https://www.cambridgeincolour.com/tutorials/color-spaces.htm 에서 확인할 수 있습니다.

colorsys 모듈은 다음 함수를 정의합니다:

colorsys.**rgb_to_yiq**(*r*, *g*, *b*)
RGB 좌표에서 YIQ 좌표로 색을 변환합니다.

colorsys.**yiq_to_rgb**(*y*, *i*, *q*)
YIQ 좌표에서 RGB 좌표로 색을 변환합니다.

colorsys.**rgb_to_hls**(*r*, *g*, *b*)
RGB 좌표에서 HLS 좌표로 색을 변환합니다.

colorsys.**hls_to_rgb**(*h*, *l*, *s*)
HLS 좌표에서 RGB 좌표로 색을 변환합니다.

colorsys.**rgb_to_hsv**(*r*, *g*, *b*)
RGB 좌표에서 HSV 좌표로 색을 변환합니다.

colorsys.**hsv_to_rgb**(*h*, *s*, *v*)
HSV 좌표에서 RGB 좌표로 색을 변환합니다.

예:

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

CHAPTER 24

국제화

이 장에서 설명하는 모듈은 프로그램 메시지에 사용할 언어를 선택하거나 현지 규칙에 맞게 출력을 조정할 수 있는 메커니즘을 제공하여 언어 및 로케일에 종속되지 않는 소프트웨어를 작성하는 데 도움이 됩니다.

이 장에서 설명하는 모듈 목록은 다음과 같습니다:

24.1 gettext — 다국어 국제화 서비스

소스 코드: Lib/gettext.py

---

gettext 모듈은 파이썬 모듈과 응용 프로그램을 위한 국제화(I18N)와 현지화(L10N) 서비스를 제공합니다. GNU **gettext** 메시지 카탈로그 API와 파이썬 파일에 더 적합한 고수준 클래스 기반 API를 모두 지원합니다. 아래 설명된 인터페이스를 사용하면 모듈과 응용 프로그램 메시지를 하나의 자연어로 작성하고, 다른 자연어로 실행하기 위해 번역된 메시지 카탈로그를 제공할 수 있습니다.

파이썬 모듈과 응용 프로그램을 현지화하는 데 대한 힌트도 제공됩니다.

24.1.1 GNU gettext API

gettext 모듈은 GNU **gettext** API와 매우 유사한 다음 API를 정의합니다. 이 API를 사용하면 전체 응용 프로그램의 번역에 전역적으로 영향을 미칩니다. 응용 프로그램이 단일 언어라면 사용자의 로케일에 따라 언어를 선택할 수 있는 것과 함께 종종 이것이 여러분이 원하는 것입니다. 파이썬 모듈을 현지화하거나, 응용 프로그램에서 언어를 실행 중에 전환해야 한다면, 아마도 클래스 기반 API를 대신 사용하고 싶을 것입니다.

gettext.**bindtextdomain**(*domain*, *localedir=None*)

*domain*을 로케일 디렉터리 *localedir*에 바인드합니다. 보다 구체적으로, *gettext*는 경로 (유닉스에서) *localedir*/*language*/LC_MESSAGES/*domain*.mo를 사용하여 지정된 도메인(domain)에 대한 바이너리 .mo 파일을 찾습니다. 여기서 *language*는 환경 변수 LANGUAGE, LC_ALL, LC_MESSAGES 및 LANG에서 각각 검색됩니다.

*localedir*이 생략되거나 None이면, *domain*에 대한 현재 바인딩이 반환됩니다.[1]

[1] 기본 로케일 디렉터리는 시스템에 따라 다릅니다; 예를 들어 Red Hat 리눅스에서는 /usr/share/locale이지만, Solaris에서는 /usr/lib/locale입니다. gettext 모듈은 이러한 시스템 종속 기본값을 지원하려고 하지 않습니다; 대신 기본값은 *sys.base_prefix*/share/locale입니다 (*sys.base_prefix*를 참조하십시오). 이런 이유로, 항상 응용 프로그램 시작 시 명시적 절대 경로를 사용하여 *bindtextdomain()*을 호출하는 것이 가장 좋습니다.

gettext.**textdomain**(*domain=None*)

현재 전역 도메인을 변경하거나 조회합니다. *domain*이 None이면, 현재 전역 도메인이 반환되고, 그렇지 않으면 전역 도메인이 *domain*으로 설정되어 반환됩니다.

gettext.**gettext**(*message*)

현재 전역 도메인, 언어 및 로케일 디렉터리를 기반으로, *message*의 현지화 된 번역을 반환합니다. 이 함수는 일반적으로 지역 이름 공간에서 _() 로 별칭이 지정됩니다 (아래 예를 참조하십시오).

gettext.**dgettext**(*domain, message*)

*gettext()*와 비슷하지만, 지정된 *domain*에서 메시지를 찾습니다.

gettext.**ngettext**(*singular, plural, n*)

*gettext()*와 비슷하지만, 복수형 (plural forms)을 고려합니다. 번역이 발견되면, 복수 공식을 *n*에 적용하고, 결과 메시지를 반환합니다 (일부 언어는 복수형이 두 개 이상입니다). 번역이 없으면, *n*이 1이면 *singular*를 반환합니다; 그렇지 않으면 *plural*을 반환합니다.

복수 공식은 카탈로그 헤더에서 취합니다. 자유 변수 *n*을 갖는 C나 파이썬 표현식입니다. 이 표현식은 카탈로그에서 복수의 인덱스로 평가됩니다. .po 파일에 사용되는 정확한 문법과 다양한 언어의 공식은 GNU gettext 설명서를 참조하십시오.

gettext.**dngettext**(*domain, singular, plural, n*)

*ngettext()*와 비슷하지만, 지정된 *domain*에서 메시지를 찾습니다.

gettext.**pgettext**(*context, message*)

gettext.**dpgettext**(*domain, context, message*)

gettext.**npgettext**(*context, singular, plural, n*)

gettext.**dnpgettext**(*domain, context, singular, plural, n*)

접두사에 p가 없는 해당 함수(즉, *gettext()*, *dgettext()*, *ngettext()*, *dngettext()*)와 유사하지만, 번역은 지정된 메시지 *context*로 제한됩니다.

Added in version 3.8.

GNU **gettext**가 dcgettext() 메서드도 정의하지만, 이것을 유용하지 않은 것으로 간주해서 현재 구현되지 않았음에 유의하십시오.

이 API의 일반적인 사용 예는 다음과 같습니다:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```

24.1.2 클래스 기반 API

gettext 모듈의 클래스 기반 API는 GNU **gettext** API보다 더 많은 유연성과 편리성을 제공합니다. 파이썬 응용 프로그램과 모듈을 현지화하는 권장되는 방법입니다. gettext는 GNU .mo 형식 파일의 구문 분석을 구현하고 문자열을 반환하는 메서드가 있는 *GNUTranslations* 클래스를 정의합니다. 이 클래스의 인스턴스는 내장 이름 공간에 함수 _() 로 자신을 설치할 수도 있습니다.

gettext.**find**(*domain, localedir=None, languages=None, all=False*)

이 함수는 표준 .mo 파일 검색 알고리즘을 구현합니다. *textdomain()*이 취하는 것과 동일한 *domain*을 취합니다. 선택적 *localedir*은 *bindtextdomain()*에서와 같습니다. 선택적 *languages*는 문자열 리스트이며, 각 문자열은 언어 코드입니다.

*localedir*이 제공되지 않으면, 기본 시스템 로케일 디렉터리가 사용됩니다.[2] *languages*가 제공되지 않으면, 다음과 같은 환경 변수가 검색됩니다: LANGUAGE, LC_ALL, LC_MESSAGES 및 LANG. 비어 있지

[2] 위의 *bindtextdomain()*에 대한 각주를 참조하십시오.

않은 값을 반환하는 첫 번째 것이 *languages* 변수에 사용됩니다. 환경 변수는 콜론으로 구분된 언어 목록을 포함해야 하며, 콜론에서 분할되어 예상되는 언어 코드 문자열 리스트를 생성합니다.

그런 다음 `find()`는 언어를 확장하고 정규화한 다음, 다음 구성 요소로 구성된 기존 파일을 검색하면서, 이들을 이터레이트 합니다:

localedir/*language*/LC_MESSAGES/*domain*.mo

존재하는 첫 번째 파일 이름이 `find()`에 의해 반환됩니다. 그러한 파일이 없으면, `None`이 반환됩니다. *all*이 제공되면, 언어 리스트나 환경 변수에 나타나는 순서대로 모든 파일 이름의 리스트를 반환합니다.

gettext.**translation**(*domain*, *localedir=None*, *languages=None*, *class_=None*, *fallback=False*)

domain, *localedir* 및 *languages*를 기반으로 하는 `*Translations` 인스턴스를 반환합니다. 이 인스턴스는 연관된 `.mo` 파일 경로 리스트를 얻기 위해 먼저 `find()`로 전달됩니다. 동일한 `.mo` 파일 이름을 갖는 인스턴스는 캐시 됩니다. 인스턴스화되는 실제 클래스는 제공된다면 *class_*이고, 그렇지 않으면 `GNUTranslations`입니다. 클래스의 생성자는 단일 파일 객체 인자를 취해야 합니다.

여러 파일이 발견되면, 이후 파일은 이전 파일에 대한 폴 백으로 사용됩니다. 폴 백을 설정하는 것을 허락하기 위해, `copy.copy()`를 사용하여 캐시에서 각 번역 객체를 복제합니다; 실제 인스턴스 데이터는 여전히 캐시와 공유됩니다.

`.mo` 파일이 없으면, 이 함수는 *fallback*이 거짓(기본값)이면 `OSError`를 발생시키고, *fallback*이 참이면 `NullTranslations` 인스턴스를 반환합니다.

버전 3.3에서 변경: 이전에는 `IOError`가 발생했지만, 이제는 `OSError`의 별칭입니다.

버전 3.11에서 변경: *codeset* 매개 변수를 제거했습니다.

gettext.**install**(*domain*, *localedir=None*, **, names=None*)

`translation()`에 전달되는 *domain* 와 *localedir*을 기반으로, 파이썬의 내장 이름 공간에 `_()` 함수를 설치합니다.

names 매개 변수에 대해서는, 번역 객체의 `install()` 메서드에 대한 설명을 참조하십시오.

아래에서 볼 수 있듯이, 일반적으로 다음과 같이 `_()` 함수에 대한 호출로 래핑하여, 응용 프로그램에 있는 번역 후보 문자열을 표시합니다:

```
print(_('This string will be translated.'))
```

편의상, `_()` 함수를 파이썬의 내장 이름 공간에 설치하여, 응용 프로그램의 모든 모듈에서 쉽게 액세스할 수 있도록 합니다.

버전 3.11에서 변경: *names*는 이제 키워드 전용 매개 변수입니다.

`NullTranslations` 클래스

번역 클래스는 원본 소스 파일 메시지 문자열을 번역된 메시지 문자열로 실제로 구현합니다. 모든 번역 클래스에서 사용하는 베이스 클래스는 `NullTranslations`입니다; 여러분 자신의 특수화된 번역 클래스를 작성하는 데 사용할 수 있는 기본 인터페이스를 제공합니다. `NullTranslations`의 메서드는 다음과 같습니다:

class gettext.**NullTranslations**(*fp=None*)

베이스 클래스에서 무시되는, 선택적인 파일 객체 *fp*를 취합니다. 파생 클래스에 의해 설정되는 "보호되는" 인스턴스 변수 *_info*와 *_charset* 뿐만 아니라 `add_fallback()`을 통해 설정되는 *_fallback*을 초기화합니다. 그런 다음 *fp*가 `None`이 아니면 `self._parse(fp)`를 호출합니다.

_parse(*fp*)

베이스 클래스에서 아무런 일도 하지 않는 이 메서드는 파일 객체 *fp*를 취하고, 이 파일에서 데이터를 읽고, 메시지 카탈로그를 초기화합니다. 지원되지 않는 메시지 카탈로그 파일 형식이 있으면, 이 메서드를 재정의하여 형식을 구문 분석해야 합니다.

add_fallback(*fallback*)

현재 번역 객체의 폴 백 객체로 *fallback*을 추가합니다. 주어진 메시지에 대한 번역을 제공할 수 없으면 번역 개체는 폴 백을 참조해야 합니다.

gettext(*message*)

폴 백이 설정되었으면, `gettext()`를 폴 백으로 전달합니다. 그렇지 않으면, *message*를 반환합니다. 파생 클래스에서 재정의됩니다.

ngettext(*singular*, *plural*, *n*)

폴 백이 설정되었으면, `ngettext()`를 폴 백으로 전달합니다. 그렇지 않으면, *n*이 1이면 *singular*를 반환합니다; 그렇지 않으면 *plural*을 반환합니다. 파생 클래스에서 재정의됩니다.

pgettext(*context*, *message*)

폴 백이 설정되었으면, *pgettext()*를 폴 백으로 전달합니다. 그렇지 않으면, 번역된 메시지를 반환합니다. 파생 클래스에서 재정의됩니다.

Added in version 3.8.

npgettext(*context*, *singular*, *plural*, *n*)

폴 백이 설정되었으면, *npgettext()*를 폴 백으로 전달합니다. 그렇지 않으면, 번역된 메시지를 반환합니다. 파생 클래스에서 재정의됩니다.

Added in version 3.8.

info()

메시지 카탈로그 파일에서 발견된 메타 데이터를 포함하는 딕셔너리를 반환합니다.

charset()

메시지 카탈로그 파일의 인코딩을 반환합니다.

install(*names=None*)

이 메서드는 *gettext()*를 내장 이름 공간에 설치하여, `_`에 연결합니다.

names 매개 변수가 제공되면, `_()`에 더해서 내장 이름 공간에 설치하려는 함수 이름이 포함된 시퀀스여야 합니다. 지원되는 이름은 `'gettext'`, `'ngettext'`, `'pgettext'` 및 `'npgettext'`입니다.

이것은 `_()` 함수를 응용 프로그램에서 사용할 수 있게 하는 가장 편리한 방법이지만, 한 가지 방법일 뿐입니다. 전체 응용 프로그램, 특히 내장 이름 공간에 영향을 주기 때문에, 현지화된 모듈은 절대 `_()`를 설치하지 않아야 합니다. 대신, 다음과 같은 코드를 사용하여 `_()`를 모듈에서 사용할 수 있게 해야 합니다:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

이는 `_()`를 모듈의 전역 이름 공간에만 넣기 때문에 이 모듈 내에서의 호출에만 영향을 줍니다.

버전 3.8에서 변경: `'pgettext'`와 `'npgettext'`를 추가했습니다.

GNUTranslations 클래스

`gettext` 모듈은 *NullTranslations*에서 파생된 클래스를 하나 더 제공합니다: *GNUTranslations*. 이 클래스는 `_parse()`를 재정의하여 빅 엔디안과 리틀 엔디안 형식의 GNU **gettext** 형식 `.mo` 파일을 읽을 수 있도록 합니다.

*GNUTranslations*는 번역 카탈로그에서 선택적 메타 데이터를 구문 분석합니다. 빈 문자열의 번역으로 메타 데이터를 포함하는 것이 GNU **gettext**의 관례입니다. 이 메타 데이터는 **RFC 822** 스타일 `key: value` 쌍이며, `Project-Id-Version` 키를 포함해야 합니다. 키 `Content-Type`이 발견되면, `charset` 프로퍼티를 사용하여 "보호된" `_charset` 인스턴스 변수를 초기화하고, 찾을 수 없으면 기본값은 `None`입니다. 문자 집합 인코딩이 지정되면, 카탈로그에서 읽은 모든 메시지 id와 메시지 문자열이 이 인코딩을 사용하여 유니코드로 변환되고, 그렇지 않으면 ASCII로 가정합니다.

메시지 id도 유니코드 문자열로 읽기 때문에, 모든 `*gettext()` 메서드는 메시지 id를 바이트 문자열이 아닌 유니코드 문자열로 가정합니다.

키/값 쌍의 전체 집합이 딕셔너리에 배치되고 "보호된" `_info` 인스턴스 변수로 설정됩니다.

.mo 파일의 매직 번호가 유효하지 않거나, 주 버전 번호가 예상치 못한 값이거나, 파일을 읽는 동안 다른 문제가 발생하면 *GNUTranslations* 클래스를 인스턴스 화할 때 *OSError*가 발생할 수 있습니다.

class gettext.**GNUTranslations**

베이스 클래스 구현에서 다음 메서드가 재정의되었습니다:

gettext(*message*)

카탈로그에서 *message* id를 찾아 해당 메시지 문자열을 유니코드 문자열로 반환합니다. 카탈로그에 *message* id에 대한 항목이 없고, 폴 백이 설정되었으면, 조회는 폴 백의 *gettext()* 메서드로 전달됩니다. 그렇지 않으면, *message* id가 반환됩니다.

ngettext(*singular*, *plural*, *n*)

메시지 id의 복수형 조회를 수행합니다. *singular*는 카탈로그에서 찾기 위해 메시지 id로 사용되는 반면, *n*은 사용할 복수형을 결정하는 데 사용됩니다. 반환된 메시지 문자열은 유니코드 문자열입니다.

카탈로그에서 메시지 id를 찾을 수 없고, 폴 백이 지정되었으면, 요청은 폴 백의 *ngettext()* 메서드로 전달됩니다. 그렇지 않으면, *n*이 1이면 *singular*가 반환되고, 다른 모든 경우에는 *plural*이 반환됩니다.

예를 들면 다음과 같습니다:

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

pgettext(*context*, *message*)

카탈로그에서 *context*와 *message* id를 찾아 해당 메시지 문자열을 유니코드 문자열로 반환합니다. 카탈로그에 *message* id와 *context*에 대한 항목이 없고, 폴 백이 설정되었으면, 조회는 폴 백의 *pgettext()* 메서드로 전달됩니다. 그렇지 않으면, *message* id가 반환됩니다.

Added in version 3.8.

npgettext(*context*, *singular*, *plural*, *n*)

메시지 ID의 복수형 조회를 수행합니다. *singular*는 카탈로그에서 찾기 위해 메시지 id로 사용되는 반면, *n*은 사용할 복수형을 결정하는 데 사용됩니다.

*context*의 메시지 id가 카탈로그에 없고, 폴 백이 지정되었으면, 요청은 폴 백의 *npgettext()* 메서드로 전달됩니다. 그렇지 않으면, *n*이 1이면 *singular*가 반환되고, 다른 모든 경우에는 *plural*이 반환됩니다.

Added in version 3.8.

Solaris 메시지 카탈로그 지원

Solaris 운영 체제는 자체 바이너리 .mo 파일 형식을 정의하지만, 이 형식에 대한 설명서를 찾을 수 없어서, 현재 지원되지 않습니다.

Catalog 생성자

GNOME은 James Henstridge의 *gettext* 모듈 버전을 사용하지만, 이 버전은 API가 약간 다릅니다. 설명된 사용법은 다음과 같습니다:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print(_('hello world'))
```

이 이전 모듈과의 호환성을 위해, 함수 `Catalog()` 는 위에서 설명한 *translation()* 함수의 별칭입니다.

이 모듈과 Henstridge 버전의 한 가지 차이점: 그의 카탈로그 객체는 매핑 API를 통한 액세스를 지원했지만, 사용되지 않는 것으로 보여서 현재 지원되지 않습니다.

24.1.3 프로그램과 모듈의 국제화

국제화(I18N)는 프로그램이 여러 언어를 인식하도록 하는 작업을 말합니다. 현지화(L10N)는 일단 국제화된 프로그램이 현지 언어와 문화적 습관에 적응하는 것을 말합니다. 파이썬 프로그램에 다국어 메시지를 제공하려면, 다음 단계를 수행해야 합니다:

1. 번역 가능한 문자열을 특별히 표시하여 프로그램이나 모듈을 준비합니다
2. 표시된 파일에 대해 도구 모음을 실행하여 원시 메시지 카탈로그를 생성합니다
3. 메시지 카탈로그의 언어별 번역을 만듭니다
4. 메시지 문자열이 올바르게 번역되도록 *gettext* 모듈을 사용합니다

I18N을 위해 여러분의 코드를 준비하려면, 파일의 모든 문자열을 확인해야 합니다. 번역해야 할 모든 문자열은 `_('...')` 로 감싸서 표시해야 합니다 — 즉, 함수 _에 대한 호출. 예를 들면:

```
filename = 'mylog.txt'
message = _('writing a log message')
with open(filename, 'w') as fp:
    fp.write(message)
```

이 예에서, 문자열 `'writing a log message'` 는 번역 후보로 표시되지만, 문자열 `'mylog.txt'` 와 `'w'` 는 그렇지 않습니다.

번역을 위한 문자열을 추출하는 몇 가지 도구가 있습니다. 원래 GNU **gettext** 는 C나 C++ 소스 코드만 지원했지만, 확장 버전 **xgettext** 는 파이썬을 포함하여 여러 언어로 작성된 코드를 스캔하여 번역 가능으로 표시된 문자열을 찾습니다. Babel은 메시지 카탈로그를 추출하고 컴파일하는 `pybabel` 스크립트를 포함하는 파이썬 국제화 라이브러리입니다. **xpot** 이라는 François Pinard의 프로그램도 비슷한 작업을 수행하며 그의 po-utils 패키지의 일부로 제공됩니다.

(파이썬에는 **pygettext.py** 와 **msgfmt.py** 라고 하는 이러한 프로그램의 순수 파이썬 버전도 포함되어 있습니다; 일부 파이썬 배포판은 이 프로그램들을 설치합니다. **pygettext.py** 는 **xgettext** 와 유사하지만, 파이썬 소스 코드만 이해하며 C나 C++ 와 같은 다른 프로그래밍 언어를 처리할 수 없습니다. **pygettext.py** 는 **xgettext** 와 유사한 명령 줄 인터페이스를 지원합니다; 사용에 대한 자세한 내용을 보려면, `pygettext.py --help` 를 실행하십시오. **msgfmt.py** 는 GNU **msgfmt** 와 바이너리 호환됩니다. 이 두 프로그램을 사용하면, 파이썬 응용 프로그램을 국제화하기 위해 GNU **gettext** 패키지가 필요하지 않을 수 있습니다.)

xgettext, **pygettext** 및 유사한 도구는 메시지 카탈로그인 `.po` 파일을 생성합니다. 이 파일은 소스 코드에 표시된 모든 문자열과 이러한 문자열의 번역된 버전에 대한 자리를 포함하는 사람이 읽을 수 있는 파일입니다.

이 `.po` 파일의 사본은 지원되는 모든 자연어에 대한 번역을 작성하는 개별 인간 번역가에게 전달됩니다. 완성된 언어별 버전을 `<language-name>.po` 파일로 다시 보내고, 이는 **msgfmt** 프로그램을 사용하여 기계가 읽을 수 있는 `.mo` 바이너리 카탈로그 파일로 컴파일됩니다. `.mo` 파일은 실행 시간에 실제 번역 처리를 위해 *gettext* 모듈에서 사용됩니다.

코드에서 *gettext* 모듈을 사용하는 방법은 단일 모듈을 국제화하는지 또는 전체 응용 프로그램을 국제화하는지에 따라 다릅니다. 다음 두 섹션에서는 각 사례에 관해 설명합니다.

모듈 현지화

모듈을 현지화한다면, 전역적인 변경을 가하지 않도록 주의해야 합니다, 예를 들어, 내장 이름 공간. GNU **gettext** API 대신 클래스 기반 API를 사용해야 합니다.

모듈이 "spam" 이고 모듈의 다양한 자연어 번역 `.mo` 파일이 `/usr/share/locale` 에 GNU **gettext** 형식으로 존재한다고 가정해 봅시다. 다음은 모듈 맨 위에 들어갈 내용입니다:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

응용 프로그램 현지화

응용 프로그램을 현지화한다면, `_()` 함수를 전역적으로 내장 이름 공간에 설치할 수 있습니다, 일반적으로 응용 프로그램의 메인 드라이버 파일에서. 이렇게 하면 모든 응용 프로그램별 파일이 각 파일에 명시적으로 설치하지 않고도 `_('...')` 를 사용할 수 있습니다.

간단한 경우에는, 응용 프로그램의 메인 드라이버 파일에 다음 코드만 추가하면 됩니다:

```
import gettext
gettext.install('myapplication')
```

로케일 디렉터리를 설정해야 하면, *install()* 함수로 전달할 수 있습니다:

```
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

실행 중 언어 변경

프로그램에서 동시에 여러 언어를 지원해야 하면, 다음과 같은 식으로 여러 번역 인스턴스를 만든 다음 명시적으로 전환할 수 있습니다:

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# 언어 1을 사용하는 것으로 시작합니다
lang1.install()

# ... 시간이 지난 후에, 사용자가 언어 2를 선택합니다
lang2.install()

# ... 시간이 더 흐른 후에, 사용자가 언어 3을 선택합니다
lang3.install()
```

지연된 번역

대부분의 코딩 상황에서, 문자열은 코딩된 위치에서 번역됩니다. 그러나 때때로, 번역을 위해 문자열을 표시하지만, 실제 번역을 뒤로 연기할 필요가 있습니다. 전형적인 예는 다음과 같습니다:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]
# ...
for a in animals:
    print(a)
```

여기서, `animals` 리스트의 문자열을 번역 가능한 것으로 표시하려고 하지만, 실제로 인쇄될 때까지 번역하고 싶지는 않습니다.

이 상황을 처리 할 수 있는 한 가지 방법은 다음과 같습니다:

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))
```

이것이 작동하는 이유는 `_()` 의 더미 정의가 단순히 문자열을 변경하지 않고 반환하기 때문입니다. 그리고 이 더미 정의는 내장 이름 공간에서 `_()` 의 정의를 일시적으로 재정의합니다 (`del` 명령까지). 지역 이름 공간에 `_()` 의 이전 정의가 있다면 주의하십시오.

`_()` 의 두 번째 사용은 매개 변수가 문자열 리터럴이 아니기 때문에 **gettext** 프로그램이 "a" 를 번역 가능하다고 식별하지 않음에 유의하십시오.

이를 처리하는 다른 방법은 다음 예제를 사용하는 것입니다:

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(_(a))
```

이 경우, 번역 가능한 문자열을 `N_()` 함수로 표시하는데, `_()` 의 정의와 충돌하지 않습니다. 그러나, `N_()` 로 표시된 번역 가능한 문자열을 찾도록 메시지 추출 프로그램을 가르쳐야 할 필요가 있습니다. **xgettext**, **pygettext**, `pybabel extract` 및 **xpot** 은 모두 `-k` 명령 줄 스위치를 사용하여 이를 지원합니다. 여기서 `N_()` 의 선택은 완전히 임의적입니다; `MarkThisStringForTranslation()` 처럼 무엇이든 될 수 있습니다.

24.1.4 감사의 말

다음 분들은 이 모듈을 만드는 데 코드, 피드백, 디자인 제안, 이전 구현 및 귀중한 경험을 제공했습니다:

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

24.2 locale — 국제화 서비스

소스 코드: Lib/locale.py

---

locale 모듈은 POSIX 로케일 데이터베이스와 기능에 대한 액세스를 엽니다. POSIX 로케일 메커니즘은 프로그래머가 소프트웨어가 실행되는 국가별로 모든 세부 사항을 알 필요 없이 응용 프로그램에서 특정 문화적 문제를 다룰 수 있도록 합니다.

locale 모듈은 _locale 모듈 위에 구현되며, 이는 다시 사용 가능하다면 ANSI C 로케일 구현을 사용합니다.

locale 모듈은 다음 예외와 함수를 정의합니다:

exception locale.**Error**

*setlocale()*에 전달된 로케일이 인식되지 않을 때 발생하는 예외.

locale.**setlocale**(*category*, *locale=None*)

If *locale* is given and not None, *setlocale()* modifies the locale setting for the *category*. The available categories are listed in the data description below. *locale* may be a *string*, or a pair, language code and encoding. An empty string specifies the user' s default settings. If the modification of the locale fails, the exception *Error* is raised. If successful, the new locale setting is returned.

If *locale* is a pair, it is converted to a locale name using the locale aliasing engine. The language code has the same format as a *locale name*, but without encoding and @-modifier. The language code and encoding can be None.

*locale*이 생략되거나 None이면, *category*의 현재 설정이 반환됩니다.

*setlocale()*은 대부분의 시스템에서 스레드 안전하지 않습니다. 응용 프로그램은 보통 다음과 같은 호출로 시작합니다

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

이는 모든 범주의 로케일을 사용자의 기본 설정(보통 LANG 환경 변수에서 지정됩니다)으로 설정합니다. 그 후에 로케일을 변경하지 않으면, 다중 스레딩을 사용해도 문제가 발생하지 않습니다.

locale.**localeconv**()

현지 규칙의 데이터베이스를 딕셔너리로 반환합니다. 이 딕셔너리에는 키로 다음 문자열이 있습니다:

범주	키	의미
LC_NUMERIC	'decimal_point'	십진수 소수점 문자.
	'grouping'	'thousands_sep'가 예상되는 상대 위치를 지정하는 숫자의 시퀀스. 시퀀스가 *CHAR_MAX*로 종료되면, 더 이상의 그룹화가 수행되지 않습니다. 시퀀스가 0으로 종료되면, 마지막 그룹 크기가 반복적으로 사용됩니다.
	'thousands_sep'	그룹 간에 사용되는 문자.
LC_MONETARY	'int_curr_symbol'	국제 통화 기호.
	'currency_symbol'	현지 통화 기호.
	'p_cs_precedes/n_cs_precedes'	통화 기호가 값 앞에 오는지 여부 (각각 양과 음의 값).
	'p_sep_by_space/n_sep_by_space'	통화 기호가 스페이스로 값과 구분되는지 여부 (각각 양과 음의 값).
	'mon_decimal_point'	화폐값에 사용되는 십진수 소수점.
	'frac_digits'	화폐값의 현지 형식에 사용되는 소수점 이하 자릿수.
	'int_frac_digits'	화폐값의 국제 형식에 사용되는 소수점 이하 자릿수.
	'mon_thousands_sep'	화폐값에 사용되는 그룹 구분자.
	'mon_grouping'	'grouping'과 동등합니다, 화폐값에 사용됩니다.
	'positive_sign'	양의 화폐값을 표현하는 데 사용되는 기호.
	'negative_sign'	음의 화폐값을 표현하는 데 사용되는 기호.
	'p_sign_posn/n_sign_posn'	부호의 위치 (각각 양과 음의 값), 아래를 참조하십시오.

모든 숫자 값은 이 로케일에서 아무런 값도 지정되지 않았음을 나타내는 *CHAR_MAX*로 설정할 수 있습니다.

'p_sign_posn'과 'n_sign_posn'에 가능한 값은 다음과 같습니다.

값	설명
0	통화와 값을 괄호로 묶습니다.
1	부호는 값과 통화 기호 앞에 와야 합니다.
2	부호는 값과 통화 기호 뒤에 와야 합니다.
3	부호는 값 바로 앞에 와야 합니다.
4	부호는 값 바로 뒤에 와야 합니다.
CHAR_MAX	이 로케일에 지정된 것이 없습니다.

로케일이 다르고 숫자나 통화 문자열이 ASCII가 아니면, 함수는 일시적으로 LC_CTYPE 로케일을 LC_NUMERIC 로케일이나 LC_MONETARY 로케일로 설정합니다. 이 임시 변경은 다른 스레드에 영향을 줍니다.

버전 3.7에서 변경: 이 함수는 이제 어떤 경우에 LC_CTYPE 로케일을 LC_NUMERIC 로케일로 임시 설정합니다.

locale.**nl_langinfo**(*option*)

로케일 특정 정보를 문자열로 반환합니다. 이 함수를 모든 시스템에서 사용할 수 있는 것은 아니며, 가능한 옵션 집합은 플랫폼마다 다를 수 있습니다. 가능한 인자 값은 숫자이며, locale 모듈에 있는 기호 상수를 사용할 수 있습니다.

nl_langinfo() 함수는 다음 키 중 하나를 받아들입니다. 대부분의 설명은 GNU C 라이브러리의 해당 설명에서 가져옵니다.

locale.**CODESET**

선택한 로케일에 사용된 문자 인코딩의 이름으로 문자열을 가져옵니다.

locale.**D_T_FMT**

로케일 특정 방식으로 날짜와 시간을 표시하기 위해 *time.strftime()*의 포맷 문자열로 사용할 수 있는 문자열을 가져옵니다.

locale.**D_FMT**

로케일 특정 방식으로 날짜를 표시하기 위해 *time.strftime()*의 포맷 문자열로 사용할 수 있는 문자열을 가져옵니다.

locale.**T_FMT**

로케일 특정 방식으로 시간을 표시하기 위해 *time.strftime()*의 포맷 문자열로 사용할 수 있는 문자열을 가져옵니다.

locale.**T_FMT_AMPM**

am/pm 형식으로 시간을 나타내는 *time.strftime()*의 포맷 문자열을 가져옵니다.

locale.**DAY_1**
locale.**DAY_2**
locale.**DAY_3**
locale.**DAY_4**
locale.**DAY_5**
locale.**DAY_6**
locale.**DAY_7**

주의 n 번째 날의 이름을 가져옵니다.

> **참고**
>
> 이것은 월요일이 주의 첫 번째 요일인 국제관례(ISO 8601)가 아니라, *DAY_1*이 일요일인 미국 관례를 따릅니다.

locale.**ABDAY_1**
locale.**ABDAY_2**
locale.**ABDAY_3**
locale.**ABDAY_4**
locale.**ABDAY_5**
locale.**ABDAY_6**
locale.**ABDAY_7**

주의 n 번째 날의 줄인 이름을 가져옵니다.

locale.**MON_1**
locale.**MON_2**
locale.**MON_3**
locale.**MON_4**
locale.**MON_5**
locale.**MON_6**
locale.**MON_7**
locale.**MON_8**
locale.**MON_9**
locale.**MON_10**
locale.**MON_11**

locale.**MON_12**

n 번째 달의 이름을 가져옵니다.

locale.**ABMON_1**

locale.**ABMON_2**

locale.**ABMON_3**

locale.**ABMON_4**

locale.**ABMON_5**

locale.**ABMON_6**

locale.**ABMON_7**

locale.**ABMON_8**

locale.**ABMON_9**

locale.**ABMON_10**

locale.**ABMON_11**

locale.**ABMON_12**

n 번째 달의 줄인 이름을 가져옵니다.

locale.**RADIXCHAR**

기수 문자(십진수 점, 십진수 쉼표, 등)를 가져옵니다.

locale.**THOUSEP**

천 단위 (3자리 그룹) 구분자 문자를 가져옵니다.

locale.**YESEXPR**

예/아니요 질문에 대한 긍정적인 응답을 인식하기 위해 regex 함수에 사용할 수 있는 정규식을 가져옵니다.

locale.**NOEXPR**

예/아니요 질문에 대한 부정적인 응답을 인식하기 위해 `regex(3)` 함수에 사용할 수 있는 정규식을 가져옵니다.

> **참고**
>
> *YESEXPR*과 *NOEXPR*를 위한 정규식은 C 라이브러리의 `regex` 함수에 적합한 문법을 사용합니다, 이는 *re*에 사용되는 문법과 다를 수 있습니다.

locale.**CRNCYSTR**

통화 기호를 가져옵니다. 기호가 값 앞에 나타나야 하면 "-", 기호가 값 뒤에 나타나야 하면 "+", 또는 기호가 기수 문자를 대체해야 하면 "." 가 앞에 나옵니다.

locale.**ERA**

로케일의 각 시대에 대해 연도를 계산하고 표시하는 방법을 설명하는 문자열을 가져옵니다.

대부분의 로케일은 이 값을 정의하지 않습니다. 이 값을 정의하는 로케일의 예는 일본입니다. 일본에서, 전통적인 날짜 표시에는 당시 군주의 통치에 해당하는 시대의 이름이 포함됩니다.

일반적으로 이 값을 직접 사용할 필요는 없습니다. 포맷 문자열에 `E` 수정자를 지정하면 *time.strftime()* 함수가 이 정보를 사용합니다. 반환된 문자열의 형식은 *The Open Group Base Specifications Issue 8*, 문단 7.3.5.2 LC_TIME C-Language Access에서 지정합니다.

locale.**ERA_D_T_FMT**

로케일 특정 시대 기반 방식으로 날짜와 시간을 나타내는 *time.strftime()*의 포맷 문자열을 가져옵니다.

locale.**ERA_D_FMT**

로케일 특정 시대 기반 방식으로 날짜를 나타내는 *time.strftime()*의 포맷 문자열을 가져옵니다.

locale.**ERA_T_FMT**

로케일 특정 시대 기반 방식으로 시간을 나타내는 *time.strftime()*의 포맷 문자열을 가져옵니다.

locale.**ALT_DIGITS**

로케일별 방식으로 0에서 99까지의 값을 나타내는 데 사용되는 세미콜론으로 구분되된 최대 100개의 기호로 구성된 문자열을 가져옵니다. 대부분의 로케일에서 빈 문자열입니다.

The function temporarily sets the `LC_CTYPE` locale to the locale of the category that determines the requested value (`LC_TIME`, `LC_NUMERIC`, `LC_MONETARY` or `LC_MESSAGES`) if locales are different and the resulting string is non-ASCII. This temporary change affects other threads.

버전 3.14에서 변경: The function now temporarily sets the `LC_CTYPE` locale in some cases.

locale.**getdefaultlocale**([*envvars*])

기본 로케일 설정을 판별하려고 시도하고 `(language code, encoding)` 형식의 튜플로 반환합니다.

POSIX에 따르면, `setlocale(LC_ALL, '')`을 호출하지 않은 프로그램은 이식성 있는 `'C'` 로케일을 사용하여 실행됩니다. `setlocale(LC_ALL, '')`을 호출하면 `LANG` 변수로 정의된 기본 로케일을 사용하도록 합니다. 현재 로케일 설정을 방해하고 싶지 않기 때문에 위에서 설명한 방식으로 동작을 에뮬레이트 합니다.

다른 플랫폼과의 호환성을 유지하기 위해, `LANG` 변수뿐만 아니라 envvars 매개 변수로 제공된 변수 리스트도 검사합니다. 가장 먼저 정의된 것으로 발견된 것이 사용됩니다. *envvars*는 GNU gettext에서 사용되는 검색 경로를 기본값으로 합니다; 항상 변수 이름 `'LANG'`을 포함해야 합니다. GNU gettext 검색 경로에는 `'LC_ALL'`, `'LC_CTYPE'`, `'LANG'` 및 `'LANGUAGE'`가 순서대로 포함됩니다.

The language code has the same format as a *locale name*, but without encoding and `@`-modifier. The language code and encoding may be `None` if their values cannot be determined. The "C" locale is represented as `(None, None)`.

Deprecated since version 3.11, will be removed in version 3.15.

locale.**getlocale**(*category=LC_CTYPE*)

Returns the current setting for the given locale category as a tuple containing the language code and encoding. *category* may be one of the `LC_*` values except *LC_ALL*. It defaults to *LC_CTYPE*.

The language code has the same format as a *locale name*, but without encoding and `@`-modifier. The language code and encoding may be `None` if their values cannot be determined. The "C" locale is represented as `(None, None)`.

locale.**getpreferredencoding**(*do_setlocale=True*)

사용자 설정(user preferences)에 따라, 텍스트 데이터에 사용된 로케일 인코딩을 반환합니다. 사용자 설정은 시스템마다 다르게 표현되며, 일부 시스템에서는 프로그래밍 방식으로 제공되지 않을 수 있어서, 이 함수는 추측만 반환합니다.

일부 시스템에서는, 사용자 설정을 얻기 위해 *setlocale()*을 호출할 필요가 있어서, 이 함수는 스레드 안전하지 않습니다. setlocale을 호출할 필요가 없거나 원하지 않으면, *do_setlocale*을 `False`로 설정해야 합니다.

안드로이드에서 또는 파이썬 *UTF-8* 모드가 활성화되면, 항상 `'utf-8'`을 반환하고, 로케일 인코딩과 *do_setlocale* 인자는 무시됩니다.

The Python preinitialization configures the LC_CTYPE locale. See also the *filesystem encoding and error handler*.

버전 3.7에서 변경: 이 함수는 이제 안드로이드에서 또는 파이썬 *UTF-8* 모드가 활성화되면 항상 `"utf-8"`을 반환합니다.

locale.**getencoding**()

Get the current *locale encoding*:

- On Android and VxWorks, return `"utf-8"`.

- On Unix, return the encoding of the current *LC_CTYPE* locale. Return "utf-8" if nl_langinfo(CODESET) returns an empty string: for example, if the current LC_CTYPE locale is not supported.
- On Windows, return the ANSI code page.

The Python preinitialization configures the LC_CTYPE locale. See also the *filesystem encoding and error handler*.

This function is similar to *getpreferredencoding(False)* except this function ignores the *Python UTF-8 Mode*.

Added in version 3.11.

locale.**normalize**(*localename*)

주어진 로케일 이름에 대해 정규화된 로케일 코드를 반환합니다. 반환된 로케일 코드는 *setlocale()*에서 사용하도록 포맷됩니다. 정규화에 실패하면, 원래 이름이 변경되지 않은 상태로 반환됩니다.

주어진 인코딩이 알려지지 않았으면, 함수는 *setlocale()*처럼 로케일 코드의 기본 인코딩을 기본값으로 사용합니다.

locale.**strcoll**(*string1*, *string2*)

현재 *LC_COLLATE* 설정에 따라 두 문자열을 비교합니다. 다른 비교 함수처럼, *string1*이 *string2* 앞이니 뒤에 오는지 또는 같은지에 따라 음수나 양수 값 또는 0을 반환합니다.

locale.**strxfrm**(*string*)

문자열을 로케일을 고려한 비교에 사용할 수 있는 문자열로 변환합니다. 예를 들어, strxfrm(s1) < strxfrm(s2)는 strcoll(s1, s2) < 0과 동등합니다. 이 함수는 같은 문자열이 반복적으로 비교될 때, 예를 들어 문자열 시퀀스를 정렬할 때, 사용할 수 있습니다.

locale.**format_string**(*format*, *val*, *grouping=False*, *monetary=False*)

현재 *LC_NUMERIC* 설정에 따라 숫자 *val*을 포맷합니다. format은 % 연산자의 규칙을 따릅니다. 부동 소수점 값의 경우, 적절하다면 소수점이 수정됩니다. *grouping*이 True이면, 그룹화도 고려합니다.

*monetary*가 참이면, 변환은 화폐 천 단위 구분 기호와 그룹화 문자열을 사용합니다.

format % val에서처럼 포맷 지정자를 처리하지만, 현재 로케일 설정을 고려합니다.

버전 3.7에서 변경: *monetary* 키워드 매개 변수가 추가되었습니다.

locale.**currency**(*val*, *symbol=True*, *grouping=False*, *international=False*)

현재 *LC_MONETARY* 설정에 따라 숫자 *val*을 포맷합니다.

*symbol*이 참이면 반환된 문자열에 통화 기호가 포함됩니다, 이것이 기본값입니다. *grouping*이 True이면 (기본값이 아닙니다), 값으로 그룹화가 수행됩니다. *international*이 True이면 (기본값이 아닙니다), 국제 통화 기호가 사용됩니다.

> **참고**
>
> 이 함수는 'C' 로케일에서 작동하지 않아서, 먼저 *setlocale()*을 통해 로케일을 설정해야 합니다.

locale.**str**(*float*)

내장 함수 str(float)와 같은 형식을 사용하여 부동 소수점 숫자를 포맷하지만, 십진수 소수점을 고려합니다.

locale.**delocalize**(*string*)

LC_NUMERIC 설정에 따라, 문자열을 정규화된 숫자 문자열로 변환합니다.

Added in version 3.5.

locale.**localize**(*string, grouping=False, monetary=False*)

LC_NUMERIC 설정에 따라 포맷되는 문자열로, 문자열을 정규화된 숫자 문자열로 변환합니다.

Added in version 3.10.

locale.**atof**(*string, func=float*)

*string*에 *delocalize()*를 호출한 결과에 *func*를 호출하여, *LC_NUMERIC* 설정에 따라, 문자열을 숫자로 변환합니다.

locale.**atoi**(*string*)

LC_NUMERIC 규칙에 따라, 문자열을 정수로 변환합니다.

locale.**LC_CTYPE**

Locale category for the character type functions. Most importantly, this category defines the text encoding, i.e. how bytes are interpreted as Unicode codepoints. See **PEP 538** and **PEP 540** for how this variable might be automatically coerced to `C.UTF-8` to avoid issues created by invalid settings in containers or incompatible settings passed over remote SSH connections.

Python doesn' t internally use locale-dependent character transformation functions from `ctype.h`. Instead, an internal `pyctype.h` provides locale-independent equivalents like `Py_TOLOWER`.

locale.**LC_COLLATE**

문자열 정렬을 위한 로케일 범주. *locale* 모듈의 함수 *strcoll()*과 *strxfrm()*이 영향을 받습니다.

locale.**LC_TIME**

시간 포매팅을 위한 로케일 범주. *time.strftime()* 함수는 이러한 규칙을 따릅니다.

locale.**LC_MONETARY**

화폐값의 포매팅을 위한 로케일 범주. 사용 가능한 옵션은 *localeconv()* 함수에서 제공됩니다.

locale.**LC_MESSAGES**

메시지 표시를 위한 로케일 범주. 파이썬은 현재 응용 프로그램에 특정한 로케일을 고려한 메시지를 지원하지 않습니다. *os.strerror()*에서 반환된 메시지처럼 운영 체제에서 표시되는 메시지는 이 범주의 영향을 받을 수 있습니다.

This value may not be available on operating systems not conforming to the POSIX standard, most notably Windows.

locale.**LC_NUMERIC**

숫자 포매팅을 위한 로케일 범주. *locale* 모듈의 함수 *format_string()*, *atoi()*, *atof()* 및 *str()*은 이 범주의 영향을 받습니다. 다른 모든 숫자 포매팅 연산은 영향을 받지 않습니다.

locale.**LC_ALL**

모든 로케일 설정의 조합. 로케일을 변경할 때 이 플래그를 사용하면, 모든 범주에 대한 로케일 설정을 시도합니다. 어떤 범주에서 실패하면, 아무런 범주도 변경되지 않습니다. 이 플래그를 사용하여 로케일을 가져오면, 모든 범주의 설정을 나타내는 문자열이 반환됩니다. 이 문자열은 나중에 설정을 복원하는 데 사용할 수 있습니다.

locale.**CHAR_MAX**

이것은 *localeconv()*가 반환한 다른 값에 사용되는 기호 상수입니다.

예:

```
>>> import locale
>>> loc = locale.getlocale()  # 현재 로케일을 가져옵니다
# 독일 로케일을 사용합니다; 이름은 플랫폼에 따라 달라질 수 있습니다
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xe4n', 'foo')  # 움라우트를 포함하는 문자열을 비교합니다
>>> locale.setlocale(locale.LC_ALL, '')   # 사용자 설정 로케일을 사용합니다
>>> locale.setlocale(locale.LC_ALL, 'C')  # 기본 (C) 로케일을 사용합니다
>>> locale.setlocale(locale.LC_ALL, loc)  # 보관된 로케일을 복원합니다
```

24.2.1 배경, 세부 사항, 힌트, 팁 및 경고

C 표준은 로케일을 변경하는 데 상대적으로 비용이 많이 드는 프로그램 전체 속성으로 정의합니다. 이 위에, 로케일을 자주 변경하면 코어 덤프가 발생할 수 있는 방식으로 일부 구현이 망가져 있습니다. 이것은 로케일을 올바르게 사용하는 것을 다소 고통스럽게 만듭니다.

처음에, 프로그램이 시작할 때, 사용자가 선호하는 로케일이 무엇이든 로케일은 `C` 로케일입니다. 한 가지 예외가 있습니다: `LC_CTYPE` 범주는 시작 시 현재 로케일 인코딩을 사용자가 선호하는 로케일 인코딩으로 설정하도록 변경됩니다. 다른 범주에 대해서는 프로그램이 `setlocale(LC_ALL, '')` 을 호출하여 사용자가 선호하는 로케일 설정을 원한다고 명시적으로 말해야 합니다.

일반적으로 어떤 라이브러리 루틴에서 `setlocale()` 을 호출하는 것은 나쁜 생각입니다. 부작용으로 전체 프로그램에 영향을 미치기 때문입니다. 저장하고 복원하는 것도 거의 비슷하게 나쁩니다: 비용이 많이 들고 설정이 복원되기 전에 실행되는 다른 스레드에 영향을 줍니다.

일반적인 용도로 모듈을 코딩할 때, 로케일에 영향을 받는 로케일 독립적인 버전의 연산(가령 `time.strftime()` 에 사용되는 특정 포맷)이 필요하면, 표준 라이브러리 루틴을 사용하지 않고 수행할 수 있는 방법을 찾아야 합니다. 로케일 설정을 사용하는 것이 좋다고 자신을 설득하는 것이 더 좋습니다. 최후의 수단으로만 모듈이 `C` 가 아닌 로케일 설정과 호환되지 않는다는 것을 문서화해야 합니다.

로케일에 따라 숫자 연산을 수행하는 유일한 방법은 이 모듈이 정의한 특수 함수인 `atof()`, `atoi()`, `format_string()`, `str()` 을 사용하는 것입니다.

로케일에 따라 대소 문자 변환과 문자 분류를 수행할 방법이 없습니다. (유니코드) 텍스트 문자열의 경우 이것은 문자 값으로만 수행되는 반면, 바이트 문자열의 경우, 바이트의 ASCII 값에 따라 수행되고, 높은 비트가 설정된 바이트(즉, ASCII가 아닌 바이트)는 절대 변환되거나 글자나 공백과 같은 문자 클래스의 일부로 고려되지 않습니다.

24.2.2 Locale names

The format of the locale name is platform dependent, and the set of supported locales can depend on the system configuration.

On Posix platforms, it usually has the format[1]:

```
language ["_" territory] ["." charset] ["@" modifier]
```

where *language* is a two- or three-letter language code from ISO 639, *territory* is a two-letter country or region code from ISO 3166, *charset* is a locale encoding, and *modifier* is a script name, a language subtag, a sort order identifier, or other locale modifier (for example, "latin", "valencia", "stroke" and "euro").

On Windows, several formats are supported.[2][3] A subset of IETF BCP 47 tags:

```
language ["-" script] ["-" territory] ["." charset]
language ["-" script] "-" territory "-" modifier
```

where *language* and *territory* have the same meaning as in Posix, *script* is a four-letter script code from ISO 15924, and *modifier* is a language subtag, a sort order identifier or custom modifier (for example, "valencia", "stroke" or "x-python"). Both hyphen (`'-'`) and underscore (`'_'`) separators are supported. Only UTF-8 encoding is allowed for BCP 47 tags.

Windows also supports locale names in the format:

```
language ["_" territory] ["." charset]
```

where *language* and *territory* are full names, such as "English" and "United States", and *charset* is either a code page number (for example, "1252") or UTF-8. Only the underscore separator is supported in this format.

The "C" locale is supported on all platforms.

[1] IEEE Std 1003.1-2024; 8.2 Internationalization Variables
[2] UCRT Locale names, Languages, and Country/Region strings
[3] Locale Names

24.2.3 확장 작성자와 파이썬을 내장하는 프로그램의 경우

확장 모듈은 현재 로케일이 무엇인지 찾는 것을 제외하고는 *setlocale()*을 호출해서는 안 됩니다. 그러나 반환 값은 이식성 있게 복원하는 데만 사용될 수 있기 때문에, 그다지 유용하지 않습니다 (아마도 로케일이 C인지 아닌지를 확인하는 것은 예외입니다).

파이썬 코드가 *locale* 모듈을 사용하여 로케일을 변경할 때, 내장하는 응용 프로그램에도 영향을 줍니다. 내장하는 응용 프로그램이 이를 원하지 않으면, `config.c` 파일의 내장 모듈 테이블에서 `_locale` 확장 모듈 (이것이 모든 작업을 수행합니다)을 제거하고, 공유 라이브러리로 `_locale` 모듈에 액세스할 수 없도록 확인하십시오.

24.2.4 메시지 카탈로그에 액세스

`locale.`**`gettext`**(*msg*)

`locale.`**`dgettext`**(*domain*, *msg*)

`locale.`**`dcgettext`**(*domain*, *msg*, *category*)

`locale.`**`textdomain`**(*domain*)

`locale.`**`bindtextdomain`**(*domain*, *dir*)

`locale.`**`bind_textdomain_codeset`**(*domain*, *codeset*)

locale 모듈은 이 인터페이스를 제공하는 시스템에서 C 라이브러리의 gettext 인터페이스를 제공합니다. *gettext()*, *dgettext()*, *dcgettext()*, *textdomain()*, *bindtextdomain()* 및 *bind_textdomain_codeset()* 함수로 구성됩니다. 이는 *gettext* 모듈의 같은 함수와 유사하지만, 메시지 카탈로그에 C 라이브러리의 바이너리 형식을 사용하고, 메시지 카탈로그를 찾는 데 C 라이브러리의 검색 알고리즘을 사용합니다.

파이썬 응용 프로그램은 일반적으로 이러한 함수를 호출할 필요가 없으며, 대신 *gettext*를 사용해야 합니다. 이 규칙의 알려진 예외는 내부적으로 C 함수 `gettext`나 `dcgettext`를 호출하는 추가 C 라이브러리와 링크되는 응용 프로그램입니다. 이러한 응용 프로그램에서는, 라이브러리가 메시지 카탈로그를 올바르게 찾을 수 있도록 텍스트 도메인을 바인드해야 할 수도 있습니다.

CHAPTER 25

Graphical user interfaces with Tk

Tk/Tcl은 오랫동안 파이썬의 중요한 부분이었습니다. 견고하고 플랫폼 독립적인 윈도우 도구상자(파이썬 프로그래머는 `tkinter` 패키지를 통해 사용할 수 있습니다)와 그 확장(`tkinter.ttk` 모듈)을 제공합니다.

`tkinter` 패키지는 Tcl/Tk 위에 올라가는 얇은 객체 지향 계층입니다. `tkinter`를 사용하기 위해 Tcl 코드를 작성할 필요는 없지만, Tk 문서와 때때로 Tcl 문서를 참고해야 합니다. `tkinter`는 Tk 위젯을 파이썬 클래스로 구현하는 래퍼 집합입니다.

`tkinter`의 가장 큰 장점은 빠르고, 일반적으로 파이썬과 함께 제공된다는 것입니다. 표준 설명서가 약하긴 하지만 레퍼런스, 자습서, 서적 및 기타 자료와 같은 훌륭한 자료를 구할 수 있습니다. `tkinter`는 낡은 모양과 느낌으로도 유명합니다만, Tk 8.5에서 크게 개선되었습니다. 그렇지만, 여러분이 관심을 기울일 만한 다른 GUI 라이브러리가 많이 있습니다. 파이썬 위키는 몇 가지 대체 GUI 프레임워크 및 도구를 나열합니다.

25.1 `tkinter` — Tcl/Tk 파이썬 인터페이스

소스 코드: Lib/tkinter/__init__.py

`tkinter` 패키지("Tk 인터페이스")는 Tcl/Tk GUI 툴킷에 대한 표준 파이썬 인터페이스입니다. Tk와 `tkinter`는 macOS를 포함하는 대부분의 유닉스 플랫폼과 윈도우 시스템에서 사용할 수 있습니다.

명령 줄에서 `python -m tkinter`를 실행하면 간단한 Tk 인터페이스를 보여주는 창을 열어, `tkinter`가 시스템에 제대로 설치되었는지 알 수 있도록 하고, 설치된 Tcl/Tk 버전을 보여주기 때문에, 그 버전에 해당하는 Tcl/Tk 설명서를 읽을 수 있습니다.

Tkinter supports a range of Tcl/Tk versions, built either with or without thread support. The official Python binary release bundles Tcl/Tk 8.6 threaded. See the source code for the `_tkinter` module for more information about supported versions.

Tkinter is not a thin wrapper, but adds a fair amount of its own logic to make the experience more pythonic. This documentation will concentrate on these additions and changes, and refer to the official Tcl/Tk documentation for details that are unchanged.

> **참고**
>
> Tcl/Tk 8.5 (2007) introduced a modern set of themed user interface components along with a new API to use them. Both old and new APIs are still available. Most documentation you will find online still uses the old API and can be woefully outdated.

더 보기

- **TkDocs**
  Extensive tutorial on creating user interfaces with Tkinter. Explains key concepts, and illustrates recommended approaches using the modern API.
- **Tkinter 8.5 reference: a GUI for Python**
  Reference documentation for Tkinter 8.5 detailing available classes, methods, and options.

Tcl/Tk Resources:

- **Tk commands**
  Comprehensive reference to each of the underlying Tcl/Tk commands used by Tkinter.
- **Tcl/Tk 홈 페이지**
  Additional documentation, and links to Tcl/Tk core development.

Books:

- **Modern Tkinter for Busy Python Developers**
  Mark Roseman 지음. (ISBN 978-1999149567)
- **Python GUI programming with Tkinter**
  Alan D. Moore 지음. (ISBN 978-1788835886)
- **Programming Python**
  Mark Lutz 지음; 탁월하게 Tkinter를 다루고 있습니다. (ISBN 978-0596158101)
- **Tcl and the Tk Toolkit (2nd edition)**
  By John Ousterhout, inventor of Tcl/Tk, and Ken Jones; does not cover Tkinter. (ISBN 978-0321336330)

25.1.1 Architecture

Tcl/Tk is not a single library but rather consists of a few distinct modules, each with separate functionality and its own official documentation. Python' s binary releases also ship an add-on module together with it.

Tcl
: Tcl is a dynamic interpreted programming language, just like Python. Though it can be used on its own as a general-purpose programming language, it is most commonly embedded into C applications as a scripting engine or an interface to the Tk toolkit. The Tcl library has a C interface to create and manage one or more instances of a Tcl interpreter, run Tcl commands and scripts in those instances, and add custom commands implemented in either Tcl or C. Each interpreter has an event queue, and there are facilities to send events to it and process them. Unlike Python, Tcl' s execution model is designed around cooperative multitasking, and Tkinter bridges this difference (see *Threading model* for details).

Tk
: Tk is a Tcl package implemented in C that adds custom commands to create and manipulate GUI widgets. Each `Tk` object embeds its own Tcl interpreter instance with Tk loaded into it. Tk' s widgets are very customizable, though at the cost of a dated appearance. Tk uses Tcl' s event queue to generate and process GUI events.

Ttk
: Themed Tk (Ttk) is a newer family of Tk widgets that provide a much better appearance on different platforms than many of the classic Tk widgets. Ttk is distributed as part of Tk, starting with Tk version 8.5. Python bindings are provided in a separate module, `tkinter.ttk`.

Internally, Tk and Ttk use facilities of the underlying operating system, i.e., Xlib on Unix/X11, Cocoa on macOS, GDI on Windows.

When your Python application uses a class in Tkinter, e.g., to create a widget, the `tkinter` module first assembles a Tcl/Tk command string. It passes that Tcl command string to an internal `_tkinter` binary module, which then calls the Tcl interpreter to evaluate it. The Tcl interpreter will then call into the Tk and/or Ttk packages, which will in turn make calls to Xlib, Cocoa, or GDI.

25.1.2 Tkinter 모듈

Support for Tkinter is spread across several modules. Most applications will need the main *tkinter* module, as well as the *tkinter.ttk* module, which provides the modern themed widget set and API:

```
from tkinter import *
from tkinter import ttk
```

class tkinter.**Tk**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=True*, *sync=False*, *use=None*)

최상위 Tk 위젯을 만들고 이 위젯에 대한 Tcl 인터프리터를 초기화합니다. 이 위젯은 일반적으로 응용 프로그램의 메인 창입니다. 인스턴스마다 고유한 Tcl 인터프리터가 연결됩니다.

The *Tk* class is typically instantiated using all default values. However, the following keyword arguments are currently recognized:

screenName
When given (as a string), sets the `DISPLAY` environment variable. (X11 only)

baseName
Name of the profile file. By default, *baseName* is derived from the program name (`sys.argv[0]`).

className
Name of the widget class. Used as a profile file and also as the name with which Tcl is invoked (*argv0* in *interp*).

useTk
If `True`, initialize the Tk subsystem. The *tkinter.Tcl()* function sets this to `False`.

sync
If `True`, execute all X server commands synchronously, so that errors are reported immediately. Can be used for debugging. (X11 only)

use
Specifies the *id* of the window in which to embed the application, instead of it being created as an independent toplevel window. *id* must be specified in the same way as the value for the -use option for toplevel widgets (that is, it has a form like that returned by `winfo_id()`).

Note that on some platforms this will only work correctly if *id* refers to a Tk frame or toplevel that has its -container option enabled.

Tk reads and interprets profile files, named `.className.tcl` and `.baseName.tcl`, into the Tcl interpreter and calls *exec()* on the contents of `.className.py` and `.baseName.py`. The path for the profile files is the `HOME` environment variable or, if that isn' t defined, then *os.curdir*.

tk
The Tk application object created by instantiating *Tk*. This provides access to the Tcl interpreter. Each widget that is attached the same instance of *Tk* has the same value for its *tk* attribute.

master
The widget object that contains this widget. For *Tk*, the *master* is *None* because it is the main window. The terms *master* and *parent* are similar and sometimes used interchangeably as argument names; however, calling `winfo_parent()` returns a string of the widget name whereas *master* returns the object. *parent/child* reflects the tree-like relationship while *master/slave* reflects the container structure.

children
The immediate descendants of this widget as a *dict* with the child widget names as the keys and the child instance objects as the values.

tkinter.**Tcl**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=False*)

Tcl() 함수는 Tk 서브 시스템을 초기화하지 않는다는 것을 제외하고는, *Tk* 클래스에 의해 만들어지는 것과 비슷한 객체를 만드는 팩토리 함수입니다. 불필요한 최상위 창을 만들고 싶지 않거나 만들 수 없는 (가령 X 서버가 없는 유닉스/리눅스 시스템) 환경에서 Tcl 인터프리터를 구동할 때 가장 유용합니다. *Tcl()* 객체에 의해 만들어진 객체는 `loadtk()` 메서드를 호출하여 만들어지는 최상위 창(과 초기화된 Tk 서브 시스템)을 가질 수 있습니다.

Tk 지원을 제공하는 모듈은 다음과 같습니다:

tkinter
주 Tkinter 모듈.

tkinter.colorchooser
사용자가 색상을 선택할 수 있게 하는 대화 상자.

tkinter.commondialog
여기에 나열된 다른 모듈에 정의된 대화 상자의 베이스 기본 클래스.

tkinter.filedialog
사용자가 열거나 저장할 파일을 지정할 수 있도록 하는 일반 대화 상자입니다.

tkinter.font
글꼴과 관련된 작업에 도움이 되는 유틸리티.

tkinter.messagebox
표준 Tk 대화 상자에 액세스합니다.

tkinter.scrolledtext
세로 스크롤 막대가 내장된 Text 위젯.

tkinter.simpledialog
기본 대화 상자와 편리 함수.

tkinter.ttk
Themed widget set introduced in Tk 8.5, providing modern alternatives for many of the classic widgets in the main `tkinter` module.

추가 모듈:

_tkinter
Tcl/Tk 에 대한 저수준 인터페이스를 포함하는 바이너리 모듈. 주 `tkinter` 모듈에 의해 자동으로 임포트되고, 응용 프로그램 프로그래머가 직접 사용해서는 안 됩니다. 일반적으로 공유 라이브러리 (또는 DLL) 이지만, 때에 따라 파이썬 인터프리터에 정적으로 링크될 수도 있습니다.

idlelib
Python' s Integrated Development and Learning Environment (IDLE). Based on `tkinter`.

tkinter.constants
Symbolic constants that can be used in place of strings when passing various parameters to Tkinter calls. Automatically imported by the main `tkinter` module.

tkinter.dnd
(실험적) `tkinter`를 위한 드래그 앤드 드롭 지원. Tk DND로 대체 될 때 폐지될 것입니다.

turtle
Tk 창에서의 터틀(turtle) 그래픽.

25.1.3 Tkinter 구명조끼

This section is not designed to be an exhaustive tutorial on either Tk or Tkinter. For that, refer to one of the external resources noted earlier. Instead, this section provides a very quick orientation to what a Tkinter application looks like, identifies foundational Tk concepts, and explains how the Tkinter wrapper is structured.

The remainder of this section will help you to identify the classes, methods, and options you' ll need in your Tkinter application, and where to find more detailed documentation on them, including in the official Tcl/Tk reference manual.

Hello World 프로그램

We' ll start by walking through a "Hello World" application in Tkinter. This isn' t the smallest one we could write, but has enough to illustrate some key concepts you' ll need to know.

```
from tkinter import *
from tkinter import ttk
root = Tk()
frm = ttk.Frame(root, padding=10)
frm.grid()
ttk.Label(frm, text="Hello World!").grid(column=0, row=0)
ttk.Button(frm, text="Quit", command=root.destroy).grid(column=1, row=0)
root.mainloop()
```

After the imports, the next line creates an instance of the `Tk` class, which initializes Tk and creates its associated Tcl interpreter. It also creates a toplevel window, known as the root window, which serves as the main window of the application.

The following line creates a frame widget, which in this case will contain a label and a button we' ll create next. The frame is fit inside the root window.

The next line creates a label widget holding a static text string. The `grid()` method is used to specify the relative layout (position) of the label within its containing frame widget, similar to how tables in HTML work.

A button widget is then created, and placed to the right of the label. When pressed, it will call the `destroy()` method of the root window.

Finally, the `mainloop()` method puts everything on the display, and responds to user input until the program terminates.

Important Tk Concepts

Even this simple program illustrates the following key Tk concepts:

위젯
: A Tkinter user interface is made up of individual *widgets*. Each widget is represented as a Python object, instantiated from classes like `ttk.Frame`, `ttk.Label`, and `ttk.Button`.

widget hierarchy
: Widgets are arranged in a *hierarchy*. The label and button were contained within a frame, which in turn was contained within the root window. When creating each *child* widget, its *parent* widget is passed as the first argument to the widget constructor.

구성 옵션
: Widgets have *configuration options*, which modify their appearance and behavior, such as the text to display in a label or button. Different classes of widgets will have different sets of options.

geometry management
: Widgets aren' t automatically added to the user interface when they are created. A *geometry manager* like `grid` controls where in the user interface they are placed.

event loop
: Tkinter reacts to user input, changes from your program, and even refreshes the display only when actively running an *event loop*. If your program isn' t running the event loop, your user interface won' t update.

Understanding How Tkinter Wraps Tcl/Tk

When your application uses Tkinter' s classes and methods, internally Tkinter is assembling strings representing Tcl/Tk commands, and executing those commands in the Tcl interpreter attached to your application' s `Tk` instance.

Whether it' s trying to navigate reference documentation, trying to find the right method or option, adapting some existing code, or debugging your Tkinter application, there are times that it will be useful to understand what those underlying Tcl/Tk commands look like.

To illustrate, here is the Tcl/Tk equivalent of the main part of the Tkinter script above.

```
ttk::frame .frm -padding 10
grid .frm
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
grid [ttk::label .frm.lbl -text "Hello World!"] -column 0 -row 0
grid [ttk::button .frm.btn -text "Quit" -command "destroy ."] -column 1 -row 0
```

Tcl' s syntax is similar to many shell languages, where the first word is the command to be executed, with arguments to that command following it, separated by spaces. Without getting into too many details, notice the following:

- The commands used to create widgets (like `ttk::frame`) correspond to widget classes in Tkinter.
- Tcl widget options (like `-text`) correspond to keyword arguments in Tkinter.
- Widgets are referred to by a *pathname* in Tcl (like `.frm.btn`), whereas Tkinter doesn' t use names but object references.
- A widget' s place in the widget hierarchy is encoded in its (hierarchical) pathname, which uses a `.` (dot) as a path separator. The pathname for the root window is just `.` (dot). In Tkinter, the hierarchy is defined not by pathname but by specifying the parent widget when creating each child widget.
- Operations which are implemented as separate *commands* in Tcl (like `grid` or `destroy`) are represented as *methods* on Tkinter widget objects. As you' ll see shortly, at other times Tcl uses what appear to be method calls on widget objects, which more closely mirror what would is used in Tkinter.

How do I…? What option does…?

If you' re not sure how to do something in Tkinter, and you can' t immediately find it in the tutorial or reference documentation you' re using, there are a few strategies that can be helpful.

First, remember that the details of how individual widgets work may vary across different versions of both Tkinter and Tcl/Tk. If you' re searching documentation, make sure it corresponds to the Python and Tcl/Tk versions installed on your system.

When searching for how to use an API, it helps to know the exact name of the class, option, or method that you' re using. Introspection, either in an interactive Python shell or with `print()`, can help you identify what you need.

To find out what configuration options are available on any widget, call its `configure()` method, which returns a dictionary containing a variety of information about each object, including its default and current values. Use `keys()` to get just the names of each option.

```
btn = ttk.Button(frm, ...)
print(btn.configure().keys())
```

As most widgets have many configuration options in common, it can be useful to find out which are specific to a particular widget class. Comparing the list of options to that of a simpler widget, like a frame, is one way to do that.

```
print(set(btn.configure().keys()) - set(frm.configure().keys()))
```

Similarly, you can find the available methods for a widget object using the standard `dir()` function. If you try it, you' ll see there are over 200 common widget methods, so again identifying those specific to a widget class is helpful.

```
print(dir(btn))
print(set(dir(btn)) - set(dir(frm)))
```

Navigating the Tcl/Tk Reference Manual

As noted, the official Tk commands reference manual (man pages) is often the most accurate description of what specific operations on widgets do. Even when you know the name of the option or method that you need, you may still have a few places to look.

While all operations in Tkinter are implemented as method calls on widget objects, you' ve seen that many Tcl/Tk operations appear as commands that take a widget pathname as its first parameter, followed by optional parameters, e.g.

```
destroy .
grid .frm.btn -column 0 -row 0
```

Others, however, look more like methods called on a widget object (in fact, when you create a widget in Tcl/Tk, it creates a Tcl command with the name of the widget pathname, with the first parameter to that command being the name of a method to call).

```
.frm.btn invoke
.frm.lbl configure -text "Goodbye"
```

In the official Tcl/Tk reference documentation, you' ll find most operations that look like method calls on the man page for a specific widget (e.g., you' ll find the `invoke()` method on the ttk::button man page), while functions that take a widget as a parameter often have their own man page (e.g., grid).

You' ll find many common options and methods in the options or ttk::widget man pages, while others are found in the man page for a specific widget class.

You' ll also find that many Tkinter methods have compound names, e.g., `winfo_x()`, `winfo_height()`, `winfo_viewable()`. You' d find documentation for all of these in the winfo man page.

> **참고**
>
> Somewhat confusingly, there are also methods on all Tkinter widgets that don' t actually operate on the widget, but operate at a global scope, independent of any widget. Examples are methods for accessing the clipboard or the system bell. (They happen to be implemented as methods in the base `Widget` class that all Tkinter widgets inherit from).

25.1.4 Threading model

Python and Tcl/Tk have very different threading models, which *tkinter* tries to bridge. If you use threads, you may need to be aware of this.

A Python interpreter may have many threads associated with it. In Tcl, multiple threads can be created, but each thread has a separate Tcl interpreter instance associated with it. Threads can also create more than one interpreter instance, though each interpreter instance can be used only by the one thread that created it.

Each `Tk` object created by *tkinter* contains a Tcl interpreter. It also keeps track of which thread created that interpreter. Calls to *tkinter* can be made from any Python thread. Internally, if a call comes from a thread other than the one that created the `Tk` object, an event is posted to the interpreter' s event queue, and when executed, the result is returned to the calling Python thread.

Tcl/Tk applications are normally event-driven, meaning that after initialization, the interpreter runs an event loop (i.e. `Tk.mainloop()`) and responds to events. Because it is single-threaded, event handlers must respond quickly, otherwise they will block other events from being processed. To avoid this, any long-running computations should not run in an event handler, but are either broken into smaller pieces using timers, or run in another thread. This is different from many GUI toolkits where the GUI runs in a completely separate thread from all application code including event handlers.

If the Tcl interpreter is not running the event loop and processing events, any *tkinter* calls made from threads other than the one running the Tcl interpreter will fail.

A number of special cases exist:

- Tcl/Tk libraries can be built so they are not thread-aware. In this case, *tkinter* calls the library from the originating Python thread, even if this is different than the thread that created the Tcl interpreter. A global lock ensures only one call occurs at a time.
- While *tkinter* allows you to create more than one instance of a `Tk` object (with its own interpreter), all interpreters that are part of the same thread share a common event queue, which gets ugly fast. In practice, don' t create more than one instance of `Tk` at a time. Otherwise, it' s best to create them in separate threads and ensure you' re running a thread-aware Tcl/Tk build.

- Blocking event handlers are not the only way to prevent the Tcl interpreter from reentering the event loop. It is even possible to run multiple nested event loops or abandon the event loop entirely. If you' re doing anything tricky when it comes to events or threads, be aware of these possibilities.
- There are a few select *tkinter* functions that presently work only when called from the thread that created the Tcl interpreter.

25.1.5 간편한 레퍼런스

옵션 설정

옵션은 위젯의 색상과 테두리 너비와 같은 것을 제어합니다. 옵션은 세 가지 방법으로 설정할 수 있습니다:

객체 생성 시, 키워드 인자 사용하기

```
fred = Button(self, fg="red", bg="blue")
```

객체 생성 후, 딕셔너리 인덱스처럼 옵션 이름을 다루기

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

config() 메서드를 사용하여 객체 생성 이후 여러 어트리뷰트를 갱신하기.

```
fred.config(fg="red", bg="blue")
```

주어진 옵션과 그 동작에 대한 완전한 설명은, 해당 위젯의 Tk 매뉴얼 페이지를 참조하십시오.

매뉴얼 페이지는 각 위젯에 대해 "표준 옵션(STANDARD OPTIONS)" 과 "위젯 특정 옵션(WIDGET SPECIFIC OPTIONS)" 을 나열합니다. 전자는 많은 위젯에 공통적인 옵션 목록이며, 후자는 그 위젯에만 적용되는 옵션입니다. 표준 옵션은 *options(3)* 매뉴얼 페이지에 설명되어 있습니다.

이 문서에서는 표준과 위젯 특정 옵션을 구분하지 않습니다. 일부 옵션은 일부 위젯에 적용되지 않습니다. 특정 위젯이 특정 옵션에 응답하는지는 위젯의 클래스에 따라 다릅니다; 버튼에는 `command` 옵션이 있는데, 레이블은 그렇지 않습니다.

주어진 위젯이 지원하는 옵션은 위젯의 매뉴얼 페이지에 나열되거나, 실행 시간에 인자 없이 `config()` 메서드를 호출하거나 해당 위젯에서 `keys()` 메서드를 호출하여 조회할 수 있습니다. 이러한 호출의 반환값은 키가 옵션의 이름인 문자열(예를 들어, `'relief'`)이고 값이 5-튜플인 딕셔너리입니다.

`bg`와 같은 일부 옵션은 긴 이름을 가진 공통 옵션의 동의어입니다 (`bg`는 "background" 의 줄임말입니다). `config()` 메서드에 줄인 옵션을 전달하면 5-튜플이 아닌 2-튜플을 반환합니다. 전달된 2-튜플에는 동의어의 이름과 "실제" 옵션이 담겨있습니다 (가령 `('bg', 'background')`).

인덱스	의미	예
0	옵션 이름	`'relief'`
1	데이터베이스 조회를 위한 옵션 이름	`'relief'`
2	데이터베이스 조회를 위한 옵션 클래스	`'Relief'`
3	기본값	`'raised'`
4	현재 값	`'groove'`

예:

```
>>> print(fred.config())
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

물론, 인쇄된 딕셔너리에는 사용 가능한 모든 옵션과 해당 값이 포함됩니다. 이것은 예시일뿐입니다.

패커

패커(packer)는 Tk의 지오메트리 관리 메커니즘 중 하나입니다. 지오메트리 관리자는 컨테이너(위젯들의 공동 *master*) 내에서의 위젯의 상대 위치를 지정하는 데 사용됩니다. 더 성가신 *placer*(잘 사용되지 않고, 여기서는 다루지 않습니다) 와는 달리, 패커는 위에, 왼쪽에, 채우기 등의 정성적(qualitative) 관계 규정을 취하고, 여러분을 위해 정확한 배치 좌표를 결정하기 위한 모든 작업을 수행합니다.

모든 *master* 위젯의 크기는 안에 있는 "슬레이브(slave) 위젯"의 크기에 의해 결정됩니다. 패커는 슬레이브 위젯이 패킹 되는 마스터 내부에 나타나는 위치를 제어하는 데 사용됩니다. 원하는 배치를 얻기 위해 프레임에 위젯을 팩하고, 프레임을 다른 프레임에 팩할 수 있습니다. 또한, 일단 팩 되면, 점진적인 구성의 변경을 수용하기 위해 배치가 동적으로 조정됩니다.

위젯은 지오메트리 관리자로 지오메트리를 지정할 때까지 표시되지 않습니다. 지오메트리 명세를 빠뜨리는 것은 초기에 흔한 실수이고, 위젯을 만들었지만, 아무것도 나타나지 않을 때 놀라게 됩니다. 위젯은 예를 들어 패커의 `pack()` 메서드가 적용된 후에만 나타납니다.

pack() 메서드는 컨테이너 내에서 위젯이 표시되는 위치와 메인 응용 프로그램 윈도우의 크기가 조정될 때 어떻게 동작할지를 제어하는 키워드 옵션/값 쌍으로 호출할 수 있습니다. 여기 몇 가지 예가 있습니다:

```
fred.pack()                    # 기본값은 side = "top"
fred.pack(side="left")
fred.pack(expand=1)
```

패커 옵션

패커와 그것이 받을 수 있는 옵션에 대한 더 자세한 정보는 매뉴얼 페이지와 John Ousterhout의 책의 183 쪽을 참조하십시오.

anchor
: 앵커(anchor) 형. 패커가 각 슬레이브를 컨테이너에 넣을 위치를 나타냅니다.

expand
: 불리언(boolean), `0` 또는 `1`.

fill
: 유효한 값: `'x'`, `'y'`, `'both'`, `'none'`.

ipadx와 ipady
: 거리(distance) - 슬레이브 위젯의 각 변의 내부 패딩을 지정합니다.

padx와 pady
: 거리(distance) - 슬레이브 위젯의 각 변의 외부 패딩을 지정합니다.

side
: 유효한 값: `'left'`, `'right'`, `'top'`, `'bottom'`.

위젯 변수 결합하기

(텍스트 입력 위젯과 같은) 일부 위젯의 현재 값 설정은 특수 옵션을 사용하여 응용 프로그램 변수에 직접 연결할 수 있습니다. 이 옵션은 `variable`, `textvariable`, `onvalue`, `offvalue` 및 `value`입니다. 이 연결은 양방향으로 작동합니다: 어떤 이유로 든 변수가 변경되면, 연결된 위젯은 새 값을 반영하도록 갱신됩니다.

불행히도, *tkinter*의 현재 구현에서는 `variable` 또는 `textvariable` 옵션을 통해 임의의 파이썬 변수를 위젯으로 넘길 수 없습니다. 작동하는 유일한 종류의 변수는 *tkinter*에 정의된 Variable이라는 클래스에서 서브 클래싱되는 변수입니다.

이미 정의된 Variable의 유용한 서브 클래스가 많이 있습니다: `StringVar`, `IntVar`, `DoubleVar` 및 `BooleanVar`. 이러한 변수의 현재 값을 읽으려면 `get()` 메서드를 호출하고, 값을 바꾸려면 `set()` 메서드를 호출합니다. 이 프로토콜을 따르면, 여러분이 더는 개입하지 않더라도 위젯은 변수의 값을 항상 추적합니다.

예를 들면:

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()

        self.entrythingy = tk.Entry()
        self.entrythingy.pack()

        # 여기에 응용 프로그램 변수를 만듭니다.
        self.contents = tk.StringVar()
        # 어떤 값으로 설정합니다
        self.contents.set("this is a variable")
        # Entry 위젯에 이 변수를 감시하도록 지시합니다.
        self.entrythingy["textvariable"] = self.contents

        # 사용자가 return 키를 눌렀을 때를 위한 콜백을 정의합니다.
        # 변수의 현재값을 인쇄합니다.
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print("Hi. The current entry content is:",
              self.contents.get())

root = tk.Tk()
myapp = App(root)
myapp.mainloop()
```

창 관리자

Tk에는, 창 관리자와 상호 작용하기 위한 유틸리티 명령인 `wm`이 있습니다. `wm` 명령 옵션을 사용하여 제목, 배치, 아이콘 비트맵 등과 같은 항목을 제어할 수 있습니다. *tkinter*에서, 이러한 명령은 `Wm` 클래스의 메서드로 구현되었습니다. 최상위 위젯은 `Wm` 클래스의 서브 클래스이므로, `Wm` 메서드를 직접 호출할 수 있습니다.

주어진 위젯을 포함하는 최상위 창을 가져오려면, 종종 위젯의 마스터를 참조하는 것만으로도 됩니다. 물론 위젯이 프레임 안에 팩 되어 있다면, 마스터는 최상위 창을 나타내지 않을 것입니다. 임의의 위젯이 포함된 최상위 창을 가져오려면, `_root()` 메서드를 호출하면 됩니다. 이 메서드는 이 함수가 구현 일부이며, Tk 기능에 대한 인터페이스가 아니라는 사실을 나타내기 위해 밑줄로 시작합니다.

다음은 일반적인 사용 예입니다:

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

# 응용 프로그램을 만듭니다
myapp = App()

#
# 이것들은 창 관리자 클래스에 대한 메서드 호출입니다
#
myapp.master.title("My Do-Nothing Application")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
myapp.master.maxsize(1000, 400)

# 프로그램을 시작합니다
myapp.mainloop()
```

Tk 옵션 데이터형

anchor
유효한 값은 나침반의 눈금입니다: `"n"`, `"ne"`, `"e"`, `"se"`, `"s"`, `"sw"`, `"w"`, `"nw"`, 그리고 `"center"`.

bitmap
8개의 내장된, 이름있는 비트맵이 있습니다: `'error'`, `'gray25'`, `'gray50'`, `'hourglass'`, `'info'`, `'questhead'`, `'question'`, `'warning'`. X 비트맵 파일명을 지정하려면 `"@/usr/contrib/bitmap/gumby.bit"`처럼 `@`를 앞에 붙인 파일의 전체 경로를 지정하십시오.

boolean
정수 0이나 1 또는 문자열 `"yes"`나 `"no"`를 전달할 수 있습니다.

callback
인자를 취하지 않는 파이썬 함수입니다. 예를 들면:

```
def print_it():
    print("hi there")
fred["command"] = print_it
```

color
색상은 rgb.txt 파일에 있는 X 색상의 이름이나, RGB 값을 4비트: `"#RGB"`, 8비트: `"#RRGGBB"`, 12비트: `"#RRRGGGBBB"`, 또는 16비트: `"#RRRRGGGGBBBB"` 범위로 표현하는 문자열로 제공됩니다. 여기서 R, G, B는 유효한 임의의 16진수를 나타냅니다. 자세한 내용은 Ousterhout의 책 160쪽을 참조하십시오.

cursor
`XC_` 접두사 없이 `cursorfont.h`의 표준 X 커서 이름을 사용할 수 있습니다. 예를 들어, 손 모양 커서(`XC_hand2`)를 얻으려면, `"hand2"` 문자열을 사용하십시오. 여러분 자신의 비트맵과 마스크 파일을 지정할 수도 있습니다. Ousterhout의 책 179쪽을 보십시오.

distance
화면 거리는 픽셀이나 절대 거리로 지정할 수 있습니다. 픽셀은 숫자로 절대 거리는 문자열로 지정되며, 끝에 붙는 문자는 단위를 나타냅니다: `c`는 센티미터, `i`는 인치, `m`은 밀리미터, `p`는 프린터의 포인트입니다. 예를 들어, 3.5 인치는 `"3.5i"`로 표현됩니다.

font
Tk는 `{courier 10 bold}`와 같은, 목록 글꼴 이름 형식을 사용합니다. 양수로 표현된 글꼴 크기는 포인트로 측정됩니다; 음수로 표현된 크기는 픽셀 단위로 측정됩니다.

geometry
이것은 `너비x높이` 형식의 문자열로, 대부분의 위젯에서 너비와 높이는 픽셀 단위로 측정됩니다 (텍스트를 표시하는 위젯에서는 문자 단위). 예를 들어: `fred["geometry"] = "200x100"`.

justify
유효한 값은 문자열입니다: `"left"`, `"center"`, `"right"` 및 `"fill"`.

region
이것은 스페이스로 구분된 네 개의 요소가 있는 문자열이며, 각 요소는 유효한 거리(위를 참조하세요)입니다. 예를 들어: `"2 3 4 5"`와 `"3i 2i 4.5i 2i"`와 `"3c 2c 4c 10.43c"`는 모두 유효한 영역(region)입니다.

relief
위젯의 테두리 스타일을 결정합니다. 유효한 값은 다음과 같습니다: `"raised"`, `"sunken"`, `"flat"`, `"groove"` 및 `"ridge"`.

scrollcommand
이것은 거의 항상 어떤 스크롤 막대 위젯의 `set()` 메서드이지만, 단일 인자를 취하는 어떤 위젯 메서

드도 가능합니다.

wrap
"none", "char" 또는 "word" 중 하나여야 합니다.

바인딩과 이벤트

위젯 명령의 bind 메서드를 사용하면 특정 이벤트를 감시하고 해당 이벤트 유형이 발생할 때 콜백 함수가 트리거 되도록 할 수 있습니다. bind 메서드의 형식은 다음과 같습니다:

```
def bind(self, sequence, func, add=''):
```

여기에서:

sequence
는 대상 이벤트의 종류를 나타내는 문자열입니다. (자세한 내용은 *bind(3tk)* 매뉴얼 페이지와 John Ousterhout의 책 *Tcl and the Tk Toolkit* (*2nd edition*)의 201쪽을 참조하십시오).

func
는 하나의 인자를 취하는 파이썬 함수로, 이벤트가 발생할 때 호출됩니다. 이벤트 인스턴스가 인자로 전달됩니다. (이런 식으로 설치되는 함수를 흔히 콜백(*callbacks*)이라고 합니다.)

add
는 선택적이고, ''나 '+'입니다. 빈 문자열을 전달하면 이 바인딩이 이 이벤트와 연관된 다른 바인딩을 대체 함을 나타냅니다. '+'를 전달하면 이 함수가 이 이벤트 유형에 바인딩 된 함수 목록에 추가됩니다.

예를 들면:

```
def turn_red(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turn_red)
```

이벤트의 widget 필드가 turn_red() 콜백에서 어떻게 액세스 되는지 주목하십시오. 이 필드는 X 이벤트를 포착한 위젯을 포함합니다. 다음 표에는 사용자가 액세스할 수 있는 다른 이벤트 필드와 Tk에서 이들을 표시하는 방법이 나열되어 있습니다. Tk 매뉴얼 페이지를 참조할 때 유용할 수 있습니다.

Tk	Tkinter 이벤트 필드	Tk	Tkinter 이벤트 필드
%f	focus	%A	char
%h	height	%E	send_event
%k	keycode	%K	keysym
%s	state	%N	keysym_num
%t	time	%T	type
%w	width	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

index 매개 변수

많은 위젯에는 "index" 매개 변수가 전달되어야 합니다. 이들은 Text 위젯의 특정 위치, Entry 위젯의 특정 문자 또는 Menu 위젯의 특정 메뉴 항목을 가리키는 데 사용됩니다.

Entry 위젯 인덱스 (인덱스, 뷰 인덱스 등)
Entry 위젯에는 표시되는 텍스트의 문자 위치를 참조하는 옵션이 있습니다. 다음 *tkinter* 함수를 사용하여 텍스트 위젯에서 이러한 특수 지점에 액세스할 수 있습니다:

Text 위젯 인덱스
Text 위젯의 인덱스 표기법은 매우 풍부하며 Tk 매뉴얼 페이지에 자세히 설명되어 있습니다.

메뉴 인덱스 (menu.invoke(), menu.entryconfig() 등)

메뉴에 대한 일부 옵션 및 메서드는 특정 메뉴 항목을 조작합니다. 옵션이나 매개 변수에 메뉴 인덱스가 필요한 때는 언제든지 다음과 같이 전달할 수 있습니다:

- 위에서부터 세고, 0에서 시작하는, 위젯에서 항목의 숫자 위치를 나타내는 정수.
- 현재 커서 아래에 있는 메뉴 위치를 나타내는, 문자열 `"active"`.
- 마지막 메뉴 항목을 나타내는, 문자열 `"last"`.
- `@6`과 같이, `@`이 앞에 오는 정수로, 정수는 메뉴의 좌표계에서 y 픽셀 좌표로 해석됩니다.
- 아무런 메뉴 항목을 가리키지 않는, 문자열 `"none"`은 menu.activate()와 함께 사용되어 모든 항목을 비활성화합니다, 마지막으로,
- 메뉴 맨 위에서 아래로 스캔할 때, 메뉴 항목의 레이블과 패턴 일치하는 텍스트 문자열. 이 인덱스 유형은 다른 모든 항목 다음에 고려되므로, `last`, `active` 또는 `none` 레이블이 붙은 메뉴 항목과의 일치가 대신 위의 리터럴로 해석될 수 있음을 의미합니다.

이미지

서로 다른 형식의 이미지를 `tkinter.Image`의 해당 서브 클래스를 통해 만들 수 있습니다:

- XBM 형식의 이미지를 위한 `BitmapImage`.
- PGM, PPM, GIF 및 PNG 형식의 이미지를 위한 `PhotoImage`. 후자는 Tk 8.6부터 지원됩니다.

두 가지 유형의 이미지는 `file` 또는 `data` 옵션을 통해 만들어집니다 (다른 옵션도 사용할 수 있습니다).

버전 3.13에서 변경: Added the `PhotoImage` method `copy_replace()` to copy a region from one image to other image, possibly with pixel zooming and/or subsampling. Add *from_coords* parameter to `PhotoImage` methods `copy()`, `zoom()` and `subsample()`. Add *zoom* and *subsample* parameters to `PhotoImage` method `copy()`.

그런 다음 `image` 옵션이 일부 위젯(예를 들어, 레이블, 버튼, 메뉴)에서 지원되는 곳이면 어디든 이미지 객체를 사용할 수 있습니다. 이 경우, Tk는 이미지에 대한 참조를 유지하지 않습니다. 이미지 객체에 대한 마지막 파이썬 참조가 삭제되면 이미지 데이터도 삭제되고, Tk는 이미지가 사용된 곳마다 빈 상자를 표시합니다.

> **더 보기**
>
> Pillow 패키지는 BMP, JPEG, TIFF 및 WebP와 같은 형식을 위한 지원을 추가합니다.

25.1.6 파일 처리기

Tk는 파일 기술자에서 I/O가 가능할 때 Tk 메인 루프에서 호출할 콜백 함수를 등록하고 등록 취소할 수 있도록 합니다. 파일 기술자당 하나의 처리기 만 등록 될 수 있습니다. 예제 코드:

```
import tkinter
widget = tkinter.Tk()
mask = tkinter.READABLE | tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

윈도우에서는 이 기능을 사용할 수 없습니다.

얼마나 많은 바이트를 읽을 수 있는지 모르므로, *BufferedIOBase*나 *TextIOBase* *read()*나 *readline()* 메서드를 사용하고 싶지 않을것입니다, 이것들은 미리 정의 된 바이트 수를 읽으려고하기 때문입니다. 소켓의 경우, *recv()* 또는 *recvfrom()* 메서드가 제대로 작동합니다; 다른 파일의 경우, 날(raw) 읽기나 `os.read(file.fileno(), maxbytecount)`를 사용하십시오.

Widget.tk.**createfilehandler**(*file*, *mask*, *func*)

파일 처리기 콜백 함수 *func*를 등록합니다. *file* 인자는 `fileno()` 메서드가 있는 객체이거나(가령 파일이나 소켓 객체), 정수 파일 기술자일 수 있습니다. *mask* 인자는 아래의 세 가지 상수들을 OR로 조합한 것입니다. 콜백은 다음과 같이 호출됩니다:

```
callback(file, mask)
```

Widget.tk.**deletefilehandler**(*file*)

파일 처리기를 등록 취소합니다.

_tkinter.**READABLE**

_tkinter.**WRITABLE**

_tkinter.**EXCEPTION**

mask 인자에 사용되는 상수입니다.

25.2 tkinter.colorchooser — 색상 선택 대화 상자

소스 코드: Lib/tkinter/colorchooser.py

`tkinter.colorchooser` 모듈은 네이티브 색상 선택기 대화 상자의 인터페이스로 `Chooser` 클래스를 제공합니다. `Chooser`는 모달(modal) 색상 선택 대화 상자 창을 구현합니다. `Chooser` 클래스는 `Dialog` 클래스를 상속합니다.

class tkinter.colorchooser.**Chooser**(*master=None*, ***options*)

tkinter.colorchooser.**askcolor**(*color=None*, ***options*)

색상 선택 대화 상자를 만듭니다. 이 메서드를 호출하면 창이 표시되고, 사용자가 선택하기를 기다렸다가, 선택한 색상(또는 `None`)을 호출자에게 반환합니다.

더 보기

모듈 ***tkinter.commondialog***
Tkinter 표준 대화 상자 모듈

25.3 tkinter.font — Tkinter 글꼴 래퍼

소스 코드: Lib/tkinter/font.py

`tkinter.font` 모듈은 명명된 글꼴을 만들고 사용하기 위한 `Font` 클래스를 제공합니다.

구별되는 글꼴 무게와 기울기는 다음과 같습니다:

tkinter.font.**NORMAL**

tkinter.font.**BOLD**

tkinter.font.**ITALIC**

tkinter.font.**ROMAN**

class tkinter.font.**Font**(*root=None*, *font=None*, *name=None*, *exists=False*, ***options*)

`Font` 클래스는 명명된 글꼴을 나타냅니다. *Font* 인스턴스에는 고유한 이름이 지정되며 패밀리, 크기 및 스타일 구성으로 지정할 수 있습니다. 명명된 글꼴은 나타날 때마다 어트리뷰트로 글꼴을 지정하지 않고, 글꼴을 단일 객체로 만들고 식별하는 Tk의 방법입니다.

인자:

font - 글꼴 지정자 튜플 (패밀리, 크기, 옵션)
name - 고유한 글꼴 이름
exists - 참이면 self가 기존의 명명된 글꼴을 가리킵니다

추가 키워드 옵션 (*font*가 지정되면 무시됩니다):

family - 글꼴 패밀리, 즉 Courier, Times
size - 글꼴 크기
*size*가 양수이면 포인트 단위의 크기로 해석됩니다.
*size*가 음수이면 절댓값을
픽셀 단위의 크기로 처리합니다.
weight - 글꼴 강조 (NORMAL, BOLD)
slant - ROMAN, ITALIC
underline - 글꼴 밑줄 (0 - 없음, 1 - 밑줄)
overstrike - 글꼴 취소선 (0 - 없음, 1 - 취소선)

actual(*option=None*, *displayof=None*)
글꼴의 어트리뷰트를 반환합니다.

cget(*option*)
글꼴의 어트리뷰트를 가져옵니다.

config(***options*)
글꼴의 어트리뷰트를 수정합니다.

copy()
현재 글꼴의 새 인스턴스를 반환합니다.

measure(*text*, *displayof=None*)
현재 글꼴로 포맷할 때 지정된 디스플레이에서 텍스트가 차지할 공간의 크기를 반환합니다. 디스플레이가 지정되지 않으면 메인 응용 프로그램 창을 가정합니다.

metrics(**options*, ***kw*)
글꼴별 데이터를 반환합니다. 옵션은 다음과 같습니다:

***ascent* - 기준선(baseline)과 글꼴의 문자가 차지할 수 있는 가장 높은 점 사이의 거리**
.

***descent* - 기준선과 글꼴의 문자가 차지할 수 있는 가장 낮은 점 사이의 거리**
.

***linespace* - 줄 사이에 수직 겹침이 없음을 보장하는, 글꼴의 임의의 두 문자 간에 필요한 최소 수직 분리**
.

fixed - 글꼴이 고정 너비이면 1, 그렇지 않으면 0

tkinter.font.**families**(*root=None*, *displayof=None*)
구별되는 글꼴 패밀리를 반환합니다.

tkinter.font.**names**(*root=None*)
정의된 글꼴의 이름을 반환합니다.

tkinter.font.**nametofont**(*name*, *root=None*)
tk 명명된 글꼴의 `Font` 표현을 반환합니다.

버전 3.10에서 변경: The *root* parameter was added.

25.4 Tkinter 대화 상자

25.4.1 `tkinter.simpledialog` — 표준 Tkinter 입력 대화 상자

소스 코드: Lib/tkinter/simpledialog.py

`tkinter.simpledialog` 모듈에는 사용자로부터 값을 얻기 위한 간단한 모달 대화 상자를 만드는 편의 클래스와 함수가 포함되어 있습니다.

`tkinter.simpledialog.askfloat`(*title*, *prompt*, ***kw*)

`tkinter.simpledialog.askinteger`(*title*, *prompt*, ***kw*)

`tkinter.simpledialog.askstring`(*title*, *prompt*, ***kw*)

위의 세 함수는 사용자에게 원하는 형의 값을 입력하도록 요구하는 대화 상자를 제공합니다.

class `tkinter.simpledialog.Dialog`(*parent*, *title=None*)

사용자 정의 대화 상자의 베이스 클래스.

body(*master*)

대화 상자의 인터페이스를 구성하고 초기 포커스가 필요한 위젯을 반환하도록 재정의하십시오.

buttonbox()

기본 동작은 OK 와 Cancel 버튼을 추가합니다. 사용자 정의 버튼 레이아웃이 필요하면 재정의하십시오.

25.4.2 `tkinter.filedialog` — 파일 선택 대화 상자

소스 코드: Lib/tkinter/filedialog.py

`tkinter.filedialog` 모듈은 파일/디렉터리 선택 창을 만들기위한 클래스와 팩토리 함수를 제공합니다.

네이티브 로드/저장 대화 상자

다음 클래스와 함수는 네이티브 모양과 느낌을 동작을 사용자 정의하는 구성 옵션과 결합하는 파일 대화 상자 창을 제공합니다. 다음 키워드 인자는 아래 나열된 클래스와 함수에 적용할 수 있습니다:

parent - 대화 상자를 그 위에 놓을 창

title - 창의 제목

initialdir - 대화 상자가 시작되는 디렉터리

initialfile - 대화 상자를 열 때 선택된 파일

filetypes - (label, pattern) 튜플의 시퀀스, '*' 와일드카드가 허용됩니다

defaultextension - 파일에 추가할 기본 확장자 (저장 대화 상자)

multiple - 참일 때, 여러 항목을 선택할 수 있습니다

정적 팩토리 함수

아래 함수는 호출될 때 모달, 네이티브 모양과 느낌의 대화 상자를 만들고, 사용자의 선택을 기다린 다음, 선택한 값이나 `None`을 호출자에게 반환합니다.

`tkinter.filedialog.askopenfile`(*mode='r'*, ***options*)

tkinter.filedialog.**askopenfiles**(*mode='r', **options*)

위의 두 함수는 `Open` 대화 상자를 만들고 열린 파일 객체를 읽기 전용 모드로 반환합니다.

tkinter.filedialog.**asksaveasfile**(*mode='w', **options*)

`SaveAs` 대화 상자를 만들고 쓰기 전용 모드로 열린 파일 객체를 반환합니다.

tkinter.filedialog.**askopenfilename**(***options*)

tkinter.filedialog.**askopenfilenames**(***options*)

위의 두 함수는 `Open` 대화 상자를 만들고 기존 파일(들)에 해당하는 선택된 파일명(들)을 반환합니다.

tkinter.filedialog.**asksaveasfilename**(***options*)

`SaveAs` 대화 상자를 만들고 선택한 파일명을 반환합니다.

tkinter.filedialog.**askdirectory**(***options*)

사용자에게 디렉터리를 선택하라는 메시지를 표시합니다.

추가 키워드 옵션:

mustexist - 선택이 기존 디렉터리여야 하는지를 결정합니다.

class tkinter.filedialog.**Open**(*master=None, **options*)

class tkinter.filedialog.**SaveAs**(*master=None, **options*)

위의 두 클래스는 파일 저장과 로드를 위한 네이티브 대화 창을 제공합니다.

편의 클래스

아래 클래스는 파일/디렉터리 창을 처음부터 만드는 데 사용됩니다. 이것들은 플랫폼의 네이티브 모양과 느낌을 모방하지 않습니다.

class tkinter.filedialog.**Directory**(*master=None, **options*)

사용자에게 디렉터리를 선택하라는 대화 상자를 만듭니다.

> **참고**
>
> *FileDialog* 클래스는 사용자 정의 이벤트 처리와 동작을 위해 서브 클래싱 되어야 합니다.

class tkinter.filedialog.**FileDialog**(*master, title=None*)

기본 파일 선택 대화 상자를 만듭니다.

cancel_command(*event=None*)

대화 창의 종료를 트리거 합니다.

dirs_double_event(*event*)

디렉터리에 대한 더블 클릭 이벤트를 위한 이벤트 처리기.

dirs_select_event(*event*)

디렉터리에 대한 클릭 이벤트를 위한 이벤트 처리기.

files_double_event(*event*)

파일에 대한 더블 클릭 이벤트를 위한 이벤트 처리기.

files_select_event(*event*)

파일에 대한 단일 클릭 이벤트를 위한 이벤트 처리기.

filter_command(*event=None*)

디렉터리로 파일을 필터링합니다.

get_filter()

현재 사용 중인 파일 필터를 가져옵니다.

get_selection()

현재 선택된 항목을 가져옵니다.

go(*dir_or_file=os.curdir*, *pattern='*'*, *default=''*, *key=None*)
대화 상자를 렌더링하고 이벤트 루프를 시작합니다.

ok_event(*event*)
현재 선택을 반환하면서 대화 상자를 종료합니다.

quit(*how=None*)
파일명을 (있다면) 반환하면서 대화 상자를 종료합니다.

set_filter(*dir*, *pat*)
파일 필터를 설정합니다.

set_selection(*file*)
현재 파일 선택을 *file*로 갱신합니다.

class tkinter.filedialog.**LoadFileDialog**(*master*, *title=None*)
기존 파일을 선택하기 위한 대화 상자 창을 만드는 FileDialog의 서브 클래스.

ok_command()
파일이 제공되고 선택이 이미 존재하는 파일을 가리키는지 테스트합니다.

class tkinter.filedialog.**SaveFileDialog**(*master*, *title=None*)
대상 파일을 선택하기 위한 대화 상자 창을 만드는 FileDialog의 서브 클래스.

ok_command()
선택이 디렉터리가 아닌 유효한 파일을 가리키는지 테스트합니다. 이미 존재하는 파일을 선택했으면 확인이 필요합니다.

25.4.3 tkinter.commondialog — 대화창 템플릿

소스 코드: Lib/tkinter/commondialog.py

---

tkinter.commondialog 모듈은 다른 지원 모듈에 정의된 대화 상자의 베이스 클래스인 *Dialog* 클래스를 제공합니다.

class tkinter.commondialog.**Dialog**(*master=None*, ***options*)

show(***options*)
대화창을 렌더링합니다.

더 보기

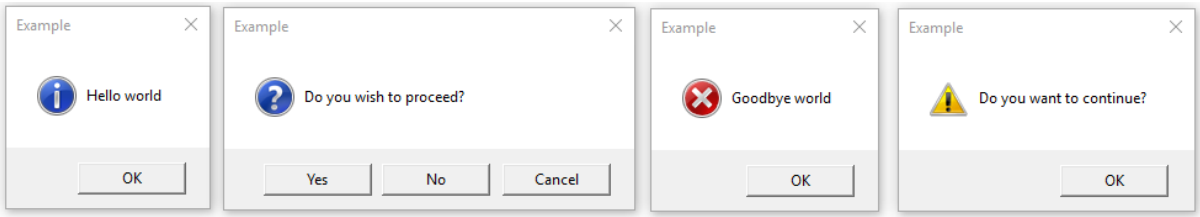
모듈 *tkinter.messagebox*, tut-files

25.5 tkinter.messagebox — Tkinter 메시지 프롬프트

소스 코드: Lib/tkinter/messagebox.py

---

tkinter.messagebox 모듈은 일반적으로 사용되는 구성을 위한 다양한 편의 메서드 뿐만 아니라 템플릿 베이스 클래스를 제공합니다. 메시지 상자는 모달(modal)이며 사용자 선택에 따라 (`True`, `False`, `None`, *OK*, *CANCEL*, *YES*, *NO*)의 부분 집합을 반환합니다. 일반적인 메시지 상자 스타일과 레이아웃에는 다음이 포함되지만 이에 국한되지는 않습니다:



class `tkinter.messagebox.`**`Message`**(*master=None*, ***options*)

Create a message window with an application-specified message, an icon and a set of buttons. Each of the buttons in the message window is identified by a unique symbolic name (see the *type* options).

The following options are supported:

command
Specifies the function to invoke when the user closes the dialog. The name of the button clicked by the user to close the dialog is passed as argument. This is only available on macOS.

default
Gives the *symbolic name* of the default button for this message window (`OK`, `CANCEL`, and so on). If this option is not specified, the first button in the dialog will be made the default.

detail
Specifies an auxiliary message to the main message given by the *message* option. The message detail will be presented beneath the main message and, where supported by the OS, in a less emphasized font than the main message.

icon
Specifies an *icon* to display. If this option is not specified, then the `INFO` icon will be displayed.

message
Specifies the message to display in this message box. The default value is an empty string.

parent
Makes the specified window the logical parent of the message box. The message box is displayed on top of its parent window.

title
Specifies a string to display as the title of the message box. This option is ignored on macOS, where platform guidelines forbid the use of a title on this kind of dialog.

type
Arranges for a *predefined set of buttons* to be displayed.

show(***options*)

Display a message window and wait for the user to select one of the buttons. Then return the symbolic name of the selected button. Keyword arguments can override options specified in the constructor.

정보 메시지 상자

`tkinter.messagebox.`**`showinfo`**(*title=None*, *message=None*, ***options*)

지정된 제목과 메시지가 포함된 정보 메시지 상자를 만들고 표시합니다.

경고 메시지 상자

`tkinter.messagebox.`**`showwarning`**(*title=None*, *message=None*, ***options*)

Creates and displays a warning message box with the specified title and message.

`tkinter.messagebox.`**`showerror`**(*title=None*, *message=None*, ***options*)

Creates and displays an error message box with the specified title and message.

질문 메시지 상자

tkinter.messagebox.**askquestion**(*title=None, message=None, *, type=YESNO, **options*)

Ask a question. By default shows buttons *YES* and *NO*. Returns the symbolic name of the selected button.

tkinter.messagebox.**askokcancel**(*title=None, message=None, **options*)

Ask if operation should proceed. Shows buttons *OK* and *CANCEL*. Returns `True` if the answer is ok and `False` otherwise.

tkinter.messagebox.**askretrycancel**(*title=None, message=None, **options*)

Ask if operation should be retried. Shows buttons *RETRY* and *CANCEL*. Return `True` if the answer is yes and `False` otherwise.

tkinter.messagebox.**askyesno**(*title=None, message=None, **options*)

Ask a question. Shows buttons *YES* and *NO*. Returns `True` if the answer is yes and `False` otherwise.

tkinter.messagebox.**askyesnocancel**(*title=None, message=None, **options*)

Ask a question. Shows buttons *YES*, *NO* and *CANCEL*. Return `True` if the answer is yes, `None` if cancelled, and `False` otherwise.

Symbolic names of buttons:

tkinter.messagebox.**ABORT = 'abort'**

tkinter.messagebox.**RETRY = 'retry'**

tkinter.messagebox.**IGNORE = 'ignore'**

tkinter.messagebox.**OK = 'ok'**

tkinter.messagebox.**CANCEL = 'cancel'**

tkinter.messagebox.**YES = 'yes'**

tkinter.messagebox.**NO = 'no'**

Predefined sets of buttons:

tkinter.messagebox.**ABORTRETRYIGNORE = 'abortretryignore'**

Displays three buttons whose symbolic names are *ABORT*, *RETRY* and *IGNORE*.

tkinter.messagebox.**OK = 'ok'**

Displays one button whose symbolic name is *OK*.

tkinter.messagebox.**OKCANCEL = 'okcancel'**

Displays two buttons whose symbolic names are *OK* and *CANCEL*.

tkinter.messagebox.**RETRYCANCEL = 'retrycancel'**

Displays two buttons whose symbolic names are *RETRY* and *CANCEL*.

tkinter.messagebox.**YESNO = 'yesno'**

Displays two buttons whose symbolic names are *YES* and *NO*.

tkinter.messagebox.**YESNOCANCEL = 'yesnocancel'**

Displays three buttons whose symbolic names are *YES*, *NO* and *CANCEL*.

Icon images:

tkinter.messagebox.**ERROR = 'error'**

tkinter.messagebox.**INFO = 'info'**

tkinter.messagebox.**QUESTION = 'question'**

tkinter.messagebox.**WARNING = 'warning'**

25.6 tkinter.scrolledtext — 스크롤 되는 Text 위젯

소스 코드: Lib/tkinter/scrolledtext.py

---

tkinter.scrolledtext 모듈은 "올바로" 동작하도록 구성된 수직 스크롤 막대가 있는 기본 텍스트 위젯을 구현하는 같은 이름의 클래스를 제공합니다. *ScrolledText* 클래스를 사용하면 텍스트 위젯과 스크롤 막대를 직접 설정하기보다 훨씬 쉽습니다.

텍스트 위젯과 스크롤 막대는 `Frame`에 함께 팩 되며, `Grid`와 `Pack` 지오메트리 관리자의 메서드를 `Frame` 객체에서 얻습니다. 이렇게 함으로써, 가장 일반적인 지오메트리 관리 동작을 달성하는데 *ScrolledText* 위젯을 직접 사용할 수 있습니다.

더욱 구체적인 제어가 필요하면, 다음 어트리뷰트를 사용할 수 있습니다:

class tkinter.scrolledtext.**ScrolledText**(*master=None*, ***kw*)

frame

텍스트 및 스크롤 막대 위젯을 둘러싼 프레임.

vbar

스크롤 막대 위젯.

25.7 tkinter.dnd — 드래그 앤드 드롭 지원

소스 코드: Lib/tkinter/dnd.py

> **참고**
>
> 이것은 실험적이며 Tk DND로 대체될 때 폐지될 예정입니다.

tkinter.dnd 모듈은 단일 응용 프로그램 내에서, 같은 창 내에서 또는 창 간에 객체에 대해 드래그 앤드 드롭(끌어서 놓기) 지원을 제공합니다. 객체를 드래그할 수 있게 하려면, 드래그 앤드 드롭 프로세스를 시작하는 이벤트 바인딩을 만들어야 합니다. 일반적으로, ButtonPress 이벤트를 여러분이 작성한 콜백 함수에 바인딩합니다 (바인딩과 이벤트를 참조하십시오). 이 함수는 *dnd_start()*를 호출해야 합니다, 여기서 'source' 는 드래그할 객체이고, 'event' 는 호출을 일으킨 이벤트입니다 (콜백 함수의 인자입니다).

대상 객체는 다음과 같이 선택됩니다:

1. 대상 위젯에 대한 마우스 아래 영역의 하향식 검색
   - 대상 위젯에는 콜러블 *dnd_accept* 어트리뷰트가 있어야 합니다
   - *dnd_accept*가 없거나 `None`을 반환하면, 검색은 부모 위젯으로 이동합니다
   - 대상 위젯이 발견되지 않으면, 대상 객체는 `None`입니다.
2. *<old_target>.dnd_leave(source, event)*를 호출합니다
3. *<new_target>.dnd_enter(source, event)*를 호출합니다
4. 드롭을 알리기 위해 *<target>.dnd_commit(source, event)*를 호출합니다
5. 드래그 앤드 드롭의 끝을 알리기 위해 *<source>.dnd_end(target, event)*를 호출합니다

class tkinter.dnd.**DndHandler**(*source*, *event*)

DndHandler 클래스는 이벤트 위젯의 루트에서 Motion과 ButtonRelease 이벤트를 추적하는 드래그 앤드 드롭 이벤트를 처리합니다.

cancel(*event=None*)

드래그 앤드 드롭 프로세스를 취소합니다.

finish(*event*, *commit=0*)

드래그 앤드 드롭 기능의 끝을 실행합니다.

on_motion(*event*)

드래그가 수행되는 동안 마우스 아래의 대상 객체를 검사합니다.

on_release(*event*)

릴리즈 패턴이 트리거 될 때 드래그의 끝을 알립니다.

tkinter.dnd.**dnd_start**(*source*, *event*)

드래그 앤드 드롭 프로세스를 위한 팩토리 함수.

더 보기

바인딩과 이벤트

25.8 tkinter.ttk — Tk 테마 위젯

소스 코드: Lib/tkinter/ttk.py

---

tkinter.ttk 모듈은 Tk 8.5에 도입된 Tk 테마 위젯 집합에 대한 액세스를 제공합니다. X11에서 안티 에일리어싱 된 글꼴 렌더링과 창 투명도(X11에서 컴포지션 창 관리자가 필요합니다)를 포함한 추가 이점을 제공합니다.

*tkinter.ttk*의 기본 아이디어는 위젯 동작을 구현하는 코드와 모양을 구현하는 코드를 가능한 한 분리하는 것입니다.

더 보기

Tk 위젯 스타일링 지원
Tk의 테마 지원을 소개하는 문서

25.8.1 Ttk 사용하기

Ttk를 사용하려면, 모듈을 임포트 하십시오:

```
from tkinter import ttk
```

기본 Tk 위젯을 재정의하려면, 임포트는 Tk 임포트 뒤에 와야 합니다:

```
from tkinter import *
from tkinter.ttk import *
```

이 코드로 인해 여러 *tkinter.ttk* 위젯(`Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale` 및 `Scrollbar`)이 Tk 위젯을 자동으로 대체합니다.

이것은 여러 플랫폼에서 더 나은 모양과 느낌을 제공하는 새로운 위젯을 사용하는 직접적인 이점이 있지만, 대체된 위젯은 완전히 호환되지 않습니다. 주요 차이점은 "fg", "bg" 및 위젯 스타일과 관련된 다른 위젯 옵션이 더는 Ttk 위젯에 존재하지 않는다는 것입니다. 대신, 스타일링 효과를 개선하려면 `ttk.Style` 클래스를 사용하십시오.

더 보기

Tile 위젯을 사용하도록 기존 응용 프로그램을 변환하기
새 위젯을 사용하도록 응용 프로그램을 바꿀 때 일반적으로 발생하는 차이점에 대한 모노그래프 (Tcl 용어로).

25.8.2 Ttk 위젯

Ttk에는 18개의 위젯이 있으며, 그중 12개는 tkinter에 이미 존재합니다: `Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale`, `Scrollbar` 및 *Spinbox*. 다른 6개는 *Combobox*, *Notebook*, *Progressbar*, `Separator`, `Sizegrip` 및 *Treeview*입니다. 그리고 이들은 모두 *Widget*의 서브 클래스입니다.

Ttk 위젯을 사용하면 응용 프로그램의 모양과 느낌이 개선됩니다. 위에서 설명한 것처럼, 스타일을 코딩하는 방법에는 차이가 있습니다.

Tk 코드:

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Ttk 코드:

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

*Tk 스타일링*에 대한 자세한 정보는, *Style* 클래스 설명서를 참조하십시오.

25.8.3 Widget

`ttk.Widget`은 Tk 테마 위젯이 지원하는 표준 옵션과 메서드를 정의하며 직접 인스턴스 화하지 않습니다.

표준 옵션

모든 `ttk` 위젯은 다음 옵션을 받아들입니다:

옵션	설명
class	윈도우 클래스를 지정합니다. 이 클래스는 창의 다른 옵션에 대한 옵션 데이터베이스를 조회하고, 창의 기본 바인딩 태그를 결정하고, 위젯의 기본 레이아웃과 스타일을 선택하는 데 사용됩니다. 이 옵션은 읽기 전용이며, 창을 만들 때만 지정할 수 있습니다.
cursor	위젯에 사용될 마우스 커서를 지정합니다. 빈 문자열(기본값)로 설정하면, 커서가 부모 위젯에서 상속됩니다.
takefocus	키보드 순회 중 창에서 포커스를 받아들일지를 결정합니다. 0, 1 또는 빈 문자열이 반환됩니다. 0이 반환되면, 키보드 순회 중에 창을 완전히 건너뛰어야 한다는 의미입니다. 1이면, 창은 볼 수 있는 한 입력 포커스를 받아야 함을 의미합니다. 빈 문자열은 순회 스크립트가 창에 포커스를 맞출지를 결정함을 의미합니다.
style	사용자 정의 위젯 스타일을 지정하는 데 사용될 수 있습니다.

스크롤 가능한 위젯 옵션

다음 옵션은 스크롤 막대로 제어되는 위젯에서 지원됩니다.

옵션	설명
xscrollcommand	가로 스크롤 막대와 통신하는 데 사용됩니다. 위젯 창에 있는 뷰가 변경될 때, 위젯은 scrollcommand를 기반한 Tcl 명령을 생성합니다. 일반적으로 이 옵션은 어떤 스크롤 막대의 메서드 `Scrollbar.set()` 으로 구성됩니다. 그러면 창의 뷰가 변경될 때마다 스크롤 막대가 갱신됩니다.
yscrollcommand	세로 스크롤 막대와 통신하는 데 사용됩니다. 자세한 내용은 위를 참조하십시오.

레이블 옵션

다음 옵션은 레이블, 버튼 및 기타 버튼류 위젯에서 지원됩니다.

옵션	설명
text	위젯 안에 표시할 텍스트 문자열을 지정합니다.
textvariable	text 옵션 자원 대신 그 값이 사용될 이름을 지정합니다.
underline	설정되면, 텍스트 문자열에서 밑줄 그을 문자의 인덱스(0에서 시작)를 지정합니다. 밑줄 있는 문자는 니모닉 활성화(mnemonic activation)에 사용됩니다.
image	표시할 이미지를 지정합니다. 이것은 하나 이상의 요소로 구성된 리스트입니다. 첫 번째 요소는 기본 이미지 이름입니다. 리스트의 나머지가 *`Style.map()`* 에 의해 정의된 대로 statespec/value 쌍의 시퀀스면, 위젯이 특정 상태나 상태의 조합에 있을 때 사용할 다른 이미지를 지정합니다. 리스트의 모든 이미지는 크기가 같아야 합니다.
compound	텍스트와 이미지 옵션이 모두 있을 때, 텍스트를 기준으로 이미지를 표시하는 방법을 지정합니다. 유효한 값은 다음과 같습니다: • text: 텍스트만 표시합니다 • image: 이미지만 표시합니다 • top, bottom, left, right: 각각 이미지를 텍스트 위, 아래, 왼쪽 또는 오른쪽에 표시합니다. • none: 기본값입니다. 있으면 이미지를 표시하고, 그렇지 않으면 텍스트를 표시합니다.
width	0보다 크면, 문자 너비 단위로, 텍스트 레이블에 할당할 공간의 크기를 지정합니다, 0보다 작으면, 최소 너비를 지정합니다. 0으로 지정하거나 지정하지 않으면, 텍스트 레이블의 자연 너비가 사용됩니다.

호환성 옵션

옵션	설명
state	"disabled" 상태 비트를 제어하기 위해 "normal" 이나 "disabled" 로 설정될 수 있습니다. 이것은 쓰기 전용 옵션입니다: 이를 설정하면 위젯 상태가 변경되지만, *`Widget.state()`* 메서드는 이 옵션에 영향을 미치지 않습니다.

위젯 상태

위젯 상태는 독립 상태 플래그의 비트 맵입니다.

플래그	설명
active	마우스 커서가 위젯 위에 있으며 마우스 버튼을 누르면 어떤 동작을 일으킵니다
disabled	프로그램 제어 하에 위젯이 비활성화되었습니다
focus	위젯에 키보드 포커스가 있습니다
pressed	위젯을 누르고 있습니다
selected	체크 버튼과 라디오 버튼과 같은 항목의 경우 "On", "true" 또는 "current"
background	윈도우와 맥에는 "활성(active)" 이나 전경(foreground) 창이라는 개념이 있습니다. *background* 상태는 배경 창의 위젯에 대해 설정되고, 전경 창의 위젯에서는 지워집니다.
readonly	위젯이 사용자 수정을 허용하지 않습니다
alternate	위젯 별 대체 디스플레이 포맷
invalid	위젯 값이 유효하지 않습니다

상태 명세는 상태 이름의 시퀀스이며, 선택적으로 비트가 꺼져 있음을 나타내는 느낌표가 접두어로 붙습니다.

ttk.Widget

아래 설명된 메서드 외에도 `ttk.Widget` 은 메서드 `tkinter.Widget.cget()` 과 `tkinter.Widget.configure()` 를 지원합니다.

class `tkinter.ttk.`**`Widget`**

identify(*x*, *y*)

위치 *x y*에 있는 요소의 이름을 반환하거나, 점이 요소 내에 없으면 빈 문자열을 반환합니다.

x 와 *y*는 위젯에 상대적인 픽셀 좌표입니다.

instate(*statespec*, *callback=None*, **args*, ***kw*)

위젯 상태를 테스트합니다. 콜백을 지정하지 않으면, 위젯 상태가 *statespec*과 일치하면 `True`를, 그렇지 않으면 `False`를 반환합니다. 콜백이 지정되면 위젯 상태가 *statespec*과 일치하면 args로 호출됩니다.

state(*statespec=None*)

위젯 상태를 수정하거나 요청합니다. *statespec*이 지정되면, 그에 따라 위젯 상태를 설정하고 변경된 플래그를 나타내는 새 *statespec*을 반환합니다. *statespec*을 지정하지 않으면, 현재 활성화된 상태 플래그를 반환합니다.

*statespec*은 일반적으로 리스트나 튜플입니다.

25.8.4 Combobox

`ttk.Combobox` 위젯은 텍스트 필드를 값의 팝-다운(pop-down) 리스트와 결합합니다. 이 위젯은 `Entry`의 서브 클래스입니다.

*`Widget`*에서 상속된 메서드 `Widget.cget()`, `Widget.configure()`, *`Widget.identify()`*, *`Widget.instate()`* 및 *`Widget.state()`*와, `Entry`에서 상속된 메서드 `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.selection()`, `Entry.xview()` 외에, `ttk.Combobox`에 설명된 다른 메서드가 있습니다.

옵션

이 위젯은 다음과 같은 특정 옵션을 받아들입니다:

옵션	설명
exportselection	불리언 값. 설정되면, 위젯 선택은 창 관리자 선택에 연결됩니다 (예를 들어, Misc.selection_get 을 호출하여 반환할 수 있습니다).
justify	위젯 내에서 텍스트가 정렬되는 방식을 지정합니다. "left", "center" 또는 "right" 중 하나입니다.
height	팝-다운 목록 상자의 높이를 행 단위로 지정합니다.
postcommand	값을 표시하기 직전에 호출되는 (Misc.register로 등록할 수 있는) 스크립트. 표시할 값을 지정할 수 있습니다.
state	"normal", "readonly" 또는 "disabled" 중 하나. "readonly" 상태에서는, 값을 직접 편집할 수 없고, 사용자는 드롭다운 목록에서 값을 선택할 수만 있습니다. "normal" 상태에서는, 텍스트 필드를 직접 편집할 수 있습니다. "disabled" 상태에서는, 상호작용이 불가능합니다.
textvariable	값이 위젯 값에 연결된 이름을 지정합니다. 해당 이름과 관련된 값이 변경될 때마다, 위젯 값이 갱신되고, 그 반대도 마찬가지입니다. `tkinter.StringVar`를 참조하십시오.
values	드롭-다운 목록 상자에 표시할 값의 리스트를 지정합니다.
width	원하는 입력 창의 너비를 나타내는 정숫값을 위젯 글꼴의 평균 크기 문자 단위로 지정합니다.

가상 이벤트

콤보 박스 위젯은 사용자가 값 목록에서 요소를 선택할 때 **«ComboboxSelected»** 가상 이벤트를 생성합니다.

ttk.Combobox

class `tkinter.ttk.`**`Combobox`**

current(*newindex=None*)

*newindex*를 지정하면, 콤보 박스 값을 요소 위치 *newindex*로 설정합니다. 그렇지 않으면, 현재 값의 인덱스를 반환하거나 현재 값이 값 목록에 없으면 -1을 반환합니다.

get()

콤보 박스의 현재 값을 반환합니다.

set(*value*)

콤보 박스의 값을 *value*로 설정합니다.

25.8.5 Spinbox

`ttk.Spinbox` 위젯은 증가와 감소 화살표로 개선된 `ttk.Entry`입니다. 숫자나 문자열 값의 리스트에 사용할 수 있습니다. 이 위젯은 `Entry`의 서브 클래스입니다.

*`Widget`*에서 상속된 메서드 `Widget.cget()`, `Widget.configure()`, *`Widget.identify()`*, *`Widget.instate()`* 및 *`Widget.state()`*와, `Entry`에서 상속된 메서드 `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.xview()` 외에도, `ttk.Spinbox`에 설명된 다른 메서드가 있습니다.

옵션

이 위젯은 다음과 같은 특정 옵션을 받아들입니다:

옵션	설명
from	부동 소수점 값. 설정하면, 감소 버튼이 감소할 최솟값입니다. `from`은 파이썬 키워드이므로, 인자로 사용될 때 `from_`을 사용해야 합니다.
to	부동 소수점 값. 설정되면, 증가 버튼이 증가할 최댓값입니다.
increment	부동 소수점 값. 증가/감소 버튼이 값을 변경할 양을 지정합니다. 기본값은 1.0입니다.
values	문자열이나 부동 소수점 값의 시퀀스. 지정되면, 증가/감소 버튼은 숫자를 늘리거나 줄이는 것이 아니라, 이 시퀀스에서 항목을 순환합니다.
wrap	불리언 값. `True`이면, 증가와 감소 버튼이 각각 `to` 값에서 `from` 값으로, `from` 값에서 `to` 값으로 순환합니다.
format	문자열 값. 증가/감소 버튼으로 설정된 숫자의 포맷을 지정합니다. "%W.Pf" 형식이어야 합니다, 여기서 W는 값의 패딩 된 너비, P는 정밀도, 그리고 '%' 와 'f' 는 리터럴 입니다.
command	파이썬 콜러블. 증가나 감소 버튼 중 하나를 누를 때마다 인자 없이 호출됩니다.

가상 이벤트

스핀 박스 위젯은 사용자가 <Up>을 누를 때 **«Increment»** 가상 이벤트를, 사용자가 <Down>을 누를 때 **«Decrement»** 가상 이벤트를 생성합니다.

ttk.Spinbox

class `tkinter.ttk.`**Spinbox**

get()

스핀 박스의 현재 값을 반환합니다.

set(*value*)

스핀 박스의 값을 *value*로 설정합니다.

25.8.6 Notebook

Ttk Notebook 위젯은 창 모음을 관리하고 한 번에 하나의 창을 표시합니다. 각 자식 창은 사용자가 현재 표시된 창을 변경하도록 선택할 수 있는 탭과 연결되어 있습니다.

옵션

이 위젯은 다음과 같은 특정 옵션을 받아들입니다:

옵션	설명
height	존재하고 0보다 크면, 팬 영역(pane area)의 원하는 높이를 지정합니다 (내부 패딩이나 탭을 포함하지 않습니다). 그렇지 않으면, 모든 팬의 최대 높이가 사용됩니다.
padding	노트북 외부에 추가할 여분의 공간을 지정합니다. 패딩은 좌(left) 상(top) 우(right) 하(bottom) 최대 4개의 길이 명세 리스트입니다. 4개보다 적은 요소가 지정되면, bottom의 기본값은 top, right의 기본값은 left, top의 기본값은 left입니다.
width	존재하고 0보다 크면, 원하는 팬 영역 너비를 지정합니다 (내부 패딩을 포함하지 않습니다). 그렇지 않으면, 모든 팬의 최대 너비가 사용됩니다.

탭 옵션

탭에 대한 특정 옵션도 있습니다:

옵션	설명
state	"normal", "disabled" 또는 "hidden". "disabled" 이면, 탭을 선택할 수 없습니다. "hidden" 이면, 탭이 표시되지 않습니다.
sticky	자식 창이 창 영역 내에서 배치되는 방법을 지정합니다. 값은 0개 이상의 문자 "n", "s", "e" 또는 "w"를 포함하는 문자열입니다. 각 문자는 `grid()` 지오메트리 관리자에 따라, 자식 창이 붙을 변(북(north), 남(south), 동(east) 또는 서(west))을 나타냅니다.
padding	노트북과 이 팬 사이에 추가할 여분의 공간을 지정합니다. 문법은 이 위젯에서 사용하는 옵션 padding과 같습니다.
text	탭에 표시할 텍스트를 지정합니다.
image	탭에 표시할 이미지를 지정합니다. *Widget*에 설명된 옵션 image를 참조하십시오.
compound	옵션 text와 image가 모두 있을 때, 텍스트에 상대적으로 이미지를 표시하는 방법을 지정합니다. 유효한 값은 레이블 옵션을 참조하십시오.
underline	텍스트 문자열에서 밑줄 그을 문자의 인덱스(0에서 시작)를 지정합니다. *Notebook.enable_traversal()*이 호출되면 밑줄 있는 문자는 니모닉 활성화(mnemonic activation)에 사용됩니다.

탭 식별자

`ttk.Notebook`의 여러 가지 메서드에 존재하는 tab_id는 다음 형식 중 하나를 취할 수 있습니다:

- 0과 탭 수 사이의 정수
- 자식 창의 이름
- 탭을 식별하는 "@x,y" 형식의 위치 명세
- 현재 선택된 탭을 식별하는 리터럴 문자열 "current"
- 탭 수를 반환하는 리터럴 문자열 "end" (*Notebook.index()*에만 유효합니다)

가상 이벤트

이 위젯은 새 탭을 선택한 후 **«NotebookTabChanged»** 가상 이벤트를 생성합니다.

ttk.Notebook

class `tkinter.ttk.`**Notebook**

add(*child*, ***kw*)

노트북에 새 탭을 추가합니다.

창이 현재 노트북으로 관리되지만 숨겨져 있으면, 창은 이전 위치로 복원됩니다.

사용 가능한 옵션 목록은 탭 옵션을 참조하십시오.

forget(*tab_id*)

*tab_id*로 지정된 탭을 제거하고, 연관된 창을 매핑 취소하고 관리 취소합니다.

hide(*tab_id*)

*tab_id*로 지정된 탭을 숨깁니다.

탭은 표시되지 않지만, 관련 창은 노트북에 의해 계속 관리되고 구성이 기억됩니다. 숨겨진 탭은 *add()* 명령으로 복원될 수 있습니다.

identify(*x*, *y*)

위치 *x*, *y*의 탭 요소 이름을 반환하거나, 없으면 비어있는 문자열을 반환합니다.

index(*tab_id*)

*tab_id*로 지정된 탭의 숫자 인덱스를 반환하거나, *tab_id*가 문자열 "end" 이면 총 탭 수를 반환합니다.

insert (*pos*, *child*, ***kw*)

지정된 위치에 팬(pane)을 삽입합니다.

*pos*는 문자열 "end", 정수 인덱스 또는 관리되는 자식의 이름입니다. 노트북에서 *child*를 이미 관리하고 있으면, 지정된 위치로 옮깁니다.

사용 가능한 옵션 목록은 탭 옵션을 참조하십시오.

select (*tab_id=None*)

지정된 *tab_id*를 선택합니다.

연결된 자식 창이 표시되고, 이전에 선택한 창이 (다르다면) 매핑이 취소됩니다. *tab_id*가 생략되면, 현재 선택된 팬의 위젯 이름을 반환합니다.

tab (*tab_id*, *option=None*, ***kw*)

특정 *tab_id*의 옵션을 조회하거나 수정합니다.

*kw*가 제공되지 않으면, 탭 옵션값의 딕셔너리를 반환합니다. *option*이 지정되면, 해당 *option*의 값을 반환합니다. 그렇지 않으면, 옵션을 해당 값으로 설정합니다.

tabs ()

노트북에서 관리하는 창의 리스트를 반환합니다.

enable_traversal ()

이 노트북이 포함된 최상위 창의 키보드 순회를 활성화합니다.

이것은 노트북을 포함하는 최상위 창에 대한 바인딩을 다음과 같이 확장합니다:

- `Control-Tab`: 현재 선택된 탭 다음에 있는 탭을 선택합니다.
- `Shift-Control-Tab`: 현재 선택된 탭 앞에 있는 탭을 선택합니다.
- `Alt-K`: 여기서 *K*는 탭의 니모닉 (밑줄이 그어진) 문자이며, 해당 탭을 선택합니다.

중첩된 노트북을 포함하여, 단일 최상위 수준에 있는 여러 노트북의 순회를 활성화할 수 있습니다. 그러나, 모든 팬에 마스터로 있는 노트북이 있을 때만 노트북 순회가 제대로 작동합니다.

25.8.7 Progressbar

`ttk.Progressbar` 위젯은 장기 실행 작업의 상태를 보여줍니다. 두 가지 모드로 동작할 수 있습니다: 1) 수행할 총작업량을 기준으로 완료된 양을 표시하는 확정적(determinate) 모드와 2) 작업이 진행 중임을 사용자에게 알리는 애니메이션 표시를 제공하는 불확정적(indeterminate) 모드입니다.

옵션

이 위젯은 다음과 같은 특정 옵션을 받아들입니다:

옵션	설명
orient	"horizontal" 또는 "vertical" 중 하나입니다. 진행 막대의 방향을 지정합니다.
length	진행 막대의 긴 축의 길이를 지정합니다 (가로면 너비, 세로면 높이).
mode	"determinate" 또는 "indeterminate" 중 하나.
maximum	최댓값을 지정하는 숫자. 기본값은 100입니다.
value	진행 막대의 현재 값. "determinate" (확정적) 모드에서는, 완료된 작업량을 나타냅니다. "indeterminate" (불확정적) 모드에서는, 모듈로 *maximum*으로 해석됩니다; 즉, 진행 막대의 값이 *maximum* 증가하면 하나의 "사이클"이 완료됩니다.
variable	옵션값에 연결된 이름. 지정되면, 진행 막대의 값은 이 이름의 값이 수정될 때마다 이 이름의 값으로 자동 설정됩니다.
phase	읽기 전용 옵션. 위젯은 값이 0보다 크고 확정적 모드에서 최댓값보다 작을 때마다 이 옵션의 값을 주기적으로 증가시킵니다. 이 옵션은 현재 테마에서 추가 애니메이션 효과를 제공하는 데 사용할 수 있습니다.

ttk.Progressbar

class tkinter.ttk.**Progressbar**

start(*interval=None*)

자동 증가 모드를 시작합니다: *interval* 밀리초마다 `Progressbar.step()`을 호출하는 반복 타이머 이벤트를 예약합니다. 생략하면, *interval*의 기본값은 50밀리초입니다.

step(*amount=None*)

진행 막대의 값을 *amount*만큼 증가시킵니다.

생략하면 *amount*의 기본값은 1.0입니다.

stop()

자동 증가 모드를 중지합니다: 이 진행 막대에 대한 `Progressbar.start()`로 시작된 반복 타이머 이벤트를 취소합니다.

25.8.8 Separator

`ttk.Separator` 위젯은 가로나 세로 구분 막대를 표시합니다.

`ttk.Widget`에서 상속된 것 외에 다른 메서드는 없습니다.

옵션

이 위젯은 다음과 같은 특정 옵션을 받아들입니다:

옵션	설명
orient	"horizontal" (가로) 이나 "vertical" (세로) 중 하나입니다. 구분 막대의 방향을 지정합니다.

25.8.9 Sizegrip

`ttk.Sizegrip` 위젯 (확장 상자(grow box) 라고도 합니다) 을 사용하면 그립 (grip) 을 누르고 드래그하여 포함하는 최상위 창의 크기를 조정할 수 있습니다.

이 위젯에는 `ttk.Widget`에서 상속된 것 외에 특정 옵션이나 메서드가 없습니다.

플랫폼별 노트

- 맥 OS에서, 최상위 창에는 기본적으로 내장 크기 그립이 자동으로 포함됩니다. 내장 그립이 위젯을 가리기 때문에 `Sizegrip`을 추가하는 것은 무해합니다.

버그

- 포함하는 최상위 수준의 위치가 화면의 오른쪽이나 아래쪽을 기준으로 지정되면 (예를 들어 …), `Sizegrip` 위젯은 창의 크기를 조정하지 않습니다.
- 이 위젯은 "동남쪽" 크기 조정만 지원합니다.

25.8.10 Treeview

`ttk.Treeview` 위젯은 계층적 항목 컬렉션을 표시합니다. 각 항목에는 텍스트 레이블, 선택적 이미지 및 선택적 데이터값 목록이 있습니다. 데이터값은 트리 레이블 다음에 연속되는 열에 표시됩니다.

위젯 옵션 `displaycolumns`를 설정하여 데이터값이 표시되는 순서를 제어할 수 있습니다. 트리 위젯은 열 제목을 표시할 수도 있습니다. 위젯 옵션 열에 나열된 숫자나 기호 이름으로 열에 액세스 할 수 있습니다. 열 식별자를 참조하십시오.

각 항목은 고유한 이름으로 식별됩니다. 호출자가 제공하지 않으면 위젯은 항목 ID를 생성합니다. `{}`라고 이름 붙은, 구별되는 루트 항목이 있습니다. 루트 항목 자체는 표시되지 않습니다; 이것의 자식들이 계층 구조의 최상위 수준에 나타납니다.

각 항목에는 이벤트 바인딩을 개별 항목과 연관시키고 항목의 모양을 제어하는 데 사용할 수 있는 태그 목록이 있습니다.

Treeview 위젯은 스크롤 가능한 위젯 옵션에 설명된 옵션과 `Treeview.xview()`와 `Treeview.yview()` 메서드에 따라 가로와 세로 스크롤을 지원합니다.

옵션

이 위젯은 다음과 같은 특정 옵션을 받아들입니다:

옵션	설명
columns	열 수와 이름을 지정하는, 열 식별자 리스트.
displaycolumns	표시할 데이터 열과 표시되는 순서를 지정하는 열 식별자 (기호나 정수 인덱스) 리스트, 또는 문자열 "#all".
height	보여야 하는 행 수를 지정합니다. 참고: 요청된 너비는 열 너비의 합계로 결정됩니다.
padding	위젯의 내부 패딩을 지정합니다. 패딩은 최대 4개의 길이 명세 리스트입니다.
selectmode	내장 클래스 바인딩이 선택을 관리하는 방법을 제어합니다. "extended", "browse" 또는 "none" 중 하나입니다. "extended" (기본값)로 설정하면, 여러 항목을 선택할 수 있습니다. "browse" 이면, 한 번에 하나의 항목 만 선택됩니다. "none" 이면 선택이 변경되지 않습니다. 이 옵션의 값과 관계없이, 응용 프로그램 코드와 태그 바인딩은 원하는 대로 선택을 설정할 수 있음에 유의하십시오.
show	표시할 트리의 요소를 지정하는, 다음 값 중 0개 이상을 포함하는 리스트. • tree: 열 #0에 트리 레이블을 표시합니다. • headings: 제목 행을 표시합니다. 기본값은 "tree headings" 입니다, 즉, 모든 요소를 표시합니다. **참고**: show="tree" 가 지정되지 않은 경우에도, #0 열은 항상 트리 열을 나타냅니다.

항목 옵션

insert와 item 위젯 명령에서 항목에 대해 다음 항목 옵션을 지정할 수 있습니다.

옵션	설명
text	항목에 표시할 텍스트 레이블.
image	레이블 왼쪽에 표시되는, Tk 이미지.
values	항목과 관련된 값의 리스트. 각 항목은 위젯 옵션 열과 같은 수의 값을 가져야 합니다. 열보다 적은 값이 있으면, 나머지 값은 비어 있다고 가정합니다. 열보다 많은 값이 있으면, 추가 값은 무시됩니다.
open	항목의 자식을 표시할지를 나타내는 `True`/`False` 값.
tags	이 항목과 관련된 태그의 리스트.

태그 옵션

태그에 다음 옵션을 지정할 수 있습니다:

옵션	설명
foreground	텍스트 전경색을 지정합니다.
background	셀이나 항목 배경색을 지정합니다.
font	텍스트를 그릴 때 사용할 글꼴을 지정합니다.
image	항목의 image 옵션이 비어있는 경우, 항목 이미지를 지정합니다.

열 식별자

열 식별자는 다음 형식 중 하나를 취합니다:

- 열 옵션 목록에 있는 기호 이름.
- n번째 데이터 열을 지정하는, 정수 n.
- #n 형식의 문자열, 여기서 n은 정수이며 n번째 표시 열을 지정합니다.

노트:

- 항목의 옵션 값은 저장된 순서와 다른 순서로 표시될 수 있습니다.
- show=" tree" 가 지정되지 않은 경우에도, #0 열은 항상 트리 열을 나타냅니다.

데이터 열 번호는 항목의 옵션 값 리스트에 대한 인덱스입니다; 표시 열 번호는 값이 표시되는 트리의 열 번호입니다. 트리 레이블은 열 #0 에 표시됩니다. displaycolumns 옵션을 설정하지 않으면, 데이터 열 n이 열 #n+1 에 표시됩니다. 다시, **#0 열은 항상 트리 열을 나타냅니다.**

가상 이벤트

Treeview 위젯은 다음과 같은 가상 이벤트를 생성합니다.

이벤트	설명
«TreeviewSelect»	선택이 변경될 때마다 생성됩니다.
«TreeviewOpen»	포커스 항목을 open=True 로 설정하기 직전에 생성됩니다.
«TreeviewClose»	포커스 항목을 open=False 로 설정한 직후 생성됩니다.

`Treeview.focus()`와 `Treeview.selection()` 메서드를 사용하여 영향을 받는 항목이나 항목들을 결정할 수 있습니다.

ttk.Treeview

class tkinter.ttk.**Treeview**

bbox(*item*, *column=None*)

지정된 *item*의 경계 상자(트리 뷰 위젯의 창에 상대적인)를 (x, y, 너비, 높이) 형식으로 반환합니다.

*column*을 지정하면, 해당 셀의 경계 상자를 반환합니다. *item*이 보이지 않으면 (즉, 닫힌 항목의 자손이거나 화면 밖으로 스크롤 되면) 빈 문자열을 반환합니다.

get_children(*item=None*)

*item*에 속하는 자식의 리스트를 반환합니다.

*item*이 지정되지 않으면, 루트 자식을 반환합니다.

set_children(*item*, **newchildren*)

*item*의 자식을 *newchildren*으로 바꿉니다.

*item*에 있지만 *newchildren*에 없는 자식은 트리에서 분리됩니다. *newchildren*의 어떤 항목도 *item*의 조상이 될 수 없습니다. *newchildren*을 지정하지 않으면 *item*의 자식이 분리됨에 유의하십시오.

column(*column*, *option=None*, ***kw*)

지정된 *column*에 대한 옵션을 조회하거나 수정합니다.

*kw*가 제공되지 않으면, 열 옵션 값의 딕셔너리를 반환합니다. *option*이 지정되면 해당 *option*의 값이 반환됩니다. 그렇지 않으면, 옵션을 해당 값으로 설정합니다.

유효한 옵션/값은 다음과 같습니다:

id

열 이름을 반환합니다. 이것은 읽기 전용 옵션입니다.

***anchor*: 표준 Tk 앵커값 중 하나.**
이 열의 텍스트가 셀을 기준으로 정렬되는 방법을 지정합니다.

***minwidth*: 너비**
열의 픽셀 단위의 최소 너비. 위젯의 크기가 조정되거나 사용자가 열을 드래그할 때 트리뷰 위젯은 이 옵션으로 지정된 것보다 작게 열을 만들지 않습니다.

***stretch*: `True/False`**
위젯 크기를 조정할 때 열 너비를 조정할지를 지정합니다.

***width*: 너비**
열의 픽셀 단위 너비.

트리 열을 구성하려면, column = "#0" 으로 호출하십시오.

`delete`(**items*)

지정된 *items*와 그들의 모든 자손을 삭제합니다.

루트 항목은 삭제되지 않을 수 있습니다.

`detach`(**items*)

지정된 *items*를 모두 트리에서 연결 해제합니다.

항목과 그들의 모든 자손은 여전히 존재하며, 트리의 다른 지점에 다시 삽입될 수 있지만, 표시되지는 않습니다.

루트 아이템은 분리되지 않을 수 있습니다.

`exists`(*item*)

지정된 *item*이 트리에 있으면 `True`를 반환합니다.

`focus`(*item=None*)

*item*이 지정되면, 포커스 항목을 *item*으로 설정합니다. 그렇지 않으면, 현재 포커스 항목을 반환하거나, 없으면 '' 을 반환합니다.

`heading`(*column*, *option=None*, ***kw*)

지정된 *column*에 대한 제목(heading) 옵션을 조회하거나 수정합니다.

*kw*가 제공되지 않으면, 제목 옵션값의 딕셔너리를 반환합니다. *option*이 지정되면 해당 *option*의 값이 반환됩니다. 그렇지 않으면, 옵션을 해당 값으로 설정합니다.

유효한 옵션/값은 다음과 같습니다:

***text*: 텍스트**
열 제목에 표시할 텍스트.

***image*: 이미지 이름**
열 제목의 오른쪽에 표시할 이미지를 지정합니다.

***anchor*: 앵커**
제목 텍스트를 정렬하는 방법을 지정합니다. 표준 Tk 앵커값 중 하나입니다.

***command*: 콜백**
제목 레이블을 누를 때 호출되는 콜백.

트리 열 제목을 구성하려면, column = "#0" 으로 호출하십시오.

`identify`(*component*, *x*, *y*)

x 와 *y*로 주어진 점 밑에 있는 지정된 *component*에 대한 설명을 반환하거나, 해당 *component*가 해당 위치에 없으면 빈 문자열을 반환합니다.

`identify_row`(*y*)

y 위치에 있는 항목의 항목 ID를 반환합니다.

`identify_column`(*x*)

위치 *x*에 있는 셀의 데이터 열 식별자를 반환합니다.

트리 열의 ID는 #0 입니다.

identify_region(*x*, *y*)

다음 중 하나를 반환합니다:

영역(region)	의미
heading	트리 제목 영역.
separator	두 열 제목 사이의 공간.
tree	트리 영역.
cell	데이터 셀.

가용성: Tk 8.6.

identify_element(*x*, *y*)

위치 *x*, *y*에 있는 요소를 반환합니다.

가용성: Tk 8.6.

index(*item*)

부모의 자식 리스트에서 *item*의 정수 인덱스를 반환합니다.

insert(*parent*, *index*, *iid=None*, ***kw*)

새 항목을 만들고 새로 만들어진 항목의 항목 ID를 반환합니다.

*parent*는 부모 항목의 항목 ID이거나, 새 최상위 항목을 만들려면 빈 문자열입니다. *index*는 정수이거나, "end" 값으로, 부모의 자식 리스트에서 새 항목을 삽입할 위치를 지정합니다. *index*가 0보다 작거나 같으면, 새 노드가 처음에 삽입됩니다; *index*가 현재 자식 수보다 크거나 같으면, 끝에 삽입됩니다. *iid*가 지정되면, 항목 식별자로 사용됩니다; *iid*는 아직 트리에 존재하지 않아야 합니다. 그렇지 않으면, 새로운 고유 식별자가 생성됩니다.

사용 가능한 옵션 목록은 *Item Options*을 참조하십시오.

item(*item*, *option=None*, ***kw*)

지정된 *item*에 대한 옵션을 조회하거나 수정합니다.

옵션이 제공되지 않으면 항목에 대한 옵션/값이 포함된 딕셔너리가 반환됩니다. *option*이 지정되면 해당 옵션의 값이 반환됩니다. 그렇지 않으면, 옵션을 *kw*에서 제공한 해당 값으로 설정합니다.

move(*item*, *parent*, *index*)

*item*을 *parent*의 자식 리스트에서 *index* 위치로 이동합니다.

항목을 그 자신의 자손 항목 중 하나 밑으로 이동하는 것은 불법입니다. *index*가 0보다 작거나 같으면, *item*이 처음으로 이동합니다; 자식 수보다 크거나 같으면, 끝으로 이동합니다. *item*이 분리되었으면 다시 연결됩니다.

next(*item*)

*item*의 다음 형제의 식별자를 반환하거나, *item*이 부모의 마지막 자식이면 '' 을 반환합니다.

parent(*item*)

*item*의 부모 ID를 반환하거나, *item*이 계층의 최상위에 있으면 '' 을 반환합니다.

prev(*item*)

*item*의 이전 형제의 식별자를 반환하거나, *item*이 부모의 첫 번째 자식이면 '' 을 반환합니다.

reattach(*item*, *parent*, *index*)

`Treeview.move()`의 별칭.

see(*item*)

*item*이 보이도록 합니다.

*item*의 모든 조상의 open 옵션을 `True`로 설정하고, 필요하면 위젯을 스크롤 하여 *item*이 트리의 보이는 부분 내에 있도록 합니다.

selection()

선택한 항목의 튜플을 반환합니다.

버전 3.8에서 변경: `selection()`은 더는 인자를 취하지 않습니다. 선택 상태를 변경하려면 다음 선택 메서드를 사용하십시오.

selection_set(**items*)

*items*가 새로운 선택이 됩니다.

버전 3.6에서 변경: *items*는 단일 튜플이 아닌 별도의 인자로 전달될 수 있습니다.

selection_add(**items*)

선택에 *items*를 추가합니다.

버전 3.6에서 변경: *items*는 단일 튜플이 아닌 별도의 인자로 전달될 수 있습니다.

selection_remove(**items*)

선택에서 *items*를 제거합니다.

버전 3.6에서 변경: *items*는 단일 튜플이 아닌 별도의 인자로 전달될 수 있습니다.

selection_toggle(**items*)

*items*에 있는 각 항목의 선택 상태를 토글 합니다.

버전 3.6에서 변경: *items*는 단일 튜플이 아닌 별도의 인자로 전달될 수 있습니다.

set(*item*, *column=None*, *value=None*)

하나의 인자로는, 지정된 *item*에 대한 열/값 쌍 딕셔너리를 반환합니다. 두 개의 인자를 사용하면, 지정된 *column*의 현재 값을 반환합니다. 세 개의 인자를 사용하면, 지정된 *item*의 지정된 *column*의 값을, 지정된 *value*로 설정합니다.

tag_bind(*tagname*, *sequence=None*, *callback=None*)

주어진 이벤트 *sequence*에 대한 콜백을 태그 *tagname*에 바인딩합니다. 이벤트가 항목에 전달되면, 각 항목의 태그 옵션에 대한 콜백이 호출됩니다.

tag_configure(*tagname*, *option=None*, ***kw*)

지정된 *tagname*에 대한 옵션을 조회하거나 수정합니다.

*kw*가 제공되지 않으면, *tagname*에 대한 옵션 설정의 딕셔너리를 반환합니다. *option*이 지정되면, 지정된 *tagname*에 대한 해당 *option*의 값을 반환합니다. 그렇지 않으면, 옵션을 주어진 *tagname*에 대해 해당하는 값으로 설정합니다.

tag_has(*tagname*, *item=None*)

*item*이 지정되면, 지정된 *item*에 지정된 *tagname*이 있는지에 따라 1이나 0을 반환합니다. 그렇지 않으면, 지정된 태그가 있는 모든 항목의 리스트를 반환합니다.

가용성: Tk 8.6

xview(**args*)

트리 뷰의 가로 위치를 조회하거나 수정합니다.

yview(**args*)

트리 뷰의 수직 위치를 조회하거나 수정합니다.

25.8.11 Ttk 스타일링

`ttk`의 각 위젯에는 스타일이 지정되는데, 이 스타일은 요소 옵션의 동적 및 기본 설정과 함께 위젯을 구성하는 요소 집합과 배열 방식을 지정합니다. 기본적으로 스타일 이름은 위젯의 클래스 이름과 같지만, 위젯의 스타일 옵션으로 재정의될 수 있습니다. 위젯의 클래스 이름을 모르면, `Misc.winfo_class()` 메서드 (somewidget.winfo_class())를 사용하십시오.

더 보기

> **Tcl' 2004 conference presentation**
> 이 문서는 테마 엔진의 작동 방식을 설명합니다

class tkinter.ttk.**Style**

이 클래스는 스타일 데이터베이스를 조작하는 데 사용됩니다.

configure(*style*, *query_opt=None*, ***kw*)

*style*에서 지정된 옵션의 기본값을 조회하거나 설정합니다.

*kw*의 각 키는 옵션이며 각 값은 해당 옵션의 값을 식별하는 문자열입니다.

예를 들어, 모든 기본 버튼을 패딩이 있고 배경색이 다른 평평한 버튼으로 바꾸려면:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat",
   background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map(*style*, *query_opt=None*, ***kw*)

*style*에서 지정된 옵션의 동적 값을 조회하거나 설정합니다.

*kw*의 각 키는 옵션이며 각 값은 (일반적으로) 튜플, 리스트 또는 다른 선호하는 것에 그룹화된 상태 명세(statespecs)를 포함하는 리스트나 튜플이어야 합니다. 상태 명세(statespec)는 하나 이상의 상태와 그다음에 값이 오는 복합체입니다.

예를 들면 더 이해하기 쉽습니다:

```
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
    )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

옵션의 (상태들, 값) 시퀀스의 순서는 중요함에 유의하십시오. 예를 들어, foreground 옵션에서 순서가 `[('active', 'blue'), ('pressed', 'red')]`로 변경되면, 예를 들어, 위젯이 active 나 pressed 상태일 때 결과는 파란색 전경이 됩니다.

lookup(*style*, *option*, *state=None*, *default=None*)

*style*에서 *option*에 지정된 값을 반환합니다.

*state*가 지정되면, 하나 이상의 상태 시퀀스일 것으로 기대됩니다. *default* 인자가 설정되면, 옵션에 대한 명세가 없을 때 폴 백값으로 사용됩니다.

Button이 기본적으로 사용하는 글꼴을 확인하려면:

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout(*style*, *layoutspec=None*)

주어진 *style*에 대한 위젯 레이아웃을 정의합니다. *layoutspec*이 생략되면, 지정된 스타일의 레이아웃 명세를 반환합니다.

지정되면, *layoutspec*은 리스트나 다른 시퀀스 형(문자열 제외)이어야 합니다. 여기서 각 항목은 튜플이어야 하고 첫 번째 항목은 레이아웃 이름이고 두 번째 항목은 레이아웃에 설명된 형식이어야 합니다.

형식을 이해하려면, 다음 예제를 참조하십시오 (유용한 것은 아닙니다):

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
    ("Menubutton.background", None),
    ("Menubutton.button", {"children":
        [("Menubutton.focus", {"children":
            [("Menubutton.padding", {"children":
                [("Menubutton.label", {"side": "left", "expand": 1})]
            })]
        })]
    }),
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element_create(*elementname*, *etype*, **args*, ***kw*)

현재 테마에서 주어진 *etype*의 새 요소를 만듭니다. *etype*은 "image", "from" 또는 "vsapi" 일 것으로 기대됩니다. 마지막 것은 윈도우 용 Tk 8.6에서만 사용할 수 있습니다.

"image" 를 사용하면, *args*는 기본 이미지 이름과 그 뒤에 오는 statespec/value 쌍 (이것이 이미지 명세 (imagespec) 입니다)을 포함해야 하며, *kw*에는 다음 옵션이 올 수 있습니다:

border=padding
패딩은 각각 좌(left), 상(top), 우(right), 하(bottom) 테두리를 지정하는 최대 4개의 정수 리스트입니다.

height=height
요소의 최소 높이를 지정합니다. 0보다 작으면, 기본 이미지의 높이가 기본값으로 사용됩니다.

padding=padding
요소의 내부 패딩을 지정합니다. 지정하지 않으면 기본값은 border의 값입니다.

sticky=spec
최종 파슬(parcel) 내에 이미지를 배치하는 방법을 지정합니다. 명세에는 0개 이상의 문자 "n", "s", "w" 또는 "e" 가 포함됩니다.

width=width
요소의 최소 너비를 지정합니다. 0보다 작으면, 기본 이미지의 너비가 기본값으로 사용됩니다.

예제:

```
img1 = tkinter.PhotoImage(master=root, file='button.png')
img1 = tkinter.PhotoImage(master=root, file='button-pressed.png')
img1 = tkinter.PhotoImage(master=root, file='button-active.png')
style = ttk.Style(root)
style.element_create('Button.button', 'image',
                     img1, ('pressed', img2), ('active', img3),
                     border=(2, 4), sticky='we')
```

"from" 이 *etype*의 값으로 사용되면, `element_create()`는 기존 요소를 복제합니다. *args*는 요소를 복제할 테마 이름(themename)과, 선택적으로 요소를 복제할 요소를 포함할 것으로 기대됩니다. 복제할 요소를 지정하지 않으면, 빈 요소가 사용됩니다. *kw*는 폐기됩니다.

예제:

```
style = ttk.Style(root)
style.element_create('plain.background', 'from', 'default')
```

If "vsapi" is used as the value of *etype*, `element_create()` will create a new element in the current theme whose visual appearance is drawn using the Microsoft Visual Styles API which is responsible for the themed styles on Windows XP and Vista. *args* is expected to contain the Visual Styles class and part as given in the Microsoft documentation followed by an optional sequence of tuples of ttk states and the corresponding Visual Styles API state value. *kw* may have the following options:

padding=padding
Specify the element' s interior padding. *padding* is a list of up to four integers specifying the left, top, right and bottom padding quantities respectively. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left. In other words, a list of three numbers specify the left, vertical, and right padding; a list of two numbers specify the horizontal and the vertical padding; a single number specifies the same padding all the way around the widget. This option may not be mixed with any other options.

margins=padding
요소 외부 패딩을 지정합니다. *padding*은 각각 좌(left), 상(top), 우(right), 하(bottom) 패딩량을 지정하는 최대 4개의 정수 리스트입니다. 이 옵션은 다른 옵션과 함께 사용할 수 없습니다.

width=width
Specifies the width for the element. If this option is set then the Visual Styles API will not be queried for the recommended size or the part. If this option is set then *height* should also be set. The *width* and *height* options cannot be mixed with the *padding* or *margins* options.

height=height
Specifies the height of the element. See the comments for *width*.

예제:

```
style = ttk.Style(root)
style.element_create('pin', 'vsapi', 'EXPLORERBAR', 3, [
                     ('pressed', '!selected', 3),
                     ('active', '!selected', 2),
                     ('pressed', 'selected', 6),
                     ('active', 'selected', 5),
                     ('selected', 4),
                     ('', 1)])
style.layout('Explorer.Pin',
             [('Explorer.Pin.pin', {'sticky': 'news'})])
pin = ttk.Checkbutton(style='Explorer.Pin')
pin.pack(expand=True, fill='both')
```

버전 3.13에서 변경: Added support of the "vsapi" element factory.

element_names()

현재 테마에 정의된 요소 목록을 반환합니다.

element_options(*elementname*)

*elementname*의 옵션 리스트를 반환합니다.

theme_create(*themename*, *parent=None*, *settings=None*)

새로운 테마를 만듭니다.

*themename*이 이미 존재하면 에러입니다. *parent*가 지정되면, 새 테마는 부모 테마에서 스타일, 요소 및 레이아웃을 상속합니다. *settings*가 있으면, `theme_settings()`에 사용되는 것과 같은 문법일 것으로 기대됩니다.

theme_settings(*themename*, *settings*)

현재 테마를 *themename*으로 임시 설정하고, 지정된 *settings*를 적용한 다음 이전 테마를 복원합니다.

*settings*의 각 키는 스타일이며 각 값에는 'configure', 'map', 'layout' 및 'element create' 키가 포함될 수 있으며 각각 `Style.configure()`, `Style.map()`, `Style.layout()` 및 `Style.element_create()`가 지정한 것과 같은 형식을 갖습니다.

예를 들어, 기본 테마의 Combobox를 약간 변경해 봅시다:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
   "TCombobox": {
       "configure": {"padding": 5},
       "map": {
           "background": [("active", "green2"),
                          ("!disabled", "green4")],
           "fieldbackground": [("!disabled", "green3")],
           "foreground": [("focus", "OliveDrab1"),
                          ("!disabled", "OliveDrab2")]
       }
   }
})

combo = ttk.Combobox().pack()

root.mainloop()
```

theme_names()

알려진 모든 테마의 리스트를 반환합니다.

theme_use(*themename=None*)

*themename*이 제공되지 않으면, 사용 중인 테마를 반환합니다. 그렇지 않으면, 현재 테마를 *themename*으로 설정하고, 모든 위젯을 새로 고치고 «ThemeChanged» 이벤트를 방출합니다.

레이아웃

레이아웃은 옵션을 취하지 않으면 `None`일 수 있습니다, 또는 요소를 정렬하는 방법을 지정하는 옵션의 딕셔너리입니다. 레이아웃 메커니즘은 pack 지오메트리 관리자의 단순화 된 버전을 사용합니다: 초기 캐비티(cavity)가 주어지면, 각 요소에는 파슬(parcel)이 할당됩니다.

유효한 옵션/값은 다음과 같습니다:

***side*: whichside**
요소를 배치할 캐비티(cavity)의 변을 지정합니다; 상(top), 우(right), 하(bottom) 또는 좌(left) 중 하나입니다. 생략하면, 요소가 전체 캐비티를 차지합니다.

***sticky*: nswe**
할당된 파슬(parcel)에서 요소가 배치되는 위치를 지정합니다.

***unit*: 0 또는 1**
1로 설정하면, `Widget.identify()` 등의 목적으로 요소와 그것의 모든 자손이 단일 요소로 처리됩니다. 그립이 있는 스크롤 막대 썸(thumbs)과 같은 것들에 사용됩니다.

***children*: [sublayout…]**
요소 안에 배치할 요소 리스트를 지정합니다. 각 요소는 첫 번째 항목이 레이아웃 이름이고, 다른 하나는 레이아웃인 튜플(또는 다른 시퀀스 형)입니다.

25.9 IDLE — 파이썬 편집기와 셸

소스 코드: Lib/idlelib/

IDLE은 파이썬의 통합 개발 및 학습 환경입니다.

IDLE에는 다음과 같은 기능이 있습니다:

- 크로스 플랫폼: 윈도우, 유닉스 및 macOS에서 거의 같게 작동합니다
- 코드 입력, 출력 및 에러 메시지를 채색하는 파이썬 셸 창 (대화식 인터프리터)
- 다중 실행 취소, 파이썬 색상 지정, 스마트 들여쓰기, 호출 팁, 자동 완성 및 기타 기능이 있는 다중 창 텍스트 편집기
- 모든 창 내에서 검색, 편집기 창 내에서 교체 및 여러 파일을 통한 검색 (grep)
- 지속적인 중단점(breakpoints), 스테핑(stepping) 및 전역과 지역 이름 공간 보기가 있는 디버거
- 구성, 브라우저 및 기타 대화 상자

25.9.1 메뉴

IDLE에는 두 가지 메인 창 유형이 있습니다, 셸 창과 편집기 창. 여러 개의 편집기 창을 동시에 가질 수 있습니다. 윈도우와 리눅스에서는, 각각 자신의 최상위 메뉴가 있습니다. 아래에 설명된 각 메뉴는 어떤 창 유형과 관련되는지를 나타냅니다.

가령 Edit => Find in Files 에서 사용되는 것과 같은 출력 창은 편집기 창의 하위 유형입니다. 현재 같은 최상위 메뉴가 있지만, 기본 제목과 문맥 메뉴가 다릅니다.

macOS에는, 하나의 응용 프로그램 메뉴가 있습니다. 현재 선택된 창에 따라 동적으로 변경됩니다. IDLE 메뉴가 있으며, 아래 설명된 일부 항목은 Apple 지침에 따라 이동됩니다.

File 메뉴 (셸과 편집기)

New File
새 파일 편집 창을 만듭니다.

Open…
열기 대화 상자로 기존 파일을 엽니다.

Open Module…
기존 모듈을 엽니다 (sys.path를 검색합니다).

Recent Files
최근 파일의 목록을 엽니다. 하나를 클릭하여 엽니다.

Module Browser
현재 편집기 파일의 함수, 클래스 및 메서드를 트리 구조로 표시합니다. 셸에서, 모듈을 먼저 여십시오.

Path Browser
sys.path 디렉터리, 모듈, 함수, 클래스 및 메서드를 트리 구조로 표시합니다.

Save
현재 창을 (있다면) 연관된 파일에 저장합니다. 열거나 마지막으로 저장한 이후에 변경된 창에는 창 제목 앞뒤에 * 가 있습니다. 연결된 파일이 없으면, 대신 Save As 를 수행합니다.

Save As…
다른 이름으로 저장(Save As) 대화 상자를 사용하여 현재 창을 저장합니다. 저장된 파일이 창의 새 연관된 파일이 됩니다. (파일 관리자가 확장자를 숨기도록 설정되어 있으면, 파일명 상자에서 현재 확장자가 생략됩니다. 새 파일명에 '.' 이 없으면, 파이썬과 텍스트 파일에 '.py' 와 '.txt' 가 추가됩니다. 맥 OS Aqua 에서는 예외인데, 모든 파일에 대해 '.py' 가 추가됩니다.)

Save Copy As…
연관된 파일을 변경하지 않고 현재 창을 다른 파일에 저장합니다. (파일명 확장자에 대해서는 위의 Save As 참고 사항을 참조하십시오.)

Print Window
현재 창을 기본 프린터로 인쇄합니다.

Close Window
현재 창을 닫습니다 (저장하지 않은 편집기면, 저장하도록 요청합니다; 저장하지 않은 셸이면, 실행을 중단하도록 요청합니다. 셸 창에서 `exit()` 나 `close()` 를 호출하면 셸을 닫습니다. 만약 이 것이 유일한 창이면, IDLE 도 종료합니다.).

Exit IDLE
모든 창을 닫고 IDLE을 종료합니다 (저장되지 않은 편집 창을 저장하도록 요청합니다).

Edit 메뉴 (셸과 편집기)

Undo
현재 창에 대한 마지막 변경을 되돌립니다. 최대 1000개의 변경을 되돌릴 수 있습니다.

Redo
마지막으로 되돌린 변경을 현재 창에 다시 실행합니다.

Select All
현재 창의 전체 내용을 선택합니다.

Cut
선택 사항을 시스템 전체 클립 보드에 복사합니다; 그런 다음 선택을 삭제합니다.

Copy
선택 사항을 시스템 전체 클립 보드에 복사합니다.

Paste
시스템 전체 클립 보드의 내용을 현재 창에 삽입합니다.

클립 보드 기능은 문맥 메뉴에서도 사용할 수 있습니다.

Find…
많은 옵션이 제공되는 검색 대화 상자를 엽니다

Find Again
마지막 검색이 있으면 반복합니다.

Find Selection
현재 선택된 문자열이 있으면 검색합니다.

Find in Files…
파일 검색 대화 상자를 엽니다. 새로운 출력 창에 결과를 넣습니다.

Replace…
검색과 치환 대화 상자를 엽니다.

Go to Line
요청한 줄의 시작 부분으로 커서를 이동하고 해당 줄을 표시합니다. 파일 끝을 지나는 요청은 파일의 끝으로 갑니다. 모든 선택을 취소하고 줄과 열 상태를 갱신합니다.

Show Completions
기존 이름을 선택할 수 있는 스크롤 할 수 있는 목록을 엽니다. 아래의 편집 및 탐색 섹션에서 완성을 참조하십시오.

Expand Word
같은 창에서 전체 단어와 일치하도록 입력한 접두사를 확장합니다; 다른 확장을 얻으려면 반복하십시오.

Show Call Tip
함수에 대해 닫히지 않은 괄호 뒤에서, 함수 매개 변수 힌트가 있는 작은 창을 엽니다. 아래의 편집과 탐색 섹션에서 콜팁을 참조하십시오.

Show Surrounding Parens
주변 괄호를 강조 표시합니다.

Format 메뉴 (편집기 창 전용)

Format Paragraph
주석 블록이나 여러 줄 문자열의 빈 줄로 구분되는 현재 단락이나 문자열에서 선택한 줄을 다시 포맷합니다. 단락의 모든 줄은 N 열 미만으로 포맷되며, 여기서 N은 기본적으로 72입니다.

Indent Region
선택한 줄을 들여쓰기 너비(기본값은 4개의 스페이스)만큼 오른쪽으로 이동합니다.

Dedent Region
선택한 줄을 들여쓰기 너비(기본값은 4개의 스페이스)만큼 왼쪽으로 이동합니다.

Comment Out Region
선택한 줄 앞에 ## 을 삽입합니다.

Uncomment Region
선택한 줄에서 선행 # 이나 ## 을 제거합니다.

Tabify Region
연속된 선행 스페이스를 탭으로 바꿉니다. (참고: 4개의 스페이스 블록을 사용하여 파이썬 코드를 들여 쓰는 것이 좋습니다.)

Untabify Region
모든 탭을 올바른 수의 스페이스로 바꿉니다.

Toggle Tabs
스페이스와 탭 들여쓰기 간을 전환하는 대화 상자를 엽니다.

New Indent Width
들여쓰기 너비를 변경하는 대화 상자를 엽니다. 파이썬 커뮤니티에서 받아들여지는 기본값은 4개의 스페이스입니다.

Strip Trailing Chitespace
여러 줄 문자열 내의 줄을 포함하여, 각 줄에 str.rstrip을 적용하여 줄의 마지막 비 공백 문자 뒤의 후행 스페이스와 기타 공백 문자를 제거합니다. 셸 창을 제외하고, 파일 끝에서 추가 줄 바꿈을 제거합니다.

Run 메뉴 (편집기 창 전용)

Run Module
*Check Module*을 수행합니다. 에러가 없으면, 셸을 다시 시작하여 환경을 정리한 다음, 모듈을 실행합니다. 출력은 셸 창에 표시됩니다. 출력에는 `print`나 `write`를 사용해야 함에 유의하십시오. 실행이 완료되면, 셸은 포커스를 유지하고 프롬프트를 표시합니다. 이 시점에서, 실행 결과를 대화식으로 탐색할 수 있습니다. 이것은 명령 줄에서 `python -i file`로 파일을 실행하는 것과 유사합니다.

Run… Customized
*Run Module*과 같지만, 사용자 정의 설정으로 모듈을 실행합니다. 명령 줄 인자(*Command Line Arguments*)는 명령 줄에 전달된 것처럼 `sys.argv`를 확장합니다. 다시 시작하지 않고 모듈을 셸에서 실행할 수 있습니다.

Check Module
편집기 창에 현재 열려 있는 모듈의 문법을 검사합니다. 모듈이 저장되지 않았으면 IDLE은 설정 대화 상자의 General 탭에서 선택한 대로 사용자에게 저장을 요구하거나 자동 저장합니다. 문법 에러가 있으면, 대략적인 위치가 편집기 창에 표시됩니다.

Python Shell
파이썬 셸 창을 열거나 깨웁니다.

Shell 메뉴 (셸 창 전용)

View Last Restart
셸 창을 마지막 셸 재시작으로 스크롤 합니다.

Restart Shell
셸을 다시 시작하여 환경을 정리하고 디스플레이와 예외 처리를 재설정합니다.

Previous History
히스토리에서 현재 항목과 일치하는 이전 명령을 순환합니다.

Next History
히스토리에서 현재 항목과 일치하는 다음 명령을 순환합니다.

Interrupt Execution
실행 중인 프로그램을 중지합니다.

Debug 메뉴 (셸 창 전용)

Go to File/Line
커서가 놓인 현재 줄과 줄 위에서 파일명과 줄 번호는 찾습니다. 발견되면, (파일이 아직 열려 있지 않으면 파일을 열고) 줄을 보여줍니다. 이를 사용하여 예외 트레이스백에서 참조된 소스 줄과 Find in Files에서 찾은 줄을 볼 수 있습니다. 셸 창과 출력 창의 문맥 메뉴에서도 사용할 수 있습니다.

Debugger (toggle)
활성화되면, 셸에 입력되거나 편집기에서 실행된 코드가 디버거에서 실행됩니다. 편집기에서, 문맥 메뉴를 사용하여 중단점을 설정할 수 있습니다. 이 기능은 아직 불완전하고 다소 실험적입니다.

Stack Viewer
locals와 globals에 대한 액세스와 함께, 트리 위젯에 마지막 예외의 스택 트레이스백을 표시합니다.

Auto-open Stack Viewer
처리되지 않은 예외에서 스택 뷰어를 자동으로 여는 것을 전환합니다.

Options 메뉴 (셸과 편집기)

Configure IDLE
구성 대화 상자를 열고 다음과 같은 것들에 대한 설정을 변경합니다: 글꼴, 들여쓰기, 키 바인딩, 텍스트 색상 테마, 시작 창과 크기, 추가 도움말 소스 및 확장. macOS의 경우, 응용 프로그램 메뉴에서 Preferences를 선택하여 구성 대화 상자를 여십시오. 자세한 내용은, 도움말과 환경 설정에서 환경 설정을 참조하십시오.

대부분의 구성 옵션은 모든 창이나 모든 미래 창에 적용됩니다. 아래의 옵션 항목은 활성 창에만 적용됩니다.

Show/Hide Code Context (편집기 창 전용)
편집 창의 상단에 창의 상단 위로 스크롤 된 코드의 블록 컨텍스트를 표시하는 팬을 엽니다. 아래의 편집과 탐색 섹션에서 코드 컨텍스트를 참조하십시오.

Show/Hide Line Numbers (편집기 창 전용)
편집 창 왼쪽에 텍스트의 줄 번호를 표시하는 열을 엽니다. 기본 설정은 꺼져 있으며, 환경 설정에서 변경될 수 있습니다 (환경 설정을 참조하십시오).

Zoom/Restore Height
창을 보통 크기와 최대 높이 사이에서 전환합니다. IDLE 구성 대화 상자의 General 탭에서 변경하지 않는 한 초기 크기의 기본값은 40줄 80문자입니다. 화면의 최대 높이는 화면에서 처음 확대할 때 창을 일시적으로 최대화하여 결정됩니다. 화면 설정을 변경하면 저장된 높이가 무효가 될 수 있습니다. 이 전환은 창이 최대화되었을 때 적용되지 않습니다.

Window 메뉴 (셸과 편집기)

열려있는 모든 창의 이름을 나열합니다; 하나를 선택하여 전경으로 가져옵니다 (필요한 경우 아이콘화를 해제합니다).

Help 메뉴 (셸과 편집기)

About IDLE
버전, 저작권, 라이선스, 크레딧 등을 표시합니다.

IDLE Help
메뉴 옵션, 기본 편집과 탐색 및 기타 팁을 자세히 설명하는 이 IDLE 설명서를 표시합니다.

Python Docs
설치되었으면, 로컬 파이썬 문서에 액세스하거나, 웹 브라우저를 시작하여 최신 파이썬 설명서를 보여주는 docs.python.org 를 엽니다.

Turtle Demo
예제 파이썬 코드와 터틀 그래픽을 제공하는 turtledemo 모듈을 실행합니다.

IDLE 구성 대화 상자의 General 탭으로 여기에 도움말 소스를 추가할 수 있습니다. Help 메뉴 선택에 대한 자세한 내용은 아래의 도움말 소스 하위 섹션을 참조하십시오.

Context 메뉴

윈도우에서 마우스 오른쪽 버튼을 클릭하여 문맥 메뉴를 엽니다 (macOS에서는 Control-클릭). 문맥 메뉴에는 Edit 메뉴에도 있는 표준 클립 보드 기능이 있습니다.

Cut
선택 사항을 시스템 전체 클립 보드에 복사합니다; 그런 다음 선택을 삭제합니다.

Copy
선택 사항을 시스템 전체 클립 보드에 복사합니다.

Paste
시스템 전체 클립 보드의 내용을 현재 창에 삽입합니다.

편집기 창에는 중단점 기능도 있습니다. 중단점이 설정된 줄은 특별히 표시됩니다. 중단점은 디버거에서 실행할 때만 영향을 미칩니다. 파일의 중단점은 사용자의 `.idlerc` 디렉터리에 저장됩니다.

Set Breakpoint
현재 줄에 중단점을 설정합니다.

Clear Breakpoint
해당 줄에서 중단점을 지웁니다.

셸과 출력 창에서는 다음과 같은 것도 있습니다.

Go to file/line
Debug 메뉴와 같습니다.

셸 창에는 아래 파이썬 셸 창 하위 섹션에 설명된 출력 압착 기능도 있습니다.

Squeeze
커서가 출력 줄 위에 있으면, 위 코드와 아래 프롬프트 사이의 모든 출력을 'Squeezed text' 레이블로 압착합니다.

25.9.2 편집과 탐색

편집기 창

설정과 IDLE 시작 방법에 따라, IDLE이 시작될 때 편집기 창을 열 수 있습니다. 그런 다음, File 메뉴를 사용하십시오. 주어진 파일에 대해 열린 편집기 창이 하나만 있을 수 있습니다.

제목 표시 줄에는 파일 이름, 전체 경로 및 창을 실행하는 파이썬과 IDLE 버전이 포함됩니다. 상태 표시 줄에는 줄 번호('Ln')와 열 번호('Col')가 있습니다. 줄 번호는 1로 시작합니다; 열 번호는 0으로 시작합니다.

IDLE은 알려진 .py* 확장자를 가진 파일에 파이썬 코드가 포함되어 있고 다른 파일은 그렇지 않다고 가정합니다. Run 메뉴를 사용하여 파이썬 코드를 실행하십시오.

키 바인딩

The IDLE insertion cursor is a thin vertical bar between character positions. When characters are entered, the insertion cursor and everything to its right moves right one character and the new character is entered in the new space.

Several non-character keys move the cursor and possibly delete characters. Deletion does not puts text on the clipboard, but IDLE has an undo list. Wherever this doc discusses keys, 'C' refers to the `Control` key on Windows and Unix and the `Command` key on macOS. (And all such discussions assume that the keys have not been re-bound to something else.)

- Arrow keys move the cursor one character or line.
- `C-LeftArrow`와 `C-RightArrow`는 한 단어 좌/우로 이동합니다.
- `Home`과 `End`는 줄의 시작/끝으로 이동합니다.
- `Page Up`과 `Page Down`은 한 화면 위/아래로 이동합니다.
- `C-Home`과 `C-End`는 파일의 시작/끝으로 이동합니다
- `Backspace`와 `Del`(또는 `C-d`)은 이전/다음 문자를 삭제합니다.
- `C-Backspace`와 `C-Del`은 좌/우 의 한 단어를 삭제합니다.
- `C-k` deletes ('kills') everything to the right.

(복사하는 `C-c`와 붙여넣기 하는 `C-v`와 같은) 표준 키 바인딩이 작동 할 수 있습니다. 키 바인딩은 IDLE 구성 대화 상자에서 선택됩니다.

자동 들여쓰기

블록을 여는 문장 다음에, 다음 줄은 4개의 스페이스로 들여쓰기 됩니다 (파이썬 셸 창에서는 한 탭씩). 특정 키워드(break, return 등)가 다음에 다음 줄이 내어 쓰기 됩니다. 앞선 들여쓰기에서, `Backspace`는 최대 4개의 스페이스가 있으면 삭제합니다. `Tab`은 스페이스를 삽입합니다 (파이썬 셸 창에서는 하나의 탭), 개수는 들여쓰기 너비에 따라 다릅니다. 현재 Tcl/Tk 제한으로 인해 탭은 4개의 스페이스로 제한됩니다.

Format 메뉴의 Indent/Dedent Region 명령도 참조하십시오.

검색과 치환

Any selection becomes a search target. However, only selections within a line work because searches are only performed within lines with the terminal newline removed. If `[x] Regular expression` is checked, the target is interpreted according to the Python re module.

완성

요청되고 사용할 수 있을 때, 모듈 이름, 클래스나 함수의 어트리뷰트 또는 파일명에 대한 완성(completions)이 제공됩니다. 각 요청 방법은 기존 이름을 가진 완성 상자가 표시됩니다. (예외는 아래의 탭 완성을 참조하십시오.) 모든 상자는, 문자를 입력하고 삭제해서; `Up`, `Down`, `PageUp`, `PageDown`, `Home` 및 `End` 키를 쳐서; 상자 안에서 한 번의 클릭으로 완성 중인 이름과 상자에서 강조 표시된 항목을 변경합니다. `Escape`,

Enter 및 이중 Tab 키나 상자 외부를 클릭하여 상자를 닫으십시오. 상자 안에서 더블 클릭하면 선택하고 닫습니다.

상자를 여는 한 가지 방법은 키 문자를 입력하고 미리 정의된 간격만큼 기다리는 것입니다. 기본값은 2 초입니다; 설정 대화 상자에서 사용자 정의하십시오. (자동 팝업을 방지하려면, 지연을 100000000과 같은 큰 밀리초로 설정하십시오.) 임포트한 모듈 이름이나 클래스나 함수 어트리뷰트의 경우, '.' 를 입력하십시오. 루트 디렉터리의 파일명은 여는 인용 부호 바로 뒤에 *os.sep*이나 *os.altsep*을 입력하십시오. (윈도우에서는, 먼저 드라이브를 지정할 수 있습니다.) 디렉터리 이름과 구분자를 입력하여 서브 디렉터리로 이동하십시오.

기다리는 대신, 또는 상자를 닫은 후, Edit 메뉴에서 Show Completions를 사용하여 즉시 완성 상자를 여십시오. 기본 단축키는 C-space 입니다. 상자를 열기 전에 원하는 이름의 접두사를 입력하면, 첫 번째 일치나 근거리 불일치가 표시됩니다. 결과는 상자가 표시된 후 접두사를 입력하는 것과 같습니다. 따옴표 뒤에서의 Show Completions는 루트 디렉터리 대신 현재 디렉터리에서 파일명을 완성합니다.

접두사 다음에 Tab을 누르면 일반적으로 Show Completions와 같은 효과가 있습니다. (접두사가 없으면 들여쓰기합니다.) 그러나, 접두사와 일치하는 항목이 하나뿐이면, 상자를 열지 않고 해당 일치가 편집기 텍스트에 즉시 추가됩니다.

문자열 밖에서 선행 '.' 없이 접두어 뒤에서 'Show Completions' 를 실행하거나 Tab을 누르면 키워드, 내장 이름 및 사용 가능한 모듈 수준 이름이 포함된 상자를 엽니다.

(셸과 달리) 편집기에서 코드를 편집할 때는, 셸을 다시 시작하지 않고 코드를 실행해서 사용 가능한 모듈 수준 이름을 늘리십시오. 파일 맨 위에 임포트를 추가한 후에 특히 유용합니다. 이것은 또한 가능한 어트리뷰트 완성을 늘립니다.

완성 상자는 '_' 로 시작하거나, 모듈의 경우 '__all__' 에 포함되지 않은 이름을 처음에는 제외합니다. 숨겨진 이름은 상자를 열기 전이나 후에 '.' 뒤에 '_' 를 입력하여 액세스 할 수 있습니다.

콜팁

액세스할 수 있는 함수 이름 다음에 (를 입력하면 콜팁이 자동으로 표시됩니다. 함수 이름 표현식에는 점과 서브스크립트가 포함될 수 있습니다. 콜팁은 클릭하거나, 커서가 인자 영역 밖으로 이동하거나,) 를 입력할 때까지 남아 있습니다. 커서가 정의의 인자 부분에 있을 때마다, 콜팁을 표시하려면 메뉴에서 Edit 와 "Show Call Tip" 을 선택하거나 바로 가기를 입력하십시오.

콜팁은 함수 서명과 독스트링의 첫 번째 빈 줄이나 다섯 번째 비어 있지 않은 줄에 이르는 줄로 구성됩니다. (일부 내장 함수에는 액세스할 수 있는 서명이 없습니다.) 서명의 '/' 나 '*' 는 앞서거나 뒤따르는 인자가 위치나 이름(키워드) 전용으로 전달됨을 가리킵니다. 세부 사항은 변경될 수 있습니다.

셸에서, 액세스할 수 있는 함수는 Idle 스스로 임포트 한 모듈을 포함하여, 사용자 프로세스로 임포트 한 모듈과 마지막 재시작 이후에 어떤 정의가 실행되었는지에 따라 다릅니다.

예를 들어, 셸을 다시 시작하고 itertools.count(를 입력하십시오. Idle이 자신의 목적으로 itertools를 사용자 프로세스로 임포트 하기 때문에 콜팁이 표시됩니다. (이것은 변경될 수 있습니다.) turtle.write(를 입력하면 아무것도 나타나지 않습니다. Idle은 직접 turtle을 임포트 하지 않습니다. 메뉴 항목과 바로 가기도 아무것도 하지 않습니다. import turtle을 입력하십시오. 그다음부터, turtle.write(가 콜팁을 표시합니다.

편집기에서, import 문은 파일을 실행할 때까지 효과가 없습니다. import 문을 작성한 후, 함수 정의를 추가한 후 또는 기존 파일을 연 후 파일을 실행하고 싶을 것입니다.

코드 컨텍스트

파이썬 코드가 포함된 편집기 창 내에서, 창의 상단에 팬을 표시하거나 숨기기 위해 코드 컨텍스트를 토글할 수 있습니다. 표시될 때, 이 분할 창은 class, def 또는 if 키워드로 시작하는 것과 같이 블록 코드를 여는 줄들을 고정합니다, 그렇지 않다면 뷰에서 스크롤 되어 빠져나갈 줄들입니다. 팬의 크기는 모든 현재 컨텍스트 수준을 표시하기 위해 필요에 따라 확장과 축소되며, IDLE 구성 대화 상자에 정의된 최대 줄 수가지 입니다 (기본값은 15). 현재 컨텍스트 줄이 없고 기능이 켜지면, 빈 줄 하나가 표시됩니다. 컨텍스트 팬에서 줄을 클릭하면 해당 줄을 편집기 맨 위로 이동합니다.

컨텍스트 팬의 텍스트와 배경색은 IDLE 구성 대화 상자의 Highlights 탭에서 구성할 수 있습니다.

셸 창

IDLE의 셸에서, 문장을 입력, 편집하고 완성된 문장을 다시 불러올 수 있습니다. (대부분의 콘솔과 터미널에서는 한 번에 하나의 물리적인 줄만 작업할 수 있습니다).

Submit a single-line statement for execution by hitting `Return` with the cursor anywhere on the line. If a line is extended with Backslash (\), the cursor must be on the last physical line. Submit a multi-line compound statement by entering a blank line after the statement.

셸에 코드를 붙여넣을 때, 위에서 지정한 대로, `Return`을 누를 때까지 컴파일되지도 실행되지도 않습니다. 붙여넣은 코드를 먼저 편집할 수 있습니다. 여러 문장을 셸에 붙여넣으면, 여러 문장이 마치 하나인 것처럼 컴파일되어 결과는 *SyntaxError*가 됩니다.

Lines containing `RESTART` mean that the user execution process has been re-started. This occurs when the user execution process has crashed, when one requests a restart on the Shell menu, or when one runs code in an editor window.

이전 서브 섹션에서 설명한 편집 기능은 대화식으로 코드를 입력할 때 작동합니다. IDLE의 셸 창은 다음에도 반응합니다:

- `C-c` 문장 실행을 중단하려고 시도합니다 (하지만 실패할 수 있습니다).
- `C-d` `>>>` 프롬프트에서 입력하면 셸을 닫습니다.
- `Alt-p` and `Alt-n` (`C-p` and `C-n` on macOS) retrieve to the current prompt the previous or next previously entered statement that matches anything already typed.
- `Return` while the cursor is on any previous statement appends the latter to anything already typed at the prompt.

텍스트 색상

Idle은 기본적으로 흰색 위의 검은색 텍스트지만, 특수한 의미로 텍스트를 채색합니다. 셸의 경우, 셸 출력, 셸 에러, 사용자 출력 및 사용자 에러입니다. 파이썬 코드의 경우, 셸 프롬프트나 편집기에서, 키워드, 내장 클래스와 함수 이름, `class`와 `def` 뒤에 오는 이름, 문자열 및 주석입니다. 모든 텍스트 창에서, 커서 (있다면), 찾은 텍스트 (가능하다면) 및 선택한 텍스트입니다.

IDLE also highlights the soft keywords `match`, `case`, and _ in pattern-matching statements. However, this highlighting is not perfect and will be incorrect in some rare cases, including some _-s in `case` patterns.

텍스트 채색은 백그라운드에서 수행되므로, 채색되지 않은 텍스트가 때때로 표시됩니다. 색 구성표를 변경하려면, IDLE 구성 대화 상자 Highlighting 탭을 사용하십시오. 편집기의 디버거 중단점 표시와 팝업과 대화 상자의 텍스트는 사용자가 구성할 수 없습니다.

25.9.3 시작과 코드 실행

`-s` 옵션으로 시작할 때, IDLE은 환경 변수 `IDLESTARTUP`이나 `PYTHONSTARTUP`가 참조하는 파일을 실행합니다. IDLE은 먼저 `IDLESTARTUP`을 확인합니다; `IDLESTARTUP`이 있으면 참조된 파일이 실행됩니다. `IDLESTARTUP`이 없으면, IDLE은 `PYTHONSTARTUP`을 확인합니다. 이러한 환경 변수가 참조하는 파일은 IDLE 셸에서 자주 사용되는 함수를 저장하거나 공통 모듈을 임포트 하기 위해 import 문을 실행하기에 편리한 장소입니다.

또한, `Tk`도 시작 파일이 있으면 로드합니다. Tk 파일은 무조건 로드됨에 유의하십시오. 이 추가 파일은 `.Idle.py`이며 사용자의 홈 디렉터리에서 찾습니다. 이 파일의 문장은 Tk 이름 공간에서 실행되므로, 이 파일은 IDLE의 파이썬 셸에서 사용할 함수를 임포트 하는 데 유용하지 않습니다.

명령 줄 사용법

IDLE can be invoked from the command line with various options. The general syntax is:

```
python -m idlelib [options] [file ...]
```

The following options are available:

-c <command>

Run the specified Python command in the shell window. For example, pass `-c "print('Hello, World!')"`. On Windows, the outer quotes must be double quotes as shown.

-d

Enable the debugger and open the shell window.

-e

편집기 창을 엽니다.

-h

Print a help message with legal combinations of options and exit.

-i

셸 창을 엽니다.

-r <file>

Run the specified file in the shell window.

-s

Run the startup file (as defined by the environment variables `IDLESTARTUP` or `PYTHONSTARTUP`) before opening the shell window.

-t <title>

셸 창의 제목을 설정합니다.

-

Read and execute standard input in the shell window. This option must be the last one before any arguments.

If arguments are provided:

- `-`, `-c` 또는 `-r`이 사용되면, 모든 인자는 `sys.argv[1:]`에 배치되고 `sys.argv[0]`은 `''`, `'-c'` 또는 `'-r'`로 각각 설정됩니다. *Options* 대화 상자에서 기본값이 설정되어 있어도, 편집기 창이 열리지 않습니다.
- 그렇지 않으면, 인자는 편집을 위해 열리는 파일로 취급되고, `sys.argv`는 IDLE 자체에 전달된 인자를 반영합니다.

시작 실패

IDLE은 소켓을 사용하여 IDLE GUI 프로세스와 사용자 코드 실행 프로세스 간에 통신합니다. 셸을 시작하거나 다시 시작할 때마다 연결을 만들어야 합니다. (후자는 'RESTART' 라고 표시된 구분 선으로 표시됩니다). 사용자 프로세스가 GUI 프로세스에 연결하지 못하면, 보통 사용자에게 알리는 'cannot connect' 메시지가 담긴 `Tk` 에러 상자가 표시됩니다. 그런 다음 종료합니다.

유닉스 시스템의 한가지 특정한 연결 실패는 시스템 네트워크 설정의 어딘가에서 잘못 구성된 마스쿼레이딩 규칙으로 인해 발생합니다. IDLE이 터미널에서 시작될 때, `** Invalid host:`로 시작하는 메시지를 보게 됩니다. 유효한 값은 `127.0.0.1 (idlelib.rpc.LOCALHOST)` 입니다. 한 터미널 창에서 `tcpconnect -irv 127.0.0.1 6543`로, 다른 창에서 `tcplisten <same args>`로 진단할 수 있습니다.

일반적인 실패 원인은 *random.py*와 *tkinter.py*처럼, 표준 라이브러리 모듈과 이름이 같은 사용자가 작성한 파일입니다. 이러한 파일이 실행하려는 파일과 같은 디렉터리에 있을 때, IDLE은 표준 라이브러리 파일을 임포트 할 수 없습니다. 현재 수선법은 사용자 파일의 이름을 바꾸는 것입니다.

앞엣것보다 덜 흔하지만, 바이러스 백신이나 방화벽 프로그램이 연결을 중지할 수 있습니다. 프로그램이 연결을 허용하도록 지시할 수 없으면, IDLE이 작동하도록 프로그램을 꺼야 합니다. 외부 포트에 데이터가 노출되지 않기 때문에 이 내부 연결을 허용하는 것은 안전합니다. 비슷한 문제는 연결을 차단하는 네트워크 구성 에러입니다.

파이썬 설치 문제로 인해 IDLE이 중지되는 경우가 있습니다: 여러 버전이 충돌하거나 단일 설치에 관리자 액세스가 필요할 수 있습니다. 충돌을 제거하거나, 관리자 권한으로 실행할 수 없거나 그러고 싶지 않으면 파이썬을 완전히 제거하고 다시 시작하기가 가장 쉽습니다.

좀비 pythonw.exe 프로세스가 문제일 수 있습니다. 윈도우에서는, 작업 관리자를 사용하여 확인하고 있다면 중지하십시오. 때때로 프로그램 충돌이나 키보드 인터럽트(control-C)에 의해 시작된 재시작이 연결에 실패할 수 있습니다. 에러 상자를 닫거나 Shell 메뉴에서 Restart Shell을 사용하면 일시적인 문제를 해결할 수 있습니다.

IDLE이 처음 시작되면, `~/.idlerc/`에서 사용자 구성 파일을 읽으려고 시도합니다(~는 사용자의 홈 디렉터리입니다). 문제가 있으면, 에러 메시지가 표시되어야 합니다. 임의의 디스크 결함은 예외로 할 때, 파일을 절대 직접 편집하지 않으면 이를 방지할 수 있습니다. 대신, Options의 구성 대화 상자를 사용하십시오. 일단 사용자 구성 파일에 에러가 발생하면, 이를 삭제하고 설정 대화 상자에서 다시 시작하는 것이 가장 좋습니다.

IDLE이 메시지 없이 종료되고, 콘솔에서 시작되지 않았으면, 콘솔이나 터미널에서 시작하고 (`python -m idlelib`) 에러 메시지가 나타나는지 확인하십시오.

Tcl/tk가 `8.6.11`(`About IDLE`을 보십시오)보다 오래된 유닉스 기반 시스템에서 특정 글꼴의 특정 문자는 터미널에 보내는 메시지와 함께 tk 실패를 일으킬 수 있습니다. 이러한 문자가 있는 파일을 편집하기 위해 IDLE을 시작하거나 나중에 이러한 문자를 입력할 때 발생할 수 있습니다. tcl/tk를 업그레이드할 수 없으면, 더 잘 작동하는 글꼴을 사용하도록 IDLE을 다시 구성하십시오.

사용자 코드 실행하기

드문 경우를 제외하고, IDLE을 사용하여 파이썬 코드를 실행한 결과는 텍스트 모드 시스템 콘솔이나 터미널 창에서 파이썬을 사용하여 기본 방법으로 같은 코드를 실행하는 것과 같아야 합니다. 그러나, 다른 인터페이스와 연산은 때때로 시각적인 결과에 영향을 줍니다. 예를 들어, `sys.modules`는 더 많은 항목으로 시작하고, `threading.active_count()`는 1 대신 2를 반환합니다.

기본적으로, IDLE은 셸과 편집기를 실행하는 사용자 인터페이스 프로세스가 아닌 별도의 OS 프로세스에서 사용자 코드를 실행합니다. 실행 프로세스에서, `sys.stdin`, `sys.stdout` 및 `sys.stderr`를 셸 창에서 입력을 받고 셸 창으로 출력을 보내는 객체로 바꿉니다. `sys.__stdin__`, `sys.__stdout__` 및 `sys.__stderr__`에 저장된 원래 값은 건드리지 않지만, `None`일 수 있습니다.

한 프로세스에서 다른 프로세스의 텍스트 위젯으로 인쇄 출력을 보내는 것은 같은 프로세스에서 시스템 터미널로 인쇄하는 것보다 느립니다. 이것은 여러 인자를 인쇄할 때 특히 그렇습니다. 각 인자의 문자열, 각 구분자, 줄 바꿈은 개별적으로 전송되기 때문입니다. 개발할 때는 일반적으로 문제가 되지 않지만, IDLE로 더 빨리 인쇄하려면 표시하려는 모든 항목을 포맷하고 결합한 다음 단일 문자열을 인쇄하십시오. 포맷 문자열과 *`str.join()`* 모두 필드와 줄을 결합하는 데 도움이 될 수 있습니다.

IDLE의 표준 스트림 대체는 (사용자 코드 직접 하거나 multiprocessing과 같은 모듈로) 실행 프로세스에서 만들어진 서브 프로세스에 의해 상속되지 않습니다. 이러한 서브 프로세스가 sys.stdin에서 `input`을 사용하거나 sys.stdout이나 sys.stderr로 `print`나 `write`를 사용하면, 명령 줄 창에서 IDLE을 시작해야 합니다. (윈도우에서는, `pythonw`나 `pyw` 대신 `python`이나 `py`를 사용하세요.) 그러면 이차 서브 프로세스가 입력과 출력을 위해 해당 창에 연결될 겁니다.

`importlib.reload(sys)`와 같은 사용자 코드에 의해 `sys`가 재설정되면, IDLE의 변경 사항이 손실되고 키보드로부터의 입력과 화면으로의 출력이 올바르게 동작하지 않게 됩니다.

셸에 포커스가 있으면, 키보드와 화면을 제어합니다. 이것은 일반적으로 투명하지만, 키보드와 화면에 직접 액세스하는 함수는 작동하지 않습니다. 여기에는 키를 눌렀는지를 판단하는 시스템 특정 함수가 포함됩니다.

실행 프로세스에서 실행 중인 IDLE 코드는 그렇지 않다면 없을 프레임을 호출 스택에 추가합니다. IDLE은 `sys.getrecursionlimit`와 `sys.setrecursionlimit`를 래핑하여 추가 스택 프레임의 영향을 줄입니다.

사용자 코드가 직접 또는 sys.exit를 호출하여 SystemExit를 발생시키면, IDLE은 종료하는 대신 셸 프롬프트로 돌아갑니다.

셸의 사용자 출력

프로그램이 텍스트를 출력할 때, 결과는 해당 출력 장치에 의해 결정됩니다. IDLE이 사용자 코드를 실행할 때, `sys.stdout`과 `sys.stderr`이 IDLE 셸의 표시 영역에 연결됩니다. 그 기능 중 일부는 하부 Tk Text 위젯에서 상속됩니다. 다른 것은 프로그래밍한 추가 사항입니다. (중요하다면) 셸은 상용 실행보다는 개발용으로 설계되었습니다.

예를 들어, 셸은 절대로 출력을 버리지 않습니다. 셸에 무제한 출력을 보내는 프로그램은 결국 메모리를 채워서, 메모리 에러를 일으킵니다. 반대로, 일부 시스템 텍스트 창은 마지막 n 줄의 출력만 유지합니다. 예를 들어, 윈도우 콘솔은 사용자가 설정하면 최대 9999줄을 유지할 수 있으며, 기본값은 300입니다.

Tk Text 위젯, 따라서 IDLE의 셸은 유니코드의 BMP (Basic Multilingual Plane) 부분 집합에서 문자(코드 포인트)를 표시합니다. 어떤 문자가 적절한 글리프로 표시될지와 어떤 문자가 대체 상자로 표시될지는 운영 체제와 설치된 글꼴에 따라 다릅니다. 탭 문자는 뒤따르는 문자가 다음 탭 정지 뒤에서 시작되도록 합니다. (탭 정지는 8 '문자' 마다 나타납니다). 줄 바꿈 문자는 뒤따르는 텍스트가 새 줄에 나타나게 합니다. 다른 제어 문자는 운영 체제와 글꼴에 따라 무시되거나 스페이스, 상자 또는 그 밖의 것으로 표시됩니다. (화살표 키를 사용하여 이러한 출력에서 텍스트 커서를 움직이면 예상치 못한 간격 동작이 나타날 수 있습니다.)

```
>>> s = 'a\tb\a<\x02><\r>\bc\nd'  # 22문자를 입력합니다.
>>> len(s)
14
>>> s  # repr(s)를 표시합니다
'a\tb\x07<\x02><\r>\x08c\nd'
>>> print(s, end='')  # s를 그대로 표시합니다.
# 결과는 OS와 글꼴에 따라 다릅니다. 해보세요.
```

`repr` 함수는 표현식 값의 대화 형 에코에 사용됩니다. 입력 문자열의 변경된 버전을 반환하는데, 제어 코드, 일부 BMP 코드 포인트 및 모든 BMP 이외의 코드 포인트가 이스케이프 코드로 대체됩니다. 위에서 예시했듯이, 어떻게 표시되는지와 관계없이 문자열의 문자를 식별할 수 있도록 합니다.

정상과 에러 출력은 일반적으로 코드 입력 및 서로와 분리되어 별도의 줄에 유지됩니다. 그들은 각각 다른 강조 색상을 얻습니다.

SyntaxError 트레이스백의 경우, 에러가 감지된 곳의 일반적인 '^' 표시는 에러 강조로 텍스트를 색칠하는 것으로 대체됩니다. 파일에서 실행한 코드가 다른 예외를 발생시킬 때, 트레이스백 줄을 마우스 오른쪽 단추로 클릭하여 IDLE 편집기의 해당 줄로 이동할 수 있습니다. 필요하면 파일이 열립니다.

셸에는 출력 라인을 'Squeezed text' 레이블로 압착하는 특수 기능이 있습니다. 이것은 N 줄을 넘어가는 출력에 대해 자동으로 수행됩니다 (기본적으로 N = 50). Settings 대화 상자의 General 페이지의 PyShell 섹션에서 N을 변경할 수 있습니다. 출력을 마우스 오른쪽 버튼으로 클릭하면 더 적은 줄의 출력을 압착할 수 있습니다. 이것은 스크롤 속도를 늦출 수 있을 정도로 긴 줄에 유용할 수 있습니다.

압착된 출력은 레이블을 더블 클릭하여 제자리에서 펼쳐집니다. 레이블을 마우스 오른쪽 버튼으로 클릭하여 클립 보드나 별도의 보기 창으로 보낼 수도 있습니다.

tkinter 응용 프로그램 개발하기

IDLE은 tkinter 프로그램 개발을 용이하게 하기 위해 표준 파이썬과 의도적으로 다릅니다. 표준 파이썬에서 `import tkinter as tk; root = tk.Tk()`를 입력하면 아무것도 나타나지 않습니다. IDLE에 동일하게 입력하면 tk 창이 나타납니다. 표준 파이썬에서는, 창을 보려면 `root.update()`도 입력해야 합니다. IDLE은 초당 약 20회, 대략 50밀리초마다, 백그라운드에서 동등한 일을 합니다. 그런 다음 `b = tk.Button(root, text='button'); b.pack()`을 입력하십시오. 마찬가지로, 표준 파이썬에서는 `root.update()`를 입력할 때까지 아무것도 시각적으로 변경되지 않습니다.

대부분의 tkinter 프로그램은 `root.mainloop()`를 실행하는데, 일반적으로 tk 앱이 파괴될 때까지 반환되지 않습니다. 프로그램이 `python -i`이나 IDLE 편집기에서 실행되면, `mainloop()`가 반환할 때까지 (상호 작용할 것이 남아있지 않을 때까지) `>>>` 셸 프롬프트가 나타나지 않습니다.

IDLE 편집기에서 tkinter 프로그램을 실행할 때, mainloop 호출을 주석 처리할 수 있습니다. 그러면 즉시 셸 프롬프트를 얻고 라이브 응용 프로그램과 상호 작용할 수 있습니다. 표준 파이썬에서 실행할 때 mainloop 호출을 다시 활성화해야 한다는 것을 기억해야 합니다.

서브 프로세스 없이 실행하기

기본적으로, IDLE은 내부 루프 백 인터페이스를 사용하는 소켓을 통해 별도의 서브 프로세스에서 사용자 코드를 실행합니다. 이 연결은 외부에서 볼 수 없으며 인터넷으로 데이터를 보내거나 받지 않습니다. 방화벽 소프트웨어가 어쨌든 불평하면, 무시할 수 있습니다.

소켓 연결 시도가 실패하면, Idle은 여러분에게 알립니다. 이러한 장애는 때때로 일시적이지만, 지속한다면, 문제는 방화벽이 연결을 차단하거나 특정 시스템의 구성이 잘못되었을 수 있습니다. 문제가 해결될

때까지, -n 명령 줄 스위치를 사용하여 Idle을 실행할 수 있습니다.

IDLE이 -n 명령 줄 스위치로 시작되면 단일 프로세스에서 실행되며 RPC 파이썬 실행 서버를 실행하는 서브 프로세스를 만들지 않습니다. 파이썬이 플랫폼에서 서브 프로세스나 RPC 소켓 인터페이스를 만들 수 없을 때 유용할 수 있습니다. 그러나, 이 모드에서 사용자 코드는 IDLE 자체와 분리되지 않습니다. 또한, Run/Run Module (F5) 가 선택될 때 환경이 다시 시작되지 않습니다. 코드가 수정되면, 변경 사항을 적용하려면 영향을 받는 모듈을 reload() 하고 특정 항목을 다시 임포트 해야 합니다 (예를 들어 from foo import baz). 이러한 이유로, 가능하다면 기본 서브 프로세스로 IDLE을 실행하는 것이 좋습니다.

버전 3.4부터 폐지됨.

25.9.4 도움말과 환경 설정

도움말 소스

Help 메뉴 항목 "IDLE Help" 는 라이브러리 레퍼런스의 IDLE 장의 포맷된 html 버전을 표시합니다. 읽기 전용 tkinter 텍스트 창에 표시되는 결과는 웹 브라우저에서 보는 것과 비슷합니다. 마우스 휠, 스크롤 바 또는 위/아래 화살표 키를 누른 상태로 텍스트를 탐색하십시오. 또는 TOC(목차) 단추를 클릭하고 열린 상자에서 섹션 머리글을 선택하십시오.

Help 메뉴 항목 "Python Docs" 는 `docs.python.org/x.y`에 있는 자습서를 포함하여 광범위한 도움말 소스를 엽니다, 여기서 'x.y' 는 현재 실행 중인 파이썬 버전입니다. 시스템에 문서의 오프라인 사본이 있으면 (설치 옵션일 수 있습니다), 대신 열립니다.

IDLE 구성 대화 상자의 General 탭을 사용하여 언제든지 도움말 메뉴에서 선택한 URL을 추가하거나 제거할 수 있습니다.

환경 설정

글꼴 설정, 강조 표시, 키 및 일반 설정은 Option 메뉴의 IDLE 구성을 통해 변경할 수 있습니다. 기본이 아닌 사용자 설정은 사용자 홈 디렉터리의 `.idlerc` 디렉터리에 저장됩니다. 잘못된 사용자 구성 파일로 인한 문제점은 `.idlerc`에서 하나 이상의 파일을 편집하거나 삭제하여 해결됩니다.

Font 탭에서, 여러 언어로 된 여러 문자의 서체와 크기 효과에 대한 텍스트 샘플을 참조하십시오. 개인적으로 관심 있는 다른 문자를 추가하려면 샘플을 편집하십시오. 샘플을 사용하여 고정 폭 글꼴을 선택하십시오. 셸이나 편집기에서 특정 문자에 문제가 있으면, 샘플 상단에 해당 문자를 추가하고 먼저 크기를 변경한 다음 글꼴을 변경해보십시오.

Highlights와 Keys 탭에서, 내장이나 사용자 정의 색상 테마와 키 집합을 선택하십시오. 이전 IDLE로 최신 내장 색상 테마나 키 집합을 사용하려면, 새 사용자 정의 색상 테마나 키 집합으로 저장하십시오, 그러면 이전 IDLE에서 쉽게 액세스 할 수 있습니다.

macOS의 IDLE

시스템 환경설정: Dock에서, "문서를 열 때 탭 사용" 을 "항상" 으로 설정할 수 있습니다. 이 설정은 IDLE에서 사용하는 tk/tkinter GUI 프레임 워크와 호환되지 않으며, 몇 가지 IDLE 기능을 훼손합니다.

확장

IDLE에는 확장 기능이 포함되어 있습니다. 설정 대화 상자의 Extensions 탭에서 확장에 대한 설정을 변경할 수 있습니다. 자세한 정보는 idlelib 디렉터리에 있는 config-extensions.def의 시작 부분을 참조하십시오. 현재 기본 확장은 zzdummy 뿐이며, 테스트에도 사용되는 예입니다.

25.9.5 idlelib — IDLE 응용 프로그램의 구현

소스 코드: Lib/idlelib

The Lib/idlelib package implements the IDLE application. See the rest of this page for how to use IDLE.

The files in idlelib are described in idlelib/README.txt. Access it either in idlelib or click Help => About IDLE on the IDLE menu. This file also maps IDLE menu items to the code that implements the item. Except for files listed under 'Startup', the idlelib code is 'private' in sense that feature changes can be backported (see **PEP 434**).

25.10 `turtle` — Turtle graphics

소스 코드: Lib/turtle.py

25.10.1 소개

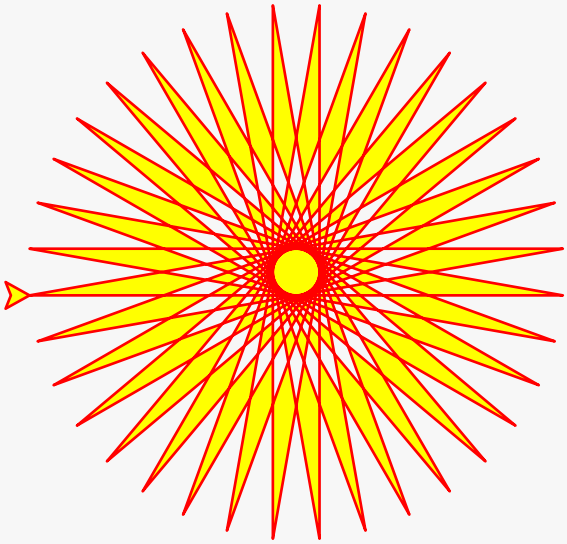
터틀(거북이) 그래픽은 1967년 Wally Feurzeig, Seymour Papert 및 Cynthia Solomon이 개발한 로고(Logo)에서 소개된 대중적인 기하학적 그리기 도구의 구현체입니다.

25.10.2 시작하기

x-y 평면의 (0, 0)에서 출발하는 로봇 거북이를 상상해보십시오. `import turtle` 후에, `turtle.forward(15)` 명령을 내리면, 그것이 향한 방향으로 15픽셀 움직이고 (화면에서!), 움직이면서 선을 그립니다. `turtle.right(25)` 명령을 내려보십시오, 그러면 제자리에서 시계 방향으로 25도 회전합니다.

Turtle star

turtle은 간단한 움직임을 반복하는 프로그램을 사용하여 복잡한 모양을 그릴 수 있습니다.



In Python, turtle graphics provides a representation of a physical "turtle" (a little robot with a pen) that draws on a sheet of paper on the floor.

It' s an effective and well-proven way for learners to encounter programming concepts and interaction with software, as it provides instant, visible feedback. It also provides convenient access to graphical output in general.

Turtle drawing was originally created as an educational tool, to be used by teachers in the classroom. For the programmer who needs to produce some graphical output it can be a way to do that without the overhead of introducing more complex or external libraries into their work.

25.10.3 Tutorial

New users should start here. In this tutorial we' ll explore some of the basics of turtle drawing.

거북이 환경 시작하기

In a Python shell, import all the objects of the `turtle` module:

```
from turtle import *
```

If you run into a `No module named '_tkinter'` error, you' ll have to install the *Tk interface package* on your system.

Basic drawing

Send the turtle forward 100 steps:

```
forward(100)
```

You should see (most likely, in a new window on your display) a line drawn by the turtle, heading East. Change the direction of the turtle, so that it turns 120 degrees left (anti-clockwise):

```
left(120)
```

Let' s continue by drawing a triangle:

```
forward(100)
left(120)
forward(100)
```

Notice how the turtle, represented by an arrow, points in different directions as you steer it.

Experiment with those commands, and also with `backward()` and `right()`.

펜 제어

Try changing the color - for example, `color('blue')` - and width of the line - for example, `width(3)` - and then drawing again.

You can also move the turtle around without drawing, by lifting up the pen: `up()` before moving. To start drawing again, use `down()`.

거북이의 위치

Send your turtle back to its starting-point (useful if it has disappeared off-screen):

```
home()
```

The home position is at the center of the turtle' s screen. If you ever need to know them, get the turtle' s x-y coordinates with:

```
pos()
```

Home is at `(0, 0)`.

And after a while, it will probably help to clear the window so we can start anew:

```
clearscreen()
```

Making algorithmic patterns

Using loops, it' s possible to build up geometric patterns:

```
for steps in range(100):
    for c in ('blue', 'red', 'green'):
        color(c)
        forward(steps)
        right(30)
```

- which of course, are limited only by the imagination!

Let' s draw the star shape at the top of this page. We want red lines, filled in with yellow:

```
color('red')
fillcolor('yellow')
```

Just as `up()` and `down()` determine whether lines will be drawn, filling can be turned on and off:

```
begin_fill()
```

Next we' ll create a loop:

```
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
```

`abs(pos()) < 1` is a good way to know when the turtle is back at its home position.

Finally, complete the filling:

```
end_fill()
```

(Note that filling only actually takes place when you give the `end_fill()` command.)

25.10.4 How to…

This section covers some typical turtle use-cases and approaches.

Get started as quickly as possible

One of the joys of turtle graphics is the immediate, visual feedback that' s available from simple commands - it' s an excellent way to introduce children to programming ideas, with a minimum of overhead (not just children, of course).

The turtle module makes this possible by exposing all its basic functionality as functions, available with `from turtle import *`. The *turtle graphics tutorial* covers this approach.

It' s worth noting that many of the turtle commands also have even more terse equivalents, such as `fd()` for `forward()`. These are especially useful when working with learners for whom typing is not a skill.

> You' ll need to have the `Tk interface package` installed on your system for turtle graphics to work. Be warned that this is not always straightforward, so check this in advance if you' re planning to use turtle graphics with a learner.

Automatically begin and end filling

Starting with Python 3.14, you can use the `fill()` *context manager* instead of `begin_fill()` and `end_fill()` to automatically begin and end fill. Here is an example:

```
with fill():
    for i in range(4):
        forward(100)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        right(90)

forward(200)
```

The code above is equivalent to:

```
begin_fill()
for i in range(4):
    forward(100)
    right(90)
end_fill()

forward(200)
```

Use the `turtle` module namespace

Using `from turtle import *` is convenient - but be warned that it imports a rather large collection of objects, and if you' re doing anything but turtle graphics you run the risk of a name conflict (this becomes even more an issue if you' re using turtle graphics in a script where other modules might be imported).

The solution is to use `import turtle` - `fd()` becomes `turtle.fd()`, `width()` becomes `turtle.width()` and so on. (If typing "turtle" over and over again becomes tedious, use for example `import turtle as t` instead.)

스크립트에서 거북이 그래픽 사용하기

It' s recommended to use the `turtle` module namespace as described immediately above, for example:

```
import turtle as t
from random import random

for i in range(100):
    steps = int(random() * 100)
    angle = int(random() * 360)
    t.right(angle)
    t.fd(steps)
```

Another step is also required though - as soon as the script ends, Python will also close the turtle' s window. Add:

```
t.mainloop()
```

to the end of the script. The script will now wait to be dismissed and will not exit until it is terminated, for example by closing the turtle graphics window.

Use object-oriented turtle graphics

> **더 보기**
>
> *Explanation of the object-oriented interface*

Other than for very basic introductory purposes, or for trying things out as quickly as possible, it' s more usual and much more powerful to use the object-oriented approach to turtle graphics. For example, this allows multiple turtles on screen at once.

In this approach, the various turtle commands are methods of objects (mostly of `Turtle` objects). You *can* use the object-oriented approach in the shell, but it would be more typical in a Python script.

The example above then becomes:

```
from turtle import Turtle
from random import random

t = Turtle()
for i in range(100):
    steps = int(random() * 100)
    angle = int(random() * 360)
    t.right(angle)
    t.fd(steps)

t.screen.mainloop()
```

Note the last line. `t.screen` is an instance of the *Screen* that a Turtle instance exists on; it' s created automatically along with the turtle.

The turtle' s screen can be customised, for example:

```
t.screen.title('Object-oriented turtle demo')
t.screen.bgcolor("orange")
```

25.10.5 Turtle graphics reference

> **참고**
>
> 다음 설명서에서 함수의 인자 목록이 제공됩니다. 물론 메서드에는 추가의 첫 번째 인자 *self* 가 있으며 여기서는 생략합니다.

Turtle 메서드

거북이 움직임

이동과 그리기

forward() | *fd()*
backward() | *bk()* | *back()*
right() | *rt()*
left() | *lt()*
goto() | *setpos()* | *setposition()*
teleport()
setx()
sety()
setheading() | *seth()*
home()
circle()
dot()
stamp()
clearstamp()
clearstamps()
undo()
speed()

거북이의 상태 보고

position() | *pos()*
towards()
xcor()

ycor()
heading()
distance()

설정과 측정

degrees()
radians()

펜 제어

그리기 상태

pendown() | *pd()* | *down()*
penup() | *pu()* | *up()*
pensize() | *width()*
pen()
isdown()

색상 제어

color()
pencolor()
fillcolor()

채우기

filling()
fill()
begin_fill()
end_fill()

더 많은 그리기 제어

reset()
clear()
write()

거북이 상태

가시성

showturtle() | *st()*
hideturtle() | *ht()*
isvisible()

외관

shape()
resizemode()
shapesize() | *turtlesize()*
shearfactor()
tiltangle()
tilt()
shapetransform()
get_shapepoly()

이벤트 사용하기

onclick()
onrelease()
ondrag()

특수 Turtle 메서드

poly()
begin_poly()
end_poly()
get_poly()
clone()
getturtle() | *getpen()*
getscreen()
setundobuffer()
undobufferentries()

TurtleScreen/Screen의 메서드

창 제어

bgcolor()
bgpic()
clearscreen()
resetscreen()
screensize()
setworldcoordinates()

애니메이션 제어

no_animation()
delay()
tracer()
update()

화면 이벤트 사용하기

listen()
onkey() | *onkeyrelease()*
onkeypress()
onclick() | *onscreenclick()*
ontimer()
mainloop() | *done()*

설정과 특수 메서드

mode()
colormode()
getcanvas()
getshapes()
register_shape() | *addshape()*
turtles()
window_height()
window_width()

입력 메서드

textinput()
numinput()

Screen 특정 메서드

bye()
exitonclick()
save()
setup()

title()

25.10.6 RawTurtle/Turtl의 메서드와 해당 함수

이 섹션의 대부분의 예제는 `turtle`이라는 Turtle 인스턴스를 참조합니다.

거북이 움직임

`turtle.`**`forward`**(*distance*)

`turtle.`**`fd`**(*distance*)

매개변수

distance – 숫자 (정수나 실수)

거북이가 향한 방향으로, 지정된 *distance*만큼 거북이를 앞으로 움직입니다.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

`turtle.`**`back`**(*distance*)

`turtle.`**`bk`**(*distance*)

`turtle.`**`backward`**(*distance*)

매개변수

distance – 숫자

거북이가 향한 반대 방향으로, *distance*만큼 거북이를 뒤로 움직입니다. 거북이의 방향을 바꾸지 않습니다.

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

`turtle.`**`right`**(*angle*)

`turtle.`**`rt`**(*angle*)

매개변수

angle – 숫자 (정수나 실수)

angle 단위만큼 거북이를 오른쪽으로 회전합니다. (단위는 기본적으로 도(degree)이지만, *degrees()*와 *radians()* 함수를 통해 설정할 수 있습니다.) 각도 방향은 거북이 모드에 따라 다릅니다, *mode()*를 참조하십시오.

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

`turtle.`**`left`**(*angle*)

turtle.**lt**(*angle*)

매개변수
angle – 숫자 (정수나 실수)

angle 단위만큼 거북이를 왼쪽으로 회전합니다. (단위는 기본적으로 도(degree)이지만, *degrees()*와 *radians()* 함수를 통해 설정할 수 있습니다.) 각도 방향은 거북이 모드에 따라 다릅니다, *mode()*를 참조하십시오.

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

turtle.**goto**(*x*, *y=None*)

turtle.**setpos**(*x*, *y=None*)

turtle.**setposition**(*x*, *y=None*)

매개변수

- **x** – 숫자나 숫자의 쌍/벡터
- **y** – 숫자나 None

*y*가 None이면, *x*는 좌표 쌍이거나 *Vec2D*여야 합니다 (예를 들어 *pos()*에서 반환된 것과 같은).

거북이를 절대 위치로 움직입니다. 펜이 내려져 있으면, 선을 그립니다. 거북이의 방향을 바꾸지 않습니다.

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

turtle.**teleport**(*x*, *y=None*, *, *fill_gap=False*)

매개변수

- **x** – 숫자나 None
- **y** – 숫자나 None
- **fill_gap** – a boolean

Move turtle to an absolute position. Unlike goto(x, y), a line will not be drawn. The turtle' s orientation does not change. If currently filling, the polygon(s) teleported from will be filled after leaving, and filling will begin again after teleporting. This can be disabled with fill_gap=True, which makes the imaginary line traveled during teleporting act as a fill barrier like in goto(x, y).

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.teleport(60)
>>> turtle.pos()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
(60.00,0.00)
>>> turtle.teleport(y=10)
>>> turtle.pos()
(60.00,10.00)
>>> turtle.teleport(20, 30)
>>> turtle.pos()
(20.00,30.00)
```

Added in version 3.12.

turtle.**setx**(*x*)

매개변수
x – 숫자 (정수나 실수)

거북이의 첫 번째 좌표를 *x*로 설정하고, 두 번째 좌표는 변경하지 않습니다.

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

turtle.**sety**(*y*)

매개변수
y – 숫자 (정수나 실수)

거북이의 두 번째 좌표를 *y*로 설정하고, 첫 번째 좌표는 변경하지 않습니다.

```
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

turtle.**setheading**(*to_angle*)
turtle.**seth**(*to_angle*)

매개변수
to_angle – 숫자 (정수나 실수)

거북이의 방향을 *to_angle*로 설정합니다. 다음은 몇 가지 일반적인 도(degree)로 나타낸 방향입니다:

표준 모드	로고 모드
0 - 동	0 - 북
90 - 북	90 - 동
180 - 서	180 - 남
270 - 남	270 - 서

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle.**home**()

거북이를 원점 – 좌표 (0,0) – 으로 이동하고 방향을 시작 방향으로 설정합니다 (모드에 따라 다릅니다, *mode()*를 참조하십시오).

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**circle**(*radius*, *extent=None*, *steps=None*)

매개변수

- **radius** – 숫자
- **extent** – 숫자 (또는 None)
- **steps** – 정수 (또는 None)

주어진 반지름(*radius*)으로 원을 그립니다. 중심은 거북이 왼쪽으로 *radius* 단위입니다; *extent* – 각도 – 는 원의 어느 부분이 그려지는지를 결정합니다. *extent*가 주어지지 않으면, 전체 원을 그립니다. *extent*가 완전한 원이 아니면, 호의 한 끝점이 현재 펜 위치입니다. *radius*가 양수면 시계 반대 방향으로, 그렇지 않으면 시계 방향으로 호를 그립니다. 마지막으로 거북이의 방향이 *extent*만큼 변경됩니다.

원은 내접하는 정다각형으로 근사되므로, *steps*가 사용할 단계 수를 결정합니다. 제공하지 않으면, 자동으로 계산됩니다. 정다각형을 그리는 데 사용할 수 있습니다.

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180)  # 반원을 그립니다
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

turtle.**dot**(*size=None*, **color*)

매개변수

- **size** – 정수 >= 1 (주어지면)
- **color** – 색상 문자열이나 숫자 색상 튜플

*color*를 사용하여 지름이 *size*인 원형 점을 그립니다. *size*가 제공되지 않으면, pensize+4와 2*pensize 중 큰 값이 사용됩니다.

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

turtle.**stamp**()

거북이 모양의 사본을 현재 거북이 위치에서 캔버스에 찍습니다. 해당 스탬프에 대한 stamp_id를 반환하는데, `clearstamp(stamp_id)` 를 호출하여 스탬프를 삭제하는 데 사용할 수 있습니다.

```
>>> turtle.color("blue")
>>> stamp_id = turtle.stamp()
>>> turtle.fd(50)
```

turtle.**clearstamp**(*stampid*)

매개변수

stampid – 정수, 이전 *stamp()* 호출의 반환 값이어야 합니다

지정된 *stampid*의 스탬프를 삭제합니다.

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

turtle.**clearstamps**(*n=None*)

매개변수

n – 정수 (또는 `None`)

거북이 스탬프의 전부나 처음/마지막 n 개를 삭제합니다. n이 `None`이면, 모든 스탬프를 삭제합니다, $n > 0$ 이면 처음 n 스탬프를 삭제하고, $n < 0$ 이면 마지막 n 스탬프를 삭제합니다.

```
>>> for i in range(8):
...     unused_stamp_id = turtle.stamp()
...     turtle.fd(30)
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

turtle.**undo**()

마지막 거북이의 행동을 (반복적으로) 되돌립니다. 되돌릴 수 있는 행동의 수는 언두버퍼(undobuffer)의 크기에 따라 결정됩니다.

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

turtle.**speed**(*speed=None*)

매개변수

speed – 0..10 범위의 정수나 속도 문자열 (아래를 참조하십시오)

거북이의 속도를 0..10 범위의 정숫값으로 설정합니다. 인자가 없으면, 현재 속도를 반환합니다.

입력이 10보다 크거나 0.5보다 작은 숫자면, 속도는 0으로 설정됩니다. 속도 문자열은 다음과 같이 속도 값에 매핑됩니다:

- "fastest": 0

- "fast": 10
- "normal": 6
- "slow": 3
- "slowest": 1

1에서 10까지의 속도는 선 그리기와 거북이 회전의 애니메이션이 점점 더 빨라집니다.

주의: *speed* = 0 은 애니메이션이 발생하지 않음을 의미합니다. forward/back은 거북이가 점프하게 만들고 마찬가지로 left/right는 거북이가 순간적으로 방향을 바꾸게 만듭니다.

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

거북이의 상태 보고

turtle.**position**()

turtle.**pos**()

거북이의 현재 위치 (x, y)를 (*Vec2D* 벡터로) 반환합니다.

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.**towards**(*x*, *y=None*)

매개변수

- **x** – 숫자 또는 숫자 쌍/벡터 또는 거북이 인스턴스
- **y** – *x*가 숫자면 숫자, 그렇지 않으면 None

거북이 위치에서 (x, y), 벡터 또는 다른 거북이로 지정된 위치로의 선과의 각도를 반환합니다. 이것은 거북이의 시작 방향에 따라 다른데, 이는 모드에 따라 다릅니다 - "standard"/"world" 또는 "logo".

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.**xcor**()

거북이의 x 좌표를 반환합니다.

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

turtle.**ycor**()

거북이의 y 좌표를 반환합니다.

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00,86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

turtle.**heading**()

거북이의 현재 방향을 반환합니다 (값은 거북이 모드에 따라 다릅니다. *mode()* 를 참조하십시오).

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.**distance**(*x*, *y=None*)

매개변수

- **x** – 숫자 또는 숫자 쌍/벡터 또는 거북이 인스턴스
- **y** – *x* 가 숫자면 숫자, 그렇지 않으면 None

거북이에서 (x, y), 주어진 벡터 또는 주어진 다른 거북이까지의 거리를 거북이 단계 단위로 반환합니다.

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

측정 설정

turtle.**degrees**(*fullcircle=360.0*)

매개변수

fullcircle – 숫자

각도 측정 단위를 설정합니다, 즉 전체 원에 대한 "도 (degrees)" 의 수를 설정합니다. 기본값은 360 도입니다.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

>>> # 각도 측정 단위를 그레이드(grad)로 변경합니다 (곤(gon),
>>> # grade, 그라디안(gradian)이라고도 하며 직각의 1/100 에 해당합니다.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

turtle.**radians**()

각도 측정 단위를 라디안으로 설정합니다. `degrees(2*math.pi)` 와 동등합니다.

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

펜 제어

그리기 상태

turtle.**pendown**()

turtle.**pd**()

turtle.**down**()

펜을 내립니다 – 움직일 때 그립니다.

turtle.**penup**()

turtle.**pu**()

turtle.**up**()

펜을 올립니다 – 움직일 때 그리지 않습니다.

turtle.**pensize**(*width=None*)

turtle.**width**(*width=None*)

매개변수

width – 양수

선 두께를 *width* 로 설정하거나 반환합니다. 크기 조정 모드(resizemode) 가 "auto" 로 설정되고 거북이 모양(shape) 이 다각형이면, 해당 다각형은 같은 선 두께로 그려집니다. 인자가 없으면, 현재 펜 크기가 반환됩니다.

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)    # 여기서부터 두께가 10인 선이 그려집니다
```

turtle.**pen**(*pen=None*, ***pendict*)

매개변수

- **pen** – 아래 나열된 키 중 일부나 전부가 포함된 딕셔너리
- **pendict** – 아래에 나열된 키를 키워드로 사용하는 하나 이상의 키워드 인자

다음 키/값 쌍으로 "펜 딕셔너리" 에 있는 펜의 속성을 반환하거나 설정합니다:

- "shown": True/False
- "pendown": True/False
- "pencolor": 색상 문자열이나 색상 튜플
- "fillcolor": 색상 문자열이나 색상 튜플
- "pensize": 양수
- "speed": 0..10 범위의 숫자
- "resizemode": "auto" 또는 "user" 또는 "noresize"
- "stretchfactor": (양수, 양수)

- "outline": 양수
- "tilt": 숫자

이 딕셔너리는 이전 펜 상태를 복원하기 위해 *pen()*의 후속 호출에 대한 인자로 사용될 수 있습니다. 또한 이러한 속성 중 하나 이상을 키워드 인자로 제공할 수 있습니다. 하나의 문장에서 여러 펜 속성을 설정하는 데 사용할 수 있습니다.

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]
```

turtle.**isdown**()

펜이 내려가 있으면 `True`를, 올라가 있으면 `False`를 반환합니다.

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

색상 제어

turtle.**pencolor**(**args*)

펜 색상(pencolor)을 반환하거나 설정합니다.

네 가지 입력 형식이 허용됩니다:

pencolor()
현재 펜 색상을 색상 지정 문자열이나 튜플로 반환합니다 (예를 참조하십시오). 다른 color/pencolor/fillcolor 호출에 대한 입력으로 사용될 수 있습니다.

pencolor(colorstring)
펜 색상을 `"red"`, `"yellow"` 또는 `"#33cc8c"`와 같은 Tk 색상 지정 문자열인 *colorstring*으로 설정합니다.

pencolor((r, g, b))
펜 색상을 *r*, *g* 및 *b*의 튜플로 표현되는 RGB 색상으로 설정합니다. *r*, *g* 및 *b* 각각은 0..colormode 범위에 있어야 합니다. 여기서 colormode는 1.0이나 255입니다 (*colormode()*를 참조하십시오).

pencolor(r, g, b)
펜 색상을 *r*, *g* 및 *b*로 표현되는 RGB 색상으로 설정합니다. *r*, *g* 및 *b*는 각각 0..colormode 범위에 있어야 합니다.

거북이 모양이 다각형이면, 해당 다각형의 외곽선은 새로 설정된 펜 색상으로 그려집니다.

```
>>> colormode()
1.0
>>> turtle.pencolor()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

turtle.**fillcolor**(**args*)

채우기 색상(fillcolor)을 반환하거나 설정합니다.

네 가지 입력 형식이 허용됩니다:

fillcolor()

현재 채우기 색상을 색상 지정 문자열로 (튜플 형식으로도 가능합니다) 반환합니다 (예를 참조하십시오). 다른 color/pencolor/fillcolor 호출에 대한 입력으로 사용될 수 있습니다.

fillcolor(colorstring)

채우기 색상을 `"red"`, `"yellow"` 또는 `"#33cc8c"`와 같은 Tk 색상 지정 문자열인 *colorstring*으로 설정합니다.

fillcolor((r, g, b))

채우기 색상을 *r*, *g* 및 *b*의 튜플로 표현되는 RGB 색상으로 설정합니다. *r*, *g* 및 *b* 각각은 0..colormode 범위에 있어야 합니다. 여기서 colormode는 1.0이나 255입니다 (*colormode()*를 참조하십시오).

fillcolor(r, g, b)

채우기 색상을 *r*, *g* 및 *b*로 표현되는 RGB 색상으로 설정합니다. *r*, *g* 및 *b*는 각각 0..colormode 범위에 있어야 합니다.

거북이 모양이 다각형이면, 해당 다각형의 내부는 새로 설정된 채우기 색상으로 그려집니다.

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor((50, 193, 143))  # 정수입니다, 실수가 아닙니다
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

turtle.**color**(**args*)

펜 색상과 채우기 색상을 반환하거나 설정합니다.

몇 가지 입력 형식이 허용됩니다. 다음과 같이 0에서 3개의 인자를 사용합니다:

color()

*pencolor()*와 *fillcolor()*에 의해 반환된 현재 펜 색상과 현재 채우기 색상을 한 쌍의 색 지정 문자열이나 튜플로 반환합니다.

color(colorstring), color((r,g,b)), color(r,g,b)
*pencolor()*에서와 같은 입력, 채우기 색상과 펜 색상을 모두 주어진 값으로 설정합니다.

color(colorstring1, colorstring2), color((r1,g1,b1), (r2,g2,b2))
pencolor(colorstring1)과 fillcolor(colorstring2)와 동등하며 다른 입력 형식을 사용하는 경우도 유사합니다.

거북이 모양이 다각형이면, 해당 다각형의 외곽선과 내부가 새로 설정된 색상으로 그려집니다.

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

참조: Screen 메서드 *colormode()*.

채우기

turtle.**filling**()
채우기 상태(fillstate)를 반환합니다(채우면 True, 그렇지 않으면 False).

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

turtle.**fill**()
Fill the shape drawn in the with turtle.fill(): block.

```
>>> turtle.color("black", "red")
>>> with turtle.fill():
...     turtle.circle(80)
```

Using fill() is equivalent to adding the *begin_fill()* before the fill-block and *end_fill()* after the fill-block:

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

Added in version 3.14.

turtle.**begin_fill**()
채울 모양을 그리기 직전에 호출됩니다.

turtle.**end_fill**()
*begin_fill()*을 마지막으로 호출한 후 그린 모양을 채웁니다.

스스로 교차하는 다각형이나 여러 도형의 겹치는 영역이 채워지는지는 운영 체제 그래픽, 겹침의 유형 및 겹침의 수에 따라 다릅니다. 예를 들어, 위의 거북이 별은 모두 노란색이거나 일부 흰색 영역이 있을 수 있습니다.

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

더 많은 그리기 제어

turtle.**reset**()

화면에서 거북이의 그림을 삭제하고, 거북이를 다시 중심으로 옮기고, 변수를 기본값으로 설정합니다.

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**clear**()

화면에서 거북이 그림을 삭제합니다. 거북이를 움직이지 않습니다. 다른 거북이의 그림뿐만 아니라 거북이의 상태와 위치는 영향을 받지 않습니다.

turtle.**write**(*arg*, *move=False*, *align='left'*, *font=('Arial', 8, 'normal')*)

매개변수

- **arg** – TurtleScreen에 기록될 객체
- **move** – True/False
- **align** – "left", "center" 또는 "right" 문자열 중 하나
- **font** – 3-튜플 (fontname, fontsize, fonttype)

align("left", "center" 또는 "right")에 따라 현재 거북이 위치에서 주어진 글꼴(font)로 텍스트 - *arg*의 문자열 표현 - 를 기록합니다. *move*가 참이면, 펜이 텍스트의 오른쪽 아래 모서리로 이동합니다. 기본적으로, *move*는 `False`입니다.

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

거북이 상태

가시성

turtle.**hideturtle**()

turtle.**ht**()

거북이를 보이지 않게 합니다. 거북이를 숨기면 그리기 속도가 눈에 띄게 빨라지므로, 복잡한 그리기를 하는 동안 이렇게 하는 것이 좋습니다.

```
>>> turtle.hideturtle()
```

turtle.**showturtle**()

turtle.**st**()

거북이가 보이게 합니다.

```
>>> turtle.showturtle()
```

turtle.**isvisible**()

거북이가 보이면 `True`를, 숨겨져 있으면 `False`를 반환합니다.

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

외관

turtle.**shape**(*name=None*)

매개변수

name – 유효한 모양 이름(shapename)인 문자열

주어진 *name*의 모양으로 거북이 모양을 설정하거나, 이름이 없으면 현재 모양의 이름을 반환합니다. *name*의 모양은 TurtleScreen의 모양 딕셔너리에 있어야 합니다. 처음에는 다음과 같은 다각형 모양이 있습니다: "arrow", "turtle", "circle", "square", "triangle", "classic". 모양을 다루는 방법에 대한 자세한 내용은 Screen 메서드 `register_shape()`을 참조하십시오.

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

turtle.**resizemode**(*rmode=None*)

매개변수

rmode – 문자열 "auto", "user", "noresize" 중 하나

크기 조정 모드(resizemode)를 다음 값 중 하나로 설정합니다: "auto", "user", "noresize". *rmode*가 제공되지 않으면, 현재 크기 조정 모드를 반환합니다. 각 크기 조정 모드는 다음과 같은 효과가 있습니다:

- "auto": 펜 크기(pensize)의 값에 맞춰 거북이의 외관을 조정합니다.
- "user": `shapesize()`로 설정된 stretchfactor와 outlinewidth (outline) 값에 따라 거북이의 외관을 조정합니다.
- "noresize": 거북이의 외관 조정이 일어나지 않습니다.

`shapesize()`에 인자를 사용하면 `resizemode("user")` 를 호출합니다.

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.**shapesize**(*stretch_wid=None, stretch_len=None, outline=None*)

turtle.**turtlesize**(*stretch_wid=None, stretch_len=None, outline=None*)

매개변수

- **stretch_wid** – 양수
- **stretch_len** – 양수
- **outline** – 양수

펜의 속성 x/y-stretchfactor 및/또는 outline을 반환하거나 설정합니다. 크기 조정 모드(resizemode)를 "user"로 설정합니다. 크기 조정 모드(resizemode)가 "user"로 설정될 때만, 거북이가 신축 계수(stretchfactor)에 따라 늘려서 표시됩니다: *stretch_wid*는 방향에 수직인 신축 계수, *stretch_len*은 방향 쪽의 신축 계수이고, *outline*은 모양의 윤곽의 너비를 결정합니다.

```
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.**shearfactor**(*shear=None*)

매개변수
shear – 숫자 (선택 사항)

기울기 계수(shearfactor)를 설정하거나 반환합니다. 주어진 기울기 계수 shear(기울기 각의 탄젠트입니다)에 따라 거북이 모양을 기울입니다. 거북이의 방향(이동 방향)을 변경하지 않습니다. shear가 제공되지 않으면: 현재 기울기 계수(shearfactor)를, 즉 기울기 각도의 탄젠트를 반환합니다. 기울기 각도는 거북이의 방향에 평행한 직선이 기울어진 각도입니다.

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

turtle.**tilt**(*angle*)

매개변수
angle – 숫자

현재 틸트 각도(tilt-angle)에서 거북이 모양을 *angle*만큼 회전합니다. 그러나 거북이의 방향(이동 방향)을 변경하지 않습니다.

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

turtle.**tiltangle**(*angle=None*)

매개변수
angle – 숫자 (선택 사항)

현재 틸트 각도(tilt-angle)를 설정하거나 반환합니다. angle이 주어지면, 현재 틸트 각도(tilt-angle)와 관계없이 거북이 모양을 angle이 지정하는 방향을 가리키도록 회전합니다. 거북이의 방향(이동 방향)을 변경하지 않습니다. angle이 주어지지 않으면: 현재 틸트 각도, 즉 거북이 모양의 방향과 거북이 방향(이동 방향) 사이의 각도를 반환합니다.

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

turtle.**shapetransform**(*t11=None, t12=None, t21=None, t22=None*)

매개변수

- **t11** – 숫자 (선택 사항)
- **t12** – 숫자 (선택 사항)
- **t21** – 숫자 (선택 사항)
- **t12** – 숫자 (선택 사항)

거북이 모양의 현재 변환 행렬을 설정하거나 반환합니다.

행렬 요소가 아무것도 제공되지 않으면, 변환 행렬을 4개 요소의 튜플로 반환합니다. 그렇지 않으면, 주어진 요소를 설정하고 첫 번째 행 t11, t12와 두 번째 행 t21, t22로 구성된 행렬에 따라 거북이 모양을 변환합니다. 행렬식(determinant) t11 * t22 - t12 * t21은 0이 아니어야 합니다, 그렇지 않으면 에러가 발생합니다. 주어진 행렬에 따라 신축 계수(stretchfactor), 기울기 계수(shearfactor) 및 틸트 각도(tiltangle)를 수정합니다.

```
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)
```

turtle.**get_shapepoly**()

현재 모양 다각형을 좌표 쌍의 튜플로 반환합니다. 이것은 새로운 모양이나 복합 모양의 구성 요소를 정의하는 데 사용할 수 있습니다.

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

이벤트 사용하기

turtle.**onclick**(*fun*, *btn=1*, *add=None*)

매개변수

- **fun** – 캔버스에서 클릭한 점의 좌표로 호출되는 두 개의 인자가 있는 함수
- **btn** – 마우스 버튼 수, 기본값은 1 (마우스 왼쪽 버튼)
- **add** – `True` 또는 `False` – `True`이면, 새 연결이 추가되고, 그렇지 않으면 이전 연결을 대체합니다

이 거북이의 마우스 클릭 이벤트에 *fun*을 연결합니다. *fun*이 `None`이면 기존 연결이 제거됩니다. 익명의 거북이, 즉 절차적 방법의 예:

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # 이제 거북이를 클릭하면 회전합니다.
>>> onclick(None)  # 이벤트 연결을 제거합니다
```

turtle.**onrelease**(*fun*, *btn=1*, *add=None*)

매개변수

- **fun** – 캔버스에서 클릭한 점의 좌표로 호출되는 두 개의 인자가 있는 함수
- **btn** – 마우스 버튼 수, 기본값은 1 (마우스 왼쪽 버튼)

- **add** – True 또는 False – True 이면, 새 연결이 추가되고, 그렇지 않으면 이전 연결을 대체합니다

이 거북이의 마우스 버튼 해제 이벤트에 *fun*을 연결합니다. *fun*이 None 이면 기존 연결이 제거됩니다.

```
>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)     # 거북이를 클릭하면 붉은색으로 채웁니다,
>>> turtle.onrelease(turtle.unglow) # 버튼 해제하면 투명하게 만듭니다.
```

turtle.**ondrag**(*fun*, *btn=1*, *add=None*)

매개변수

- **fun** – 캔버스에서 클릭한 점의 좌표로 호출되는 두 개의 인자가 있는 함수
- **btn** – 마우스 버튼 수, 기본값은 1 (마우스 왼쪽 버튼)
- **add** – True 또는 False – True 이면, 새 연결이 추가되고, 그렇지 않으면 이전 연결을 대체합니다

이 거북이의 마우스 이동 이벤트에 *fun*을 연결합니다. *fun*이 None 이면 기존 연결이 제거됩니다.

참고: 거북이의 모든 마우스 이동 이벤트에 앞서 해당 거북이의 마우스 클릭 이벤트가 선행합니다.

```
>>> turtle.ondrag(turtle.goto)
```

그 후, 거북이를 클릭하고 드래그하면 화면을 가로질러 거북이가 움직여 손 그림을 생성합니다 (펜이 내려가 있다면).

특수 Turtle 메서드

turtle.**poly**()

Record the vertices of a polygon drawn in the `with turtle.poly():` block. The first and last vertices will be connected.

```
>>> with turtle.poly():
...     turtle.forward(100)
...     turtle.right(60)
...     turtle.forward(100)
```

Added in version 3.14.

turtle.**begin_poly**()

다각형의 꼭짓점 기록을 시작합니다. 현재 거북이 위치가 다각형의 첫 번째 꼭짓점입니다.

turtle.**end_poly**()

다각형의 꼭짓점 기록을 중지합니다. 현재 거북이 위치가 다각형의 마지막 꼭짓점입니다. 첫 번째 꼭짓점과 연결됩니다.

turtle.**get_poly**()

마지막으로 기록된 다각형을 반환합니다.

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

turtle.**clone**()

같은 위치, 방향 및 거북이 속성을 가진 거북이 복제본을 만들고 반환합니다.

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle.**getturtle**()

turtle.**getpen**()

거북이 객체 자체를 반환합니다. 합리적인 용도로만 사용하십시오: "익명 거북이"를 반환하는 함수로:

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

turtle.**getscreen**()

거북이가 그리는 *TurtleScreen* 객체를 반환합니다. 그런 다음 해당 객체에 대해 TurtleScreen 메서드를 호출할 수 있습니다.

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle.**setundobuffer**(*size*)

매개변수

size – 정수나 None

언두버퍼(undobuffer)를 설정하거나 비활성화합니다. *size*가 정수이면, 지정된 크기의 빈 언두버퍼가 설치됩니다. *size*는 *undo()* 메서드/함수로 취소할 수 있는 최대 거북이 액션 수를 제공합니다. *size*가 None이면, 언두버퍼가 비활성화됩니다.

```
>>> turtle.setundobuffer(42)
```

turtle.**undobufferentries**()

언두버퍼에 있는 항목 수를 반환합니다.

```
>>> while undobufferentries():
...     undo()
```

복합 모양

다른 색상의 여러 다각형으로 구성된 복합 거북이 모양을 사용하려면, 아래 설명된 대로 도우미 클래스 *Shape*을 명시적으로 사용해야 합니다:

1. "compound" 유형의 빈 Shape 객체를 만듭니다.
2. *addcomponent()* 메서드를 사용하여, 원하는 만큼 이 객체에 구성 요소를 추가합니다.

   예를 들면:

```
>>> s = Shape("compound")
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. 이제 Shape을 Screen의 모양 리스트(shapelist)에 추가하고 사용합니다:

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

> **참고**
>
> *Shape* 클래스는 *register_shape()* 메서드에 의해 내부적으로 다른 방식으로 사용됩니다. 응용 프로그램 프로그래머는 위와 같이 복합 모양을 사용할 때 만 Shape 클래스를 다뤄야 합니다!

25.10.7 TurtleScreen/Screen 메서드와 해당 함수

이 섹션의 대부분의 예제는 `screen`이라는 TurtleScreen 인스턴스를 참조합니다.

창 제어

`turtle.`**`bgcolor`**(**args*)

매개변수
`args` – 색상 문자열이나 0..colormode 범위의 3개의 숫자 또는 이러한 숫자의 3-튜플

TurtleScreen의 배경색을 설정하거나 반환합니다.

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128.0, 0.0, 128.0)
```

`turtle.`**`bgpic`**(*picname=None*)

매개변수
`picname` – a string, name of an image file (PNG, GIF, PGM, and PPM) or `"nopic"`, or `None`

배경 이미지를 설정하거나 현재 배경 이미지(backgroundimage)의 이름을 반환합니다. *picname*이 파일명이면, 해당 이미지를 배경으로 설정합니다. *picname*이 `"nopic"`이면, 배경 이미지가 있다면 삭제합니다. *picname*이 `None`이면, 현재 배경 이미지(backgroundimage)의 파일명을 반환합니다.

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

`turtle.`**`clear`**()

> **참고**
>
> 이 TurtleScreen 메서드는 `clearscreen`이라는 이름으로만 전역 함수로 사용할 수 있습니다. 전역 함수 `clear`는 Turtle 메서드 `clear`에서 파생된 다른 것입니다.

turtle.**clearscreen**()

TurtleScreen에서 모든 그림과 모든 거북이를 삭제합니다. 이제 비어있는 TurtleScreen을 초기 상태로 재설정합니다: 흰색 배경, 배경 이미지 없음, 이벤트 연결과 추적 없음.

turtle.**reset**()

> **참고**
>
> 이 TurtleScreen 메서드는 `resetscreen`이라는 이름으로만 전역 함수로 사용할 수 있습니다. 전역 함수 `reset`은 Turtle 메서드 `reset`에서 파생된 또 다른 함수입니다.

turtle.**resetscreen**()

Screen의 모든 거북이를 초기 상태로 재설정합니다.

turtle.**screensize**(*canvwidth=None, canvheight=None, bg=None*)

매개변수

- **canvwidth** – 양의 정수, 픽셀 단위의 새 캔버스 너비
- **canvheight** – 양의 정수, 픽셀 단위의 새 캔버스 높이
- **bg** – 색상 문자열(colorstring)이나 색상 튜플, 새 배경색

인자가 제공되지 않으면, 현재 (canvaswidth, canvasheight)를 반환합니다. 그렇지 않으면 거북이가 그리는 캔버스의 크기를 조정합니다. 그리는 창을 변경하지 마십시오. 캔버스의 숨겨진 부분을 보려면, 스크롤 막대를 사용하십시오. 이 메서드를 사용하면, 이전에 캔버스 외부에 있던 그림의 부분을 볼 수 있습니다.

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

예를 들어 잘못 탈출한 거북이를 찾기 위해 ;-)

turtle.**setworldcoordinates**(*llx, lly, urx, ury*)

매개변수

- **llx** – 숫자, 캔버스의 왼쪽 아래 모서리의 x-좌표
- **lly** – 숫자, 캔버스의 왼쪽 아래 모서리의 y-좌표
- **urx** – 숫자, 캔버스의 오른쪽 상단 모서리의 x-좌표
- **ury** – 숫자, 캔버스의 오른쪽 상단 모서리의 y-좌표

사용자 정의 좌표계를 설정하고 필요하면 "world" 모드로 전환합니다. 이것은 `screen.reset()`을 수행합니다. "world" 모드가 이미 활성화되었으면, 모든 그림은 새 좌표에 따라 다시 그려집니다.

주의: 사용자 정의 좌표계에서 각도가 왜곡되어 나타날 수 있습니다.

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)   # 정 팔각형
```

애니메이션 제어

turtle.**no_animation**()

Temporarily disable turtle animation. The code written inside the `no_animation` block will not be animated; once the code block is exited, the drawing will appear.

```
>>> with screen.no_animation():
...     for dist in range(2, 400, 2):
...         fd(dist)
...         rt(90)
```

Added in version 3.14.

turtle.**delay**(*delay=None*)

매개변수

delay – 양의 정수

그리기 지연(*delay*)을 밀리초 단위로 설정하거나 반환합니다. (이는 대략 두 개의 연속 캔버스 갱신 사이의 시간 간격입니다.) 그리기 지연이 길수록, 애니메이션이 느려집니다.

선택적 인자:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

turtle.**tracer**(*n=None, delay=None*)

매개변수

- **n** – 음이 아닌 정수
- **delay** – 음이 아닌 정수

거북이 애니메이션을 켜거나 끄고 그림 갱신 지연을 설정합니다. *n*이 제공되면, n 번째 정기 화면 갱신만 실제로 수행됩니다. (복잡한 그래픽의 그리기를 가속하는 데 사용할 수 있습니다.) 인자 없이 호출되면, 현재 저장된 n 값을 반환합니다. 두 번째 인자는 지연(delay) 값을 설정합니다 (`delay()`를 참조하십시오).

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

turtle.**update**()

TurtleScreen 갱신을 수행합니다. tracer가 꺼져있을 때 사용됩니다.

RawTurtle/Turtle 메서드 `speed()`도 참조하십시오.

화면 이벤트 사용하기

turtle.**listen**(*xdummy=None, ydummy=None*)

(키 이벤트를 수집하기 위해) TurtleScreen에 포커스를 설정합니다. `listen()`을 onclick 메서드에 전달할 수 있도록 더미 인자가 제공됩니다.

turtle.**onkey**(*fun, key*)

turtle.**onkeyrelease**(*fun*, *key*)

매개변수

- **fun** – 인자가 없는 함수나 None
- **key** – 문자열: 키 (예를 들어 "a") 또는 키-기호 (예를 들어 "space")

key의 키-릴리스 이벤트에 *fun*을 연결합니다. *fun*이 None이면, 이벤트 연결이 제거됩니다. 비고: 키 이벤트를 등록하려면, TurtleScreen에 포커스가 있어야 합니다. (메서드 *listen()*을 참조하십시오.)

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onkeypress**(*fun*, *key=None*)

매개변수

- **fun** – 인자가 없는 함수나 None
- **key** – 문자열: 키 (예를 들어 "a") 또는 키-기호 (예를 들어 "space")

key가 제공되면 key의 키-누르기 이벤트에 또는 key를 제공하지 않으면 임의의 키 누르기 이벤트에 *fun*을 연결합니다. 비고: 키 이벤트를 등록하려면, TurtleScreen에 포커스가 있어야 합니다. (메서드 *listen()*을 참조하십시오.)

```
>>> def f():
...     fd(50)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onclick**(*fun*, *btn=1*, *add=None*)
turtle.**onscreenclick**(*fun*, *btn=1*, *add=None*)

매개변수

- **fun** – 캔버스에서 클릭한 점의 좌표로 호출되는 두 개의 인자가 있는 함수
- **btn** – 마우스 버튼 수, 기본값은 1 (마우스 왼쪽 버튼)
- **add** – True 또는 False – True이면, 새 연결이 추가되고, 그렇지 않으면 이전 연결을 대체합니다

이 화면의 마우스-클릭 이벤트에 *fun*을 연결합니다. *fun*이 None이면, 기존 연결이 제거됩니다.

이름이 screen인 TurtleScreen 인스턴스와 이름이 turtle인 거북이 인스턴스의 예:

```
>>> screen.onclick(turtle.goto) # 이후 TurtleScreen을 클릭하면 거북이가 클릭한 지
점으로
>>>                             # 이동합니다.
>>> screen.onclick(None)        # 이벤트 연결을 다시 제거합니다
```

참고

이 TurtleScreen 메서드는 onscreenclick이라는 이름으로만 전역 함수로 사용할 수 있습니다. 전역 함수 onclick은 Turtle 메서드 onclick에서 파생된 또 다른 함수입니다.

turtle.**ontimer**(*fun*, *t=0*)

매개변수

- **fun** – 인자가 없는 함수
- **t** – 숫자 >= 0

t 밀리초 후에 *fun*을 호출하는 타이머를 설치합니다.

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()   ### 거북이가 주위를 행진하게 만듭니다
>>> running = False
```

turtle.**mainloop**()

turtle.**done**()

이벤트 루프를 시작합니다 - Tkinter의 mainloop 함수를 호출합니다. 터틀 그래픽 프로그램의 마지막 문장이어야 합니다. 터틀 그래픽을 대화식으로 사용하기 위해 -n 모드(서브 프로세스 없음)로 IDLE에서 스크립트를 실행할 때는 사용되지 않아야 합니다.

```
>>> screen.mainloop()
```

입력 메서드

turtle.**textinput**(*title*, *prompt*)

매개변수

- **title** – 문자열
- **prompt** – 문자열

문자열 입력을 위한 대화 상자 창을 띄웁니다. 매개변수 title은 대화 상자 창의 제목이고, prompt는 주로 어떤 정보를 입력해야 하는지 설명하는 텍스트입니다. 문자열 입력을 반환합니다. 대화 상자가 취소되면, `None`을 반환합니다.

```
>>> screen.textinput("NIM", "Name of first player:")
```

turtle.**numinput**(*title*, *prompt*, *default=None*, *minval=None*, *maxval=None*)

매개변수

- **title** – 문자열
- **prompt** – 문자열
- **default** – 숫자 (선택 사항)
- **minval** – 숫자 (선택 사항)
- **maxval** – 숫자 (선택 사항)

숫자 입력을 위한 대화 상자 창을 띄웁니다. title은 대화 창의 제목이고, prompt는 주로 어떤 숫자 정보를 입력해야 하는지 설명하는 텍스트입니다. default: 기본값, minval: 입력의 최솟값, maxval: 입력의 최댓값. 이것들이 주어지면 숫자 입력은 minval .. maxval 범위에 있어야 합니다. 그렇지 않으면, 힌트가 발행되고 수정을 위해 대화 상자가 열려 있습니다. 숫자 입력을 반환합니다. 대화 상자가 취소되면, `None`을 반환합니다.

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

설정과 특수 메서드

turtle.**mode**(*mode=None*)

매개변수
mode – 문자열 "standard", "logo" 또는 "world" 중 하나

거북이 모드("standard", "logo" 또는 "world")를 설정하고 재설정을 수행합니다. mode가 제공되지 않으면, 현재 모드가 반환됩니다.

"standard" 모드는 이전 *turtle*과 호환됩니다. "logo" 모드는 대부분의 로고 터틀 그래픽과 호환됩니다. "world" 모드는 사용자 정의 "세계 좌표"를 사용합니다. **주의**: 이 모드에서는 x/y 단위 비율이 1이 아니면 각도가 왜곡되어 나타납니다.

모드	초기 거북이 방향	양의 각도
"standard"	오른쪽 (동)	시계 반대 방향
"logo"	위쪽 (북)	시계 방향

```
>>> mode("logo")   # 거북이 방향을 북쪽으로 재설정합니다
>>> mode()
'logo'
```

turtle.**colormode**(*cmode=None*)

매개변수
cmode – 값 1.0이나 255중 하나

색상 모드(colormode)를 반환하거나 1.0이나 255로 설정합니다. 이후 색상 트리플의 *r*, *g*, *b* 값은 0..*cmode* 범위에 있어야 합니다.

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
     ...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

turtle.**getcanvas**()

이 TurtleScreen의 캔버스를 반환합니다. Tkinter Canvas로 작업하는 법을 알고 있는 내부자에게 유용합니다.

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

turtle.**getshapes**()

현재 사용 가능한 모든 거북이 모양의 이름 리스트를 반환합니다.

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

turtle.**register_shape**(*name, shape=None*)

turtle.**addshape**(*name*, *shape=None*)

There are four different ways to call this function:

(1) *name* is the name of an image file (PNG, GIF, PGM, and PPM) and *shape* is `None`: Install the corresponding image shape.

```
>>> screen.register_shape("turtle.gif")
```

> **참고**
>
> 거북이를 회전할 때 이미지 모양은 회전하지 않아서, 거북이 방향을 표시하지 않습니다!

(2) *name* is an arbitrary string and *shape* is the name of an image file (PNG, GIF, PGM, and PPM): Install the corresponding image shape.

```
>>> screen.register_shape("turtle", "turtle.gif")
```

> **참고**
>
> 거북이를 회전할 때 이미지 모양은 회전하지 않아서, 거북이 방향을 표시하지 않습니다!

(3) *name*은 임의의 문자열이고 *shape*은 좌표 쌍의 튜플입니다: 해당 다각형 모양을 설치합니다.

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

(4) *name*은 임의의 문자열이고 *shape*은 (복합) `Shape` 객체입니다: 해당 복합 모양을 설치합니다.

TurtleScreen의 모양 리스트(shapelist)에 거북이 모양을 추가합니다. `shape(shapename)` 명령을 실행하면 이처럼 등록된 모양만 사용할 수 있습니다.

버전 3.14에서 변경: Added support for PNG, PGM, and PPM image formats. Both a shape name and an image file name can be specified.

turtle.**turtles**()

화면에 있는 거북이의 리스트를 반환합니다.

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

turtle.**window_height**()

거북이 창의 높이를 반환합니다.

```
>>> screen.window_height()
480
```

turtle.**window_width**()

거북이 창의 너비를 반환합니다.

```
>>> screen.window_width()
640
```

TurtleScreen에서 상속되지 않은, Screen만의 메서드

turtle.**bye**()

터틀 그래픽 창을 닫습니다.

turtle.**exitonclick**()

bye() 메서드를 Screen에서의 마우스 클릭에 연결합니다.

구성 딕셔너리의 "using_IDLE" 값이 False(기본값)면, 에인 루프에 진입하기도 합니다. 비고: -n 스위치로 IDLE(서브 프로세스 없음)을 사용하면, 이 값은 turtle.cfg에서 True로 설정되어야 합니다. 이 경우 IDLE의 자체 메인 루프는 클라이언트 스크립트에서도 활성화됩니다.

turtle.**save**(*filename*, *overwrite=False*)

Save the current turtle drawing (and turtles) as a PostScript file.

매개변수

- **filename** – the path of the saved PostScript file
- **overwrite** – if False and there already exists a file with the given filename, then the function will raise a FileExistsError. If it is True, the file will be overwritten.

```
>>> screen.save("my_drawing.ps")
>>> screen.save("my_drawing.ps", overwrite=True)
```

Added in version 3.14.

turtle.**setup**(*width=_CFG['width']*, *height=_CFG['height']*, *startx=_CFG['leftright']*, *starty=_CFG['topbottom']*)

메인 창의 크기와 위치를 설정합니다. 인자의 기본값은 구성 딕셔너리에 저장되며 turtle.cfg 파일을 통해 변경할 수 있습니다.

매개변수

- **width** – 정수면 픽셀 단위의 크기, 실수면 화면의 비율; 기본값은 화면의 50% 입니다
- **height** – 정수면 픽셀 단위의 높이, 실수면 화면의 비율; 기본값은 화면의 75% 입니다
- **startx** – 양수이면 화면의 왼쪽 가장자리에서부터의 픽셀 단위의 시작 위치, 음수이면 오른쪽 가장자리에서부터, None이면 창을 가로로 가운데 정렬합니다
- **starty** – 양수이면 화면 위쪽 가장자리에서부터의 픽셀 단위의 시작 위치, 음수이면 아래쪽 가장자리에서부터, None이면 창을 세로로 가운데 정렬합니다

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>              # 창을 200x200 픽셀 크기로 화면 왼쪽 상단에 설정합니다
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>              # 창을 화면 너비의 75%, 화면 높이의 50% 크기로 가운데에 설정합니다
```

turtle.**title**(*titlestring*)

매개변수

titlestring – 터틀 그래픽 창의 제목 표시 줄에 표시되는 문자열

거북이 창의 제목을 *titlestring*으로 설정합니다.

```
>>> screen.title("Welcome to the turtle zoo!")
```

25.10.8 공개 클래스

class turtle.**RawTurtle**(*canvas*)

class turtle.**RawPen**(*canvas*)

매개변수

canvas – tkinter.Canvas, *ScrolledCanvas*나 *TurtleScreen*

거북이를 만듭니다. 거북이는 위에서 "Turtle/RawTurtle의 메서드"라고 언급한 모든 메서드를 갖습니다.

class turtle.**Turtle**

RawTurtle의 서브 클래스. 인터페이스는 같지만 처음 필요할 때 자동으로 생성된 기본 *Screen* 객체에 그립니다.

class turtle.**TurtleScreen**(*cv*)

매개변수

cv – tkinter.Canvas

위에서 설명한 *bgcolor()* 등과 같은 화면 지향 메서드를 제공합니다.

class turtle.**Screen**

TurtleScreen의 서브 클래스. *4개의 메서드가 추가되었습니다.*

class turtle.**ScrolledCanvas**(*master*)

매개변수

master – ScrolledCanvas, 즉 스크롤 막대가 추가된 Tkinter-Canvas를 담을 어떤 Tkinter 위젯

클래스 Screen에서 사용되며, 거북 놀이터로 ScrolledCanvas를 자동으로 제공합니다.

class turtle.**Shape**(*type_*, *data*)

매개변수

type_ – 문자열 "polygon", "image", "compound" 중 하나

모양을 모델링하는 데이터 구조. 쌍 (type_, data)는 다음 명세를 따라야 합니다:

*type_*	*data*
"polygon"	다각형 튜플, 즉 좌표 쌍의 튜플
"image"	이미지 (이 형식은 내부적으로만 사용됩니다!)
"compound"	None (복합 모양을 *addcomponent()* 메서드를 사용하여 구성해야 합니다)

addcomponent(*poly*, *fill*, *outline=None*)

매개변수

- **poly** – 다각형, 즉 숫자 쌍의 튜플
- **fill** – *poly*가 채워질 색상
- **outline** – poly 외곽선의 색상 (제공되면)

예:

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... 더 많은 구성 요소를 추가한 다음 register_shape()을 사용합니다
```

복합 모양을 참조하십시오.

class turtle.**Vec2D**(*x*, *y*)

2차원 벡터 클래스. 터틀 그래픽을 구현하기 위한 도우미 클래스로 사용됩니다. 터틀 그래픽 프로그램에도 유용할 수 있습니다. 튜플에서 파생되기 때문에, 벡터는 튜플입니다!

다음을 제공합니다 (*a*, *b*는 벡터, *k*는 번호):

- a + b 벡터 덧셈
- a - b 벡터 뺄셈

- `a * b` 내적(inner product)
- `k * a`와 `a * k` 스칼라를 사용한 곱셈
- `abs(a)` a의 절댓값
- `a.rotate(angle)` 회전

25.10.9 Explanation

A turtle object draws on a screen object, and there a number of key classes in the turtle object-oriented interface that can be used to create them and relate them to each other.

A *Turtle* instance will automatically create a *Screen* instance if one is not already present.

`Turtle` is a subclass of *RawTurtle*, which *doesn't* automatically create a drawing surface - a *canvas* will need to be provided or created for it. The *canvas* can be a `tkinter.Canvas`, *ScrolledCanvas* or *TurtleScreen*.

TurtleScreen is the basic drawing surface for a turtle. *Screen* is a subclass of `TurtleScreen`, and includes *some additional methods* for managing its appearance (including size and title) and behaviour. `TurtleScreen`'s constructor needs a `tkinter.Canvas` or a *ScrolledCanvas* as an argument.

거북이 그래픽의 함수적 인터페이스는 `Turtle`과 `TurtleScreen`/`Screen`의 다양한 메서드를 사용합니다. 내부적으로, Screen 객체는 Screen 메서드에서 파생된 함수가 호출될 때마다 자동으로 만들어집니다. 마찬가지로, Turtle 메서드에서 파생된 함수가 호출될 때마다 Turtle 객체가 자동으로 생성됩니다.

화면에 여러 거북을 사용하려면, 객체 지향 인터페이스를 사용해야 합니다.

25.10.10 도움말과 구성

도움말 사용법

Screen과 Turtle 클래스의 공개 메서드는 독스트링을 통해 광범위하게 설명됩니다. 파이썬 도움말 기능을 통해 온라인 도움말로 사용할 수 있습니다:

- IDLE을 사용할 때, 툴팁은 입력된 함수/메서드 호출의 서명과 독스트링의 첫 줄을 보여줍니다.
- 메서드나 함수에 대해 *help()*를 호출하면 독스트링을 표시합니다:

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.


    >>> screen.bgcolor("orange")
    >>> screen.bgcolor()
    "orange"
    >>> screen.bgcolor(0.5,0,0.5)
    >>> screen.bgcolor()
    "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    No argument

    >>> turtle.penup()
```

- 메서드에서 파생된 함수의 독스트링은 다음과 같이 수정된 형식을 갖습니다:

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    Example::

      >>> bgcolor("orange")
      >>> bgcolor()
      "orange"
      >>> bgcolor(0.5,0,0.5)
      >>> bgcolor()
      "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

이러한 수정된 독스트링은 임포트 시점에 메서드에서 파생된 함수 정의와 함께 자동으로 만들어집니다.

독스트링을 다른 언어로 번역

키가 메서드 이름이고 값이 Screen과 Turtle 클래스의 공개 메서드의 독스트링인 딕셔너리를 만드는 유틸리티가 있습니다.

turtle.**write_docstringdict**(*filename='turtle_docstringdict'*)

매개변수

filename – 파일명으로 사용되는 문자열

주어진 파일명의 파이썬 스크립트에 독스트링-딕셔너리를 만들고 씁니다. 이 함수는 명시적으로 호출해야 합니다(터틀 그래픽 클래스에서는 사용되지 않습니다). 독스트링 딕셔너리는 파이썬 스크립트 *filename*.py에 기록됩니다. 독스트링을 다른 언어로 번역하기 위한 템플릿으로 사용하려는 목적입니다.

여러분(또는 여러분의 학생)이 모국어로 된 온라인 도움말과 함께 *turtle*을 사용하려면, 독스트링을 번역하고 결과 파일을 예를 들어 turtle_docstringdict_german.py와 같은 파일로 저장해야 합니다.

여러분의 turtle.cfg 파일에 적절한 항목이 있으면 임포트 시점에 이 딕셔너리를 읽고 원래 영어 독스트링을 대체합니다.

이 글을 쓰는 시점에는 독일어와 이탈리아어로 된 독스트링 딕셔너리가 있습니다. (glingl@aon.at 에 요청하십시오.)

Screen과 Turtle을 구성하는 방법

내장된 기본 구성은 최상의 호환성을 유지하기 위해 기존 turtle 모듈의 외관과 동작을 모방합니다.

이 모듈의 기능을 더 잘 반영하거나 예를 들어 교실에서 사용하기 위해 필요에 더 잘 맞는 다른 구성을 사용하려면, 임포트 시점에 읽는 구성 파일 `turtle.cfg`를 준비하고 구성을 그 설정에 따라 수정할 수 있습니다.

내장 구성은 다음 `turtle.cfg`에 해당합니다:

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

선택된 항목에 대한 간단한 설명:

- 처음 네 줄은 *Screen.setup* 메서드의 인자에 해당합니다.
- 줄 5와 6은 *Screen.screensize* 메서드의 인자에 해당합니다.
- *shape*은 내장 도형 중 하나일 수 있습니다, 예를 들어 arrow, turtle 등. 자세한 정보는 `help(shape)` 을 시도해 보십시오.
- 아무런 채우기 색상(fill color)도 사용하지 않으려면 (즉, 거북이를 투명하게 만들려면), `fillcolor = ""`라고 작성해야 합니다 (그러나 비어 있지 않은 모든 문자열은 cfg 파일에서 따옴표가 없어야 합니다).
- 거북이의 상태를 반영하려면, `resizemode = auto`를 사용해야 합니다.
- 예를 들어 `language = italian`을 설정하면 임포트 시점에 독스트링 딕셔너리 `turtle_docstringdict_italian.py`가 로드됩니다 (임포트 경로에 있다면, 예를 들어 *turtle*과 같은 디렉터리에).
- *exampleturtle*과 *examplescreen* 항목은 독스트링에서 등장하는 이러한 객체들의 이름을 정의합니다. 메서드 독스트링을 함수 독스트링으로 변환하면 이러한 이름이 독스트링에서 삭제됩니다.
- *using_IDLE*: IDLE과 그것의 `-n` 스위치("서브 프로세스 없음")를 정기적으로 사용하면 이 값을 `True`로 설정하십시오. 이것은 *exitonclick()*이 메인 루프에 들어가지 못하게 합니다.

*turtle*이 저장된 디렉터리에 `turtle.cfg` 파일이 있고 현재 작업 디렉터리에도 추가 파일이 있을 수 있습니다. 후자가 첫 번째 설정을 재정의합니다.

`Lib/turtledemo` 디렉터리는 `turtle.cfg` 파일을 포함합니다. 예제로 공부하고 데모를 실행할 때 그 효과를 볼 수 있습니다 (바람직하게는 데모 뷰어 내에서는 아닙니다).

25.10.11 turtledemo — 데모 스크립트

turtledemo 패키지에는 데모 스크립트 집합이 포함되어 있습니다. 이 스크립트는 다음과 같이 제공된 데모 뷰어를 사용하여 실행하고 볼 수 있습니다:

```
python -m turtledemo
```

또는, 데모 스크립트를 개별적으로 실행할 수 있습니다. 예를 들면,

```
python -m turtledemo.bytedesign
```

turtledemo 패키지 디렉터리는 다음과 같은 것들을 포함합니다:

- 스크립트의 소스 코드를 보고 동시에 실행하는 데 사용할 수 있는 데모 뷰어 `__main__.py`.
- *turtle* 모듈의 다른 기능을 보여주는 여러 스크립트. 예제는 Examples 메뉴를 통해 액세스할 수 있습니다. 독립형으로 실행할 수도 있습니다.
- 구성 파일을 작성하고 사용하는 방법의 예인 `turtle.cfg` 파일.

데모 스크립트는 다음과 같습니다:

이름	설명	기능
bytedesign	복잡한 클래식 터틀 그래픽 패턴	*tracer()*, delay, *update()*
chaos	베르홀스트(Verhulst) 동역학의 그래프를 그립니다, 컴퓨터의 계산이 때때로 상식적인 기대에 반하는 결과를 생 할 수 있음을 보여줍니다	세계 좌표
clock	컴퓨터의 시간을 보여주는 아날로그 시계	시곗바늘 거북이, ontimer
colormixer	r, g, b 실험	*ondrag()*
forest	3개의 너비 우선 나무	무작위화
fractalcurves	힐버트(Hilbert)와 코흐(Koch) 곡선	재귀
lindenmayer	민족 수학 (인도 콜람)	L-System
minimal_hanoi	하노이의 탑	하노이 디스크로 쓰는 직사각형 거북 (모양, 모양 크기)
nim	3개의 막대 더미로 컴퓨터와 고전적인 님 게임을 합니다.	님 막대로 쓰는 거북, 이벤트 구동 (마우스, 키보드)
paint	극도로 단순한 그리기 프로그램	*onclick()*
peace	기초	거북이: 외관과 애니메이션
penrose	연과 다트가 있는 비주기적 타일링	*stamp()*
planet_and_moon	중력 시스템 시뮬레이션	복합 모양, *Vec2D*
rosette	터틀 그래픽에 관한 위키피디아 기사의 패턴	*clone()*, *undo()*
round_dance	반대 방향으로 쌍으로 회전하는 춤추는 거북이	복합 모양, 모양 크기 복제, tilt, get_shapepoly, update
sorting_animate	여러 정렬 방법의 시각적 데모	간단한 정렬, 무작위화
tree	(그래픽) 너비 우선 나무 (제너레이터 사용)	*clone()*
two_canvases	간단한 디자인	두 캔버스의 거북이
yinyang	또 하나의 기초 예제	*circle()*

즐기세요!

25.10.12 파이썬 2.6 이후의 변화

- 메서드 *Turtle.tracer*, *Turtle.window_width* 및 *Turtle.window_height*가 제거되었습니다. 이러한 이름과 기능을 가진 메서드는 이제 *Screen*의 메서드로만 사용 가능합니다. 이들에서 파생된 함수는 계속 사용할 수 있습니다. (실제로 파이썬 2.6에서 이 메서드는 해당 *TurtleScreen*/*Screen* 메서드의 복제일 뿐입니다.)
- 메서드 `Turtle.fill()`이 제거되었습니다. *begin_fill()*과 *end_fill()*의 동작이 약간 변경되었습니다: 이제 모든 채우기 프로세스를 `end_fill()` 호출로 완료해야 합니다.
- 메서드 *Turtle.filling*이 추가되었습니다. 불리언 값을 반환합니다: 채우기 프로세스가 진행 중이면 `True`를, 그렇지 않으면 `False`를 반환합니다. 이 동작은 파이썬 2.6에서 인자가 없는 `fill()` 호출에 해당합니다.

25.10.13 파이썬 3.0 이후의 변화

- *Turtle* 메서드 *shearfactor()*, *shapetransform()* 및 *get_shapepoly()*가 추가되었습니다. 따라서 이제 거북이 모양을 변환하기 위해 전체 범위의 일반 선형 변환을 사용할 수 있습니다. *tiltangle()*의 기능이 향상되었습니다: 이제 틸트 각도를 가져오거나 설정하는 데 사용할 수 있습니다.
- *Screen* 메서드 *onkeypress()*는 *onkey()*의 보완으로 추가되었습니다. 후자는 키 릴리스 이벤트에 액션을 연결하므로, 이를 위한 별칭 *onkeyrelease()*도 추가되었습니다.
- The method *Screen.mainloop* has been added, so there is no longer a need to use the standalone *mainloop()* function when working with *Screen* and *Turtle* objects.
- 두 가지 입력 메서드가 추가되었습니다: *Screen.textinput*과 *Screen.numinput*. 입력 대화 상자를 띄우고 각각 문자열과 숫자를 반환합니다.

CHAPTER 26

개발 도구

이 장에서 설명하는 모듈은 소프트웨어를 작성하는 것을 돕습니다. 예를 들어, *pydoc* 모듈은 모듈을 가져와서 모듈의 내용을 기반으로 설명서를 만듭니다. *doctest* 와 *unittest* 모듈에는 예상 출력이 만들어지는지 코드를 자동으로 실행하고 확인하는 단위 테스트를 작성하기 위한 프레임워크가 포함되어 있습니다.

이 장에서 설명하는 모듈 목록은 다음과 같습니다:

26.1 `typing` — Support for type hints

Added in version 3.5.

소스 코드: Lib/typing.py

> **참고**
>
> The Python runtime does not enforce function and variable type annotations. They can be used by third party tools such as *type checkers*, IDEs, linters, etc.

This module provides runtime support for type hints.

Consider the function below:

```
def surface_area_of_cube(edge_length: float) -> str:
    return f"The surface area of the cube is {6 * edge_length ** 2}."
```

The function `surface_area_of_cube` takes an argument expected to be an instance of *float*, as indicated by the *type hint* `edge_length: float`. The function is expected to return an instance of *str*, as indicated by the `-> str` hint.

While type hints can be simple classes like *float* or *str*, they can also be more complex. The *typing* module provides a vocabulary of more advanced type hints.

New features are frequently added to the `typing` module. The typing_extensions package provides backports of these new features to older versions of Python.

더 보기

Typing cheat sheet
A quick overview of type hints (hosted at the mypy docs)

Type System Reference section of the mypy docs
The Python typing system is standardised via PEPs, so this reference should broadly apply to most Python type checkers. (Some parts may still be specific to mypy.)

Static Typing with Python
Type-checker-agnostic documentation written by the community detailing type system features, useful typing related tools and typing best practices.

26.1.1 Specification for the Python Type System

The canonical, up-to-date specification of the Python type system can be found at Specification for the Python type system.

26.1.2 형 에일리어스

A type alias is defined using the `type` statement, which creates an instance of `TypeAliasType`. In this example, `Vector` and `list[float]` will be treated equivalently by static type checkers:

```
type Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# passes type checking; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

형 에일리어스는 복잡한 형 서명을 단순화하는 데 유용합니다. 예를 들면:

```
from collections.abc import Sequence

type ConnectionOptions = dict[str, str]
type Address = tuple[str, int]
type Server = tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...

# The static type checker will treat the previous type signature as
# being exactly equivalent to this one.
def broadcast_message(
    message: str,
    servers: Sequence[tuple[tuple[str, int], dict[str, str]]]
) -> None:
    ...
```

The `type` statement is new in Python 3.12. For backwards compatibility, type aliases can also be created through simple assignment:

```
Vector = list[float]
```

Or marked with `TypeAlias` to make it explicit that this is a type alias, not a normal variable assignment:

```
from typing import TypeAlias

Vector: TypeAlias = list[float]
```

26.1.3 NewType

Use the *NewType* helper to create distinct types:

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

정적 형 검사기는 새 형을 원래 형의 서브 클래스인 것처럼 다룹니다. 논리 에러를 잡는 데 유용합니다:

```
def get_user_name(user_id: UserId) -> str:
    ...

# passes type checking
user_a = get_user_name(UserId(42351))

# fails type checking; an int is not a UserId
user_b = get_user_name(-1)
```

`UserId` 형의 변수에 대해 모든 `int` 연산을 여전히 수행할 수 있지만, 결과는 항상 `int` 형이 됩니다. 이것은 `int` 가 기대되는 모든 곳에 `UserId`를 전달할 수 있지만, 잘못된 방식으로 의도하지 않게 `UserId`를 만들지 않도록 합니다:

```
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

Note that these checks are enforced only by the static type checker. At runtime, the statement `Derived = NewType('Derived', Base)` will make `Derived` a callable that immediately returns whatever parameter you pass it. That means the expression `Derived(some_value)` does not create a new class or introduce much overhead beyond that of a regular function call.

더욱 정확하게, 표현식 `some_value is Derived(some_value)` 는 실행 시간에 항상 참입니다.

It is invalid to create a subtype of `Derived`:

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not pass type checking
class AdminUserId(UserId): pass
```

However, it is possible to create a *NewType* based on a 'derived' `NewType`:

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

그리고 `ProUserId`에 대한 형 검사는 예상대로 작동합니다.

자세한 내용은 **PEP 484**를 참조하십시오.

> **참고**
>
> Recall that the use of a type alias declares two types to be *equivalent* to one another. Doing `type Alias = Original` will make the static type checker treat `Alias` as being *exactly equivalent* to `Original` in all cases. This is useful when you want to simplify complex type signatures.
>
> 반면에, `NewType`은 한 형을 다른 형의 서브 형으로 선언합니다. `Derived = NewType('Derived', Original)`은 정적 형 검사기가 `Derived`를 `Original`의 서브 클래스로 취급하게 합니다. 이는 `Original` 형의 값이 `Derived` 형의 값이 예상되는 위치에서 사용될 수 없음을 의미합니다. 실행 시간 비용을 최소화하면서 논리 에러를 방지하려는 경우에 유용합니다.

Added in version 3.5.2.

버전 3.10에서 변경: `NewType` is now a class rather than a function. As a result, there is some additional runtime cost when calling `NewType` over a regular function.

버전 3.11에서 변경: The performance of calling `NewType` has been restored to its level in Python 3.9.

26.1.4 Annotating callable objects

Functions – or other *callable* objects – can be annotated using *`collections.abc.Callable`* or deprecated *`typing.Callable`*. `Callable[[int], str]` signifies a function that takes a single parameter of type *`int`* and returns a *`str`*.

For example:

```
from collections.abc import Callable, Awaitable

def feeder(get_next_item: Callable[[], str]) -> None:
    ...  # Body

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    ...  # Body

async def on_update(value: str) -> None:
    ...  # Body

callback: Callable[[str], Awaitable[None]] = on_update
```

The subscription syntax must always be used with exactly two values: the argument list and the return type. The argument list must be a list of types, a *`ParamSpec`*, *`Concatenate`*, or an ellipsis `...`. The return type must be a single type.

If a literal ellipsis `...` is given as the argument list, it indicates that a callable with any arbitrary parameter list would be acceptable:

```
def concat(x: str, y: str) -> str:
    return x + y

x: Callable[..., str]
x = str     # OK
x = concat  # Also OK
```

`Callable` cannot express complex signatures such as functions that take a variadic number of arguments, *overloaded functions*, or functions that have keyword-only parameters. However, these signatures can be expressed by defining a *`Protocol`* class with a `__call__()` method:

```
from collections.abc import Iterable
from typing import Protocol
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
class Combiner(Protocol):
    def __call__(self, *vals: bytes, maxlen: int | None = None) -> list[bytes]: ...

def batch_proc(data: Iterable[bytes], cb_results: Combiner) -> bytes:
    for item in data:
        ...

def good_cb(*vals: bytes, maxlen: int | None = None) -> list[bytes]:
    ...
def bad_cb(*vals: bytes, maxitems: int | None) -> list[bytes]:
    ...

batch_proc([], good_cb)  # OK
batch_proc([], bad_cb)   # Error! Argument 2 has incompatible type because of
                         # different name and kind in the callback
```

Callables which take other callables as arguments may indicate that their parameter types are dependent on each other using *ParamSpec*. Additionally, if that callable adds or removes arguments from other callables, the *Concatenate* operator may be used. They take the form `Callable[ParamSpecVariable, ReturnType]` and `Callable[Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable], ReturnType]` respectively.

버전 3.10에서 변경: `Callable` now supports *ParamSpec* and *Concatenate*. See **PEP 612** for more details.

더 보기

The documentation for *ParamSpec* and *Concatenate* provides examples of usage in `Callable`.

26.1.5 제네릭

Since type information about objects kept in containers cannot be statically inferred in a generic way, many container classes in the standard library support subscription to denote the expected types of container elements.

```
from collections.abc import Mapping, Sequence

class Employee: ...

# Sequence[Employee] indicates that all elements in the sequence
# must be instances of "Employee".
# Mapping[str, str] indicates that all keys and all values in the mapping
# must be strings.
def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

Generic functions and classes can be parameterized by using type parameter syntax:

```
from collections.abc import Sequence

def first[T](l: Sequence[T]) -> T:  # Function is generic over the TypeVar "T"
    return l[0]
```

Or by using the *TypeVar* factory directly:

```
from collections.abc import Sequence
from typing import TypeVar
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
U = TypeVar('U')                  # Declare type variable "U"

def second(l: Sequence[U]) -> U:  # Function is generic over the TypeVar "U"
    return l[1]
```

버전 3.12에서 변경: Syntactic support for generics is new in Python 3.12.

26.1.6 Annotating tuples

For most containers in Python, the typing system assumes that all elements in the container will be of the same type. For example:

```
from collections.abc import Mapping

# Type checker will infer that all elements in ``x`` are meant to be ints
x: list[int] = []

# Type checker error: ``list`` only accepts a single type argument:
y: list[int, str] = [1, 'foo']

# Type checker will infer that all keys in ``z`` are meant to be strings,
# and that all values in ``z`` are meant to be either strings or ints
z: Mapping[str, str | int] = {}
```

`list` only accepts one type argument, so a type checker would emit an error on the `y` assignment above. Similarly, `Mapping` only accepts two type arguments: the first indicates the type of the keys, and the second indicates the type of the values.

Unlike most other Python containers, however, it is common in idiomatic Python code for tuples to have elements which are not all of the same type. For this reason, tuples are special-cased in Python' s typing system. `tuple` accepts *any number* of type arguments:

```
# OK: ``x`` is assigned to a tuple of length 1 where the sole element is an int
x: tuple[int] = (5,)

# OK: ``y`` is assigned to a tuple of length 2;
# element 1 is an int, element 2 is a str
y: tuple[int, str] = (5, "foo")

# Error: the type annotation indicates a tuple of length 1,
# but ``z`` has been assigned to a tuple of length 3
z: tuple[int] = (1, 2, 3)
```

To denote a tuple which could be of *any* length, and in which all elements are of the same type `T`, use the literal ellipsis `...`: `tuple[T, ...]`. To denote an empty tuple, use `tuple[()]`. Using plain `tuple` as an annotation is equivalent to using `tuple[Any, ...]`:

```
x: tuple[int, ...] = (1, 2)
# These reassignments are OK: ``tuple[int, ...]`` indicates x can be of any length
x = (1, 2, 3)
x = ()
# This reassignment is an error: all elements in ``x`` must be ints
x = ("foo", "bar")

# ``y`` can only ever be assigned to an empty tuple
y: tuple[()] = ()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
z: tuple = ("foo", "bar")
# These reassignments are OK: plain ``tuple`` is equivalent to ``tuple[Any, ...]``
z = (1, 2, 3)
z = ()
```

26.1.7 The type of class objects

A variable annotated with `C` may accept a value of type `C`. In contrast, a variable annotated with `type[C]` (or deprecated *`typing.Type[C]`*) may accept values that are classes themselves – specifically, it will accept the *class object* of `C`. For example:

```
a = 3         # Has type ``int``
b = int       # Has type ``type[int]``
c = type(a)   # Also has type ``type[int]``
```

Note that `type[C]` is covariant:

```
class User: ...
class ProUser(User): ...
class TeamUser(User): ...

def make_new_user(user_class: type[User]) -> User:
    # ...
    return user_class()

make_new_user(User)       # OK
make_new_user(ProUser)    # Also OK: ``type[ProUser]`` is a subtype of␣
↪``type[User]``
make_new_user(TeamUser)   # Still fine
make_new_user(User())     # Error: expected ``type[User]`` but got ``User``
make_new_user(int)        # Error: ``type[int]`` is not a subtype of ``type[User]``
```

The only legal parameters for *`type`* are classes, *`Any`*, *type variables*, and unions of any of these types. For example:

```
def new_non_team_user(user_class: type[BasicUser | ProUser]): ...

new_non_team_user(BasicUser)   # OK
new_non_team_user(ProUser)     # OK
new_non_team_user(TeamUser)    # Error: ``type[TeamUser]`` is not a subtype
                               # of ``type[BasicUser | ProUser]``
new_non_team_user(User)        # Also an error
```

`type[Any]` is equivalent to *`type`*, which is the root of Python' s metaclass hierarchy.

26.1.8 Annotating generators and coroutines

A generator can be annotated using the generic type *`Generator[YieldType, SendType, ReturnType]`*. For example:

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

Note that unlike many other generic classes in the standard library, the `SendType` of *`Generator`* behaves contravariantly, not covariantly or invariantly.

The `SendType` and `ReturnType` parameters default to `None`:

```
def infinite_stream(start: int) -> Generator[int]:
    while True:
        yield start
        start += 1
```

It is also possible to set these types explicitly:

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

Simple generators that only ever yield values can also be annotated as having a return type of either *`Iterable[YieldType]`* or *`Iterator[YieldType]`*:

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

Async generators are handled in a similar fashion, but don't expect a `ReturnType` type argument (*`AsyncGenerator[YieldType, SendType]`*). The `SendType` argument defaults to `None`, so the following definitions are equivalent:

```
async def infinite_stream(start: int) -> AsyncGenerator[int]:
    while True:
        yield start
        start = await increment(start)

async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

As in the synchronous case, *`AsyncIterable[YieldType]`* and *`AsyncIterator[YieldType]`* are available as well:

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

Coroutines can be annotated using *`Coroutine[YieldType, SendType, ReturnType]`*. Generic arguments correspond to those of *`Generator`*, for example:

```
from collections.abc import Coroutine
c: Coroutine[list[str], str, int]  # Some coroutine defined elsewhere
x = c.send('hi')                   # Inferred type of 'x' is list[str]
async def bar() -> None:
    y = await c                    # Inferred type of 'y' is int
```

26.1.9 사용자 정의 제네릭 형

사용자 정의 클래스는 제네릭 클래스로 정의 할 수 있습니다.

```
from logging import Logger

class LoggedVar[T]:
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

This syntax indicates that the class `LoggedVar` is parameterised around a single *type variable* `T` . This also makes `T` valid as a type within the class body.

Generic classes implicitly inherit from *`Generic`*. For compatibility with Python 3.11 and lower, it is also possible to inherit explicitly from *`Generic`* to indicate a generic class:

```
from typing import TypeVar, Generic

T = TypeVar('T')

class LoggedVar(Generic[T]):
    ...
```

Generic classes have `__class_getitem__()` methods, meaning they can be parameterised at runtime (e.g. `LoggedVar[int]` below):

```
from collections.abc import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

A generic type can have any number of type variables. All varieties of *`TypeVar`* are permissible as parameters for a generic type:

```
from typing import TypeVar, Generic, Sequence

class WeirdTrio[T, B: Sequence[bytes], S: (int, str)]:
    ...

OldT = TypeVar('OldT', contravariant=True)
OldB = TypeVar('OldB', bound=Sequence[bytes], covariant=True)
OldS = TypeVar('OldS', int, str)

class OldWeirdTrio(Generic[OldT, OldB, OldS]):
    ...
```

*`Generic`*에 대한 각 형 변수 인자는 달라야 합니다. 그래서 이것은 잘못되었습니다:

```
from typing import TypeVar, Generic
...

class Pair[M, M]:  # SyntaxError
    ...

T = TypeVar('T')

class Pair(Generic[T, T]):   # INVALID
    ...
```

Generic classes can also inherit from other classes:

```
from collections.abc import Sized

class LinkedList[T](Sized):
    ...
```

When inheriting from generic classes, some type parameters could be fixed:

```
from collections.abc import Mapping

class MyDict[T](Mapping[str, T]):
    ...
```

이 경우 `MyDict`는 단일 매개 변수 `T`를 갖습니다.

Using a generic class without specifying type parameters assumes *Any* for each position. In the following example, `MyIterable` is not generic but implicitly inherits from `Iterable[Any]`:

```
from collections.abc import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
    ...
```

User-defined generic type aliases are also supported. Examples:

```
from collections.abc import Iterable

type Response[S] = Iterable[S] | int

# Return type here is same as Iterable[str] | int
def response(query: str) -> Response[str]:
    ...

type Vec[T] = Iterable[tuple[T, T]]

def inproduct[T: (int, float, complex)](v: Vec[T]) -> T: # Same as␣
↪Iterable[tuple[T, T]]
    return sum(x*y for x, y in v)
```

For backward compatibility, generic type aliases can also be created through a simple assignment:

```
from collections.abc import Iterable
from typing import TypeVar

S = TypeVar("S")
Response = Iterable[S] | int
```

버전 3.7에서 변경: *Generic*에는 더는 사용자 정의 메타 클래스가 없습니다.

버전 3.12에서 변경: Syntactic support for generics and type aliases is new in version 3.12. Previously, generic classes had to explicitly inherit from *Generic* or contain a type variable in one of their bases.

User-defined generics for parameter expressions are also supported via parameter specification variables in the form `[**P]`. The behavior is consistent with type variables' described above as parameter specification variables are treated by the `typing` module as a specialized type variable. The one exception to this is that a list of types can be used to substitute a *ParamSpec*:

```
>>> class Z[T, **P]: ...  # T is a TypeVar; P is a ParamSpec
...
>>> Z[int, [dict, float]]
__main__.Z[int, [dict, float]]
```

Classes generic over a *ParamSpec* can also be created using explicit inheritance from *Generic*. In this case, `**` is not used:

```
from typing import ParamSpec, Generic

P = ParamSpec('P')

class Z(Generic[P]):
    ...
```

Another difference between *TypeVar* and *ParamSpec* is that a generic with only one parameter specification variable will accept parameter lists in the forms `X[[Type1, Type2, ...]]` and also `X[Type1, Type2, ...]` for aesthetic reasons. Internally, the latter is converted to the former, so the following are equivalent:

```
>>> class X[**P]: ...
...
>>> X[int, str]
__main__.X[[int, str]]
>>> X[[int, str]]
__main__.X[[int, str]]
```

Note that generics with *ParamSpec* may not have correct `__parameters__` after substitution in some cases because they are intended primarily for static type checking.

버전 3.10에서 변경: *Generic* can now be parameterized over parameter expressions. See *ParamSpec* and **PEP 612** for more details.

A user-defined generic class can have ABCs as base classes without a metaclass conflict. Generic metaclasses are not supported. The outcome of parameterizing generics is cached, and most types in the `typing` module are *hashable* and comparable for equality.

26.1.10 Any 형

특수한 종류의 형은 *Any*입니다. 정적 형 검사기는 모든 형을 *Any*와 호환되는 것으로, *Any*를 모든 형과 호환되는 것으로 취급합니다.

이것은 *Any* 형의 값에 대해 어떤 연산이나 메서드 호출을 수행하고, 그것을 임의의 변수에 대입할 수 있다는 것을 의미합니다:

```
from typing import Any

a: Any = None
a = []          # OK
a = 2           # OK

s: str = ''
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
s = a            # OK

def foo(item: Any) -> int:
    # Passes type checking; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

Notice that no type checking is performed when assigning a value of type *Any* to a more precise type. For example, the static type checker did not report an error when assigning `a` to `s` even though `s` was declared to be of type *str* and receives an *int* value at runtime!

또한, 반환형이나 매개 변수 형이 없는 모든 함수는 묵시적으로 *Any* 기본값을 사용합니다:

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

이 동작은 여러분이 동적으로 형이 지정되는 코드와 정적으로 형이 지정되는 코드를 혼합해야 할 때 *Any*를 탈출구로 사용할 수 있도록 합니다.

*Any*의 동작과 *object*의 동작을 대조하십시오. *Any*와 유사하게, 모든 형은 *object*의 서브 형입니다. 그러나, *Any*와는 달리, 그 반대는 사실이 아닙니다: *object*는 다른 모든 형의 서브 형이 아닙니다.

이것은 값의 형이 *object*일 때, 형 검사기가 그것에 대한 거의 모든 연산을 거부하고, 그것을 더 특수한 형의 변수에 대입(또는 그것을 반환 값으로 사용)하는 것이 형 에러임을 의미합니다. 예를 들면:

```
def hash_a(item: object) -> int:
    # Fails type checking; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Passes type checking
    item.magic()
    ...

# Passes type checking, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")

# Passes type checking, since Any is compatible with all types
hash_b(42)
hash_b("foo")
```

값이 형 안전한 방식으로 모든 형이 될 수 있음을 표시하려면 *object*를 사용하십시오. 값이 동적으로 형이 지정됨을 표시하려면 *Any*를 사용하십시오.

26.1.11 명목적 대 구조적 서브 타이핑

Initially **PEP 484** defined the Python static type system as using *nominal subtyping*. This means that a class `A` is allowed where a class `B` is expected if and only if `A` is a subclass of `B`.

이 요구 사항은 이전에 *Iterable*과 같은 추상 베이스 클래스에도 적용되었습니다. 이 접근 방식의 문제점은 이것을 지원하려면 클래스를 명시적으로 표시해야만 한다는 점입니다. 이는 파이썬답지 않고 관용적인 동적으로 형이 지정된 파이썬 코드에서 일반적으로 수행하는 것과는 다릅니다. 예를 들어, 이것은 **PEP 484**를 만족합니다:

```
from collections.abc import Sized, Iterable, Iterator

class Bucket(Sized, Iterable[int]):
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...
```

PEP 544는 사용자가 클래스 정의에서 명시적인 베이스 클래스 없이 위의 코드를 작성할 수 있게 함으로써 이 문제를 풀도록 합니다. 정적 형 검사기가 `Bucket`을 `Sized`와 `Iterable[int]`의 서브 형으로 묵시적으로 취급하도록 합니다. 이것은 구조적 서브 타이핑 (*structural subtyping*) (또는 정적 덕 타이핑) 으로 알려져 있습니다:

```
from collections.abc import Iterator, Iterable

class Bucket:  # Note: no base classes
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...

def collect(items: Iterable[int]) -> int: ...
result = collect(Bucket())  # Passes type check
```

또한, 특별한 클래스 *Protocol*을 서브 클래싱 함으로써, 사용자는 새로운 사용자 정의 프로토콜을 정의하여 구조적 서브 타이핑을 완전히 누릴 수 있습니다 (아래 예를 참조하십시오).

26.1.12 모듈 내용

The `typing` module defines the following classes, functions and decorators.

특수 타이핑 프리미티브

특수형

These can be used as types in annotations. They do not support subscription using `[]`.

`typing.`**`Any`**

제한되지 않는 형을 나타내는 특수형.

- 모든 형은 *Any*와 호환됩니다.
- *Any*는 모든 형과 호환됩니다.

버전 3.11에서 변경: *Any* can now be used as a base class. This can be useful for avoiding type checker errors with classes that can duck type anywhere or are highly dynamic.

`typing.`**`AnyStr`**

A *constrained type variable*.

Definition:

```
AnyStr = TypeVar('AnyStr', str, bytes)
```

AnyStr is meant to be used for functions that may accept *str* or *bytes* arguments but cannot allow the two to mix.

예를 들면:

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat("foo", "bar")     # OK, output has type 'str'
concat(b"foo", b"bar")   # OK, output has type 'bytes'
concat("foo", b"bar")    # Error, cannot mix str and bytes
```

Note that, despite its name, AnyStr has nothing to do with the *Any* type, nor does it mean "any string". In particular, AnyStr and str | bytes are different from each other and have different use cases:

```
# Invalid use of AnyStr:
# The type variable is used only once in the function signature,
# so cannot be "solved" by the type checker
def greet_bad(cond: bool) -> AnyStr:
    return "hi there!" if cond else b"greetings!"

# The better way of annotating this function:
def greet_proper(cond: bool) -> str | bytes:
    return "hi there!" if cond else b"greetings!"
```

Deprecated since version 3.13, will be removed in version 3.18: Deprecated in favor of the new type parameter syntax. Use class A[T: (str, bytes)]: ... instead of importing AnyStr. See **PEP 695** for more details.

In Python 3.16, AnyStr will be removed from typing.__all__, and deprecation warnings will be emitted at runtime when it is accessed or imported from typing. AnyStr will be removed from typing in Python 3.18.

typing.**LiteralString**

Special type that includes only literal strings.

Any string literal is compatible with LiteralString, as is another LiteralString. However, an object typed as just str is not. A string created by composing LiteralString-typed objects is also acceptable as a LiteralString.

Example:

```
def run_query(sql: LiteralString) -> None:
    ...

def caller(arbitrary_string: str, literal_string: LiteralString) -> None:
    run_query("SELECT * FROM students")  # OK
    run_query(literal_string)  # OK
    run_query("SELECT * FROM " + literal_string)  # OK
    run_query(arbitrary_string)  # type checker error
    run_query(  # type checker error
        f"SELECT * FROM students WHERE name = {arbitrary_string}"
    )
```

LiteralString is useful for sensitive APIs where arbitrary user-generated strings could generate problems. For example, the two cases above that generate type checker errors could be vulnerable to an SQL injection attack.

See **PEP 675** for more details.

Added in version 3.11.

typing.**Never**

typing.**NoReturn**

`Never` and `NoReturn` represent the bottom type, a type that has no members.

They can be used to indicate that a function never returns, such as *sys.exit()*:

```
from typing import Never  # or NoReturn

def stop() -> Never:
    raise RuntimeError('no way')
```

Or to define a function that should never be called, as there are no valid arguments, such as *assert_never()*:

```
from typing import Never  # or NoReturn

def never_call_me(arg: Never) -> None:
    pass

def int_or_str(arg: int | str) -> None:
    never_call_me(arg)  # type checker error
    match arg:
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _:
            never_call_me(arg)  # OK, arg is of type Never (or NoReturn)
```

`Never` and `NoReturn` have the same meaning in the type system and static type checkers treat both equivalently.

Added in version 3.6.2: Added *NoReturn*.

Added in version 3.11: Added *Never*.

typing.**Self**

Special type to represent the current enclosed class.

예를 들면:

```
from typing import Self, reveal_type

class Foo:
    def return_self(self) -> Self:
        ...
        return self

class SubclassOfFoo(Foo): pass

reveal_type(Foo().return_self())  # Revealed type is "Foo"
reveal_type(SubclassOfFoo().return_self())  # Revealed type is "SubclassOfFoo"
```

This annotation is semantically equivalent to the following, albeit in a more succinct fashion:

```
from typing import TypeVar

Self = TypeVar("Self", bound="Foo")

class Foo:
    def return_self(self: Self) -> Self:
        ...
        return self
```

In general, if something returns `self`, as in the above examples, you should use `Self` as the return annotation. If `Foo.return_self` was annotated as returning `"Foo"`, then the type checker would infer the object returned from `SubclassOfFoo.return_self` as being of type `Foo` rather than `SubclassOfFoo`.

Other common use cases include:

- *classmethod*s that are used as alternative constructors and return instances of the `cls` parameter.
- Annotating an `__enter__()` method which returns self.

You should not use `Self` as the return annotation if the method is not guaranteed to return an instance of a subclass when the class is subclassed:

```
class Eggs:
    # Self would be an incorrect return annotation here,
    # as the object returned is always an instance of Eggs,
    # even in subclasses
    def returns_eggs(self) -> "Eggs":
        return Eggs()
```

See **PEP 673** for more details.

Added in version 3.11.

`typing.`**`TypeAlias`**

Special annotation for explicitly declaring a *type alias*.

예를 들면:

```
from typing import TypeAlias

Factors: TypeAlias = list[int]
```

`TypeAlias` is particularly useful on older Python versions for annotating aliases that make use of forward references, as it can be hard for type checkers to distinguish these from normal variable assignments:

```
from typing import Generic, TypeAlias, TypeVar

T = TypeVar("T")

# "Box" does not exist yet,
# so we have to use quotes for the forward reference on Python <3.12.
# Using ``TypeAlias`` tells the type checker that this is a type alias␣
↪declaration,
# not a variable assignment to a string.
BoxOfStrings: TypeAlias = "Box[str]"

class Box(Generic[T]):
    @classmethod
    def make_box_of_strings(cls) -> BoxOfStrings: ...
```

See **PEP 613** for more details.

Added in version 3.10.

버전 3.12부터 폐지됨: *TypeAlias* is deprecated in favor of the `type` statement, which creates instances of *TypeAliasType* and which natively supports forward references. Note that while *TypeAlias* and *TypeAliasType* serve similar purposes and have similar names, they are distinct and the latter is not the type of the former. Removal of *TypeAlias* is not currently planned, but users are encouraged to migrate to `type` statements.

특수 형태

These can be used as types in annotations. They all support subscription using `[]`, but each has a unique syntax.

class typing.**Union**

Union type; `Union[X, Y]` is equivalent to `X | Y` and means either X or Y.

To define a union, use e.g. `Union[int, str]` or the shorthand `int | str`. Using that shorthand is recommended. Details:

- 인자는 형이어야 하며 적어도 하나 있어야 합니다.
- 공용체의 공용체는 펼쳐집니다, 예를 들어:

```
Union[Union[int, str], float] == Union[int, str, float]
```

However, this does not apply to unions referenced through a type alias, to avoid forcing evaluation of the underlying *TypeAliasType*:

```
type A = Union[int, str]
Union[A, float] != Union[int, str, float]
```

- 단일 인자의 공용체는 사라집니다. 예를 들어:

```
Union[int] == int  # The constructor actually returns int
```

- 중복 인자는 건너뜁니다. 예를 들어:

```
Union[int, str, int] == Union[int, str] == int | str
```

- 공용체를 비교할 때, 인자 순서가 무시됩니다, 예를 들어:

```
Union[int, str] == Union[str, int]
```

- You cannot subclass or instantiate a `Union`.
- `Union[X][Y]` 라고 쓸 수 없습니다.

버전 3.7에서 변경: 실행 시간에 공용체의 명시적 서브 클래스를 제거하지 않습니다.

버전 3.10에서 변경: Unions can now be written as `X | Y`. See *union type expressions*.

버전 3.14에서 변경: *types.UnionType* is now an alias for *Union*, and both `Union[int, str]` and `int | str` create instances of the same class. To check whether an object is a `Union` at runtime, use `isinstance(obj, Union)`. For compatibility with earlier versions of Python, use `get_origin(obj) is typing.Union or get_origin(obj) is types.UnionType`.

typing.**Optional**

`Optional[X]` is equivalent to `X | None` (or `Union[X, None]`).

이는 기본값을 갖는 선택적 인자와 같은 개념이 아님에 유의하십시오. 단지 선택적이기 때문에 기본값을 갖는 선택적 인자가 형 어노테이션에 `Optional` 한정자가 필요하지는 않습니다. 예를 들면:

```
def foo(arg: int = 0) -> None:
    ...
```

한편, 명시적인 `None` 값이 허용되면, 인자가 선택적인지와 관계없이 `Optional`을 사용하는 것이 적합합니다. 예를 들면:

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

버전 3.10에서 변경: Optional can now be written as `X | None`. See *union type expressions*.

typing.**Concatenate**

Special form for annotating higher-order functions.

`Concatenate` can be used in conjunction with *Callable* and `ParamSpec` to annotate a higher-order callable which adds, removes, or transforms parameters of another callable. Usage is in the form `Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable]`. `Concatenate` is currently only valid when used as the first argument to a *Callable*. The last parameter to `Concatenate` must be a `ParamSpec` or ellipsis (`...`).

For example, to annotate a decorator `with_lock` which provides a `threading.Lock` to the decorated function, `Concatenate` can be used to indicate that `with_lock` expects a callable which takes in a `Lock` as the first argument, and returns a callable with a different type signature. In this case, the `ParamSpec` indicates that the returned callable' s parameter types are dependent on the parameter types of the callable being passed in:

```
from collections.abc import Callable
from threading import Lock
from typing import Concatenate

# Use this lock to ensure that only one thread is executing a function
# at any time.
my_lock = Lock()

def with_lock[**P, R](f: Callable[Concatenate[Lock, P], R]) -> Callable[P, R]:
    '''A type-safe decorator which provides a lock.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> R:
        # Provide the lock as the first argument.
        return f(my_lock, *args, **kwargs)
    return inner

@with_lock
def sum_threadsafe(lock: Lock, numbers: list[float]) -> float:
    '''Add a list of numbers together in a thread-safe manner.'''
    with lock:
        return sum(numbers)

# We don't need to pass in the lock ourselves thanks to the decorator.
sum_threadsafe([1.1, 2.2, 3.3])
```

Added in version 3.10.

더 보기

- **PEP 612** – Parameter Specification Variables (the PEP which introduced `ParamSpec` and `Concatenate`)
- *ParamSpec*
- *Annotating callable objects*

typing.**Literal**

Special typing form to define "literal types".

`Literal` can be used to indicate to type checkers that the annotated object has a value equivalent to one of the provided literals.

예를 들면:

```
def validate_simple(data: Any) -> Literal[True]:  # always returns True
    ...

type Mode = Literal['r', 'rb', 'w', 'wb']
def open_helper(file: str, mode: Mode) -> str:
    ...

open_helper('/some/path', 'r')       # Passes type check
open_helper('/other/path', 'typo')   # Error in type checker
```

`Literal[...]`은 서브 클래싱 될 수 없습니다. 실행 시간에는, 임의의 값이 `Literal[...]`에 대한 형 인자로 허용되지만, 형 검사기는 제한을 부과할 수 있습니다. 리터럴 형에 대한 자세한 내용은 **PEP 586**을 참조하십시오.

Additional details:

- The arguments must be literal values and there must be at least one.
- Nested `Literal` types are flattened, e.g.:

  ```
  assert Literal[Literal[1, 2], 3] == Literal[1, 2, 3]
  ```

  However, this does not apply to `Literal` types referenced through a type alias, to avoid forcing evaluation of the underlying *`TypeAliasType`*:

  ```
  type A = Literal[1, 2]
  assert Literal[A, 3] != Literal[1, 2, 3]
  ```

- 중복 인자는 건너뜁니다. 예를 들어:

  ```
  assert Literal[1, 2, 1] == Literal[1, 2]
  ```

- When comparing literals, the argument order is ignored, e.g.:

  ```
  assert Literal[1, 2] == Literal[2, 1]
  ```

- You cannot subclass or instantiate a `Literal`.
- You cannot write `Literal[X][Y]`.

Added in version 3.8.

버전 3.9.1에서 변경: `Literal` now de-duplicates parameters. Equality comparisons of `Literal` objects are no longer order dependent. `Literal` objects will now raise a *`TypeError`* exception during equality comparisons if one of their parameters are not *hashable*.

`typing.`**`ClassVar`**

클래스 변수를 표시하기 위한 특수 형 구조물.

PEP 526에서 소개된 것처럼, ClassVar로 감싼 변수 어노테이션은 주어진 어트리뷰트가 클래스 변수로 사용되도록 의도되었으며 해당 클래스의 인스턴스에 설정되어서는 안 됨을 나타냅니다. 용법:

```
class Starship:
    stats: ClassVar[dict[str, int]] = {} # class variable
    damage: int = 10                     # instance variable
```

*`ClassVar`*는 형만 받아들이며 더는 서브 스크립트 할 수 없습니다.

*`ClassVar`*는 클래스 자체가 아니므로, *`isinstance()`*나 *`issubclass()`*와 함께 사용하면 안 됩니다. *`ClassVar`*는 파이썬 실행 시간 동작을 변경하지 않지만, 제삼자 형 검사기에서 사용할 수 있습니다. 예를 들어, 형 검사기는 다음 코드를 에러로 표시 할 수 있습니다:

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on instance
Starship.stats = {}     # This is OK
```

Added in version 3.5.3.

버전 3.13에서 변경: *ClassVar* can now be nested in *Final* and vice versa.

typing.**Final**

Special typing construct to indicate final names to type checkers.

Final names cannot be reassigned in any scope. Final names declared in class scopes cannot be overridden in subclasses.

예를 들면:

```
MAX_SIZE: Final = 9000
MAX_SIZE += 1  # Error reported by type checker

class Connection:
    TIMEOUT: Final[int] = 10

class FastConnector(Connection):
    TIMEOUT = 1  # Error reported by type checker
```

이러한 속성에 대한 실행 시간 검사는 없습니다. 자세한 내용은 **PEP 591**을 참조하십시오.

Added in version 3.8.

버전 3.13에서 변경: *Final* can now be nested in *ClassVar* and vice versa.

typing.**Required**

Special typing construct to mark a *TypedDict* key as required.

This is mainly useful for `total=False` TypedDicts. See *TypedDict* and **PEP 655** for more details.

Added in version 3.11.

typing.**NotRequired**

Special typing construct to mark a *TypedDict* key as potentially missing.

See *TypedDict* and **PEP 655** for more details.

Added in version 3.11.

typing.**ReadOnly**

A special typing construct to mark an item of a *TypedDict* as read-only.

예를 들면:

```
class Movie(TypedDict):
   title: ReadOnly[str]
   year: int

def mutate_movie(m: Movie) -> None:
   m["year"] = 1999  # allowed
   m["title"] = "The Matrix"  # typechecker error
```

There is no runtime checking for this property.

See *TypedDict* and **PEP 705** for more details.

Added in version 3.13.

typing.**Annotated**

Special typing form to add context-specific metadata to an annotation.

Add metadata `x` to a given type `T` by using the annotation `Annotated[T, x]`. Metadata added using `Annotated` can be used by static analysis tools or at runtime. At runtime, the metadata is stored in a `__metadata__` attribute.

If a library or tool encounters an annotation `Annotated[T, x]` and has no special logic for the metadata, it should ignore the metadata and simply treat the annotation as `T`. As such, `Annotated` can be useful for code that wants to use annotations for purposes outside Python' s static typing system.

Using `Annotated[T, x]` as an annotation still allows for static typechecking of `T`, as type checkers will simply ignore the metadata `x`. In this way, `Annotated` differs from the *`@no_type_check`* decorator, which can also be used for adding annotations outside the scope of the typing system, but completely disables typechecking for a function or class.

The responsibility of how to interpret the metadata lies with the tool or library encountering an `Annotated` annotation. A tool or library encountering an `Annotated` type can scan through the metadata elements to determine if they are of interest (e.g., using *`isinstance()`*).

Annotated[<type>, <metadata>]

Here is an example of how you might use `Annotated` to add metadata to type annotations if you were doing range analysis:

```
@dataclass
class ValueRange:
    lo: int
    hi: int

T1 = Annotated[int, ValueRange(-10, 5)]
T2 = Annotated[T1, ValueRange(-20, 3)]
```

The first argument to `Annotated` must be a valid type. Multiple metadata elements can be supplied as `Annotated` supports variadic arguments. The order of the metadata elements is preserved and matters for equality checks:

```
@dataclass
class ctype:
     kind: str

a1 = Annotated[int, ValueRange(3, 10), ctype("char")]
a2 = Annotated[int, ctype("char"), ValueRange(3, 10)]

assert a1 != a2  # Order matters
```

It is up to the tool consuming the annotations to decide whether the client is allowed to add multiple metadata elements to one annotation and how to merge those annotations.

Nested `Annotated` types are flattened. The order of the metadata elements starts with the innermost annotation:

```
assert Annotated[Annotated[int, ValueRange(3, 10)], ctype("char")] ==␣
↪Annotated[
    int, ValueRange(3, 10), ctype("char")
]
```

However, this does not apply to `Annotated` types referenced through a type alias, to avoid forcing evaluation of the underlying *`TypeAliasType`*:

```
type From3To10[T] = Annotated[T, ValueRange(3, 10)]
assert Annotated[From3To10[int], ctype("char")] != Annotated[
   int, ValueRange(3, 10), ctype("char")
]
```

Duplicated metadata elements are not removed:

```
assert Annotated[int, ValueRange(3, 10)] != Annotated[
    int, ValueRange(3, 10), ValueRange(3, 10)
]
```

`Annotated` can be used with nested and generic aliases:

```
@dataclass
class MaxLen:
    value: int

type Vec[T] = Annotated[list[tuple[T, T]], MaxLen(10)]

# When used in a type annotation, a type checker will treat "V" the␣
↪same as
# ``Annotated[list[tuple[int, int]], MaxLen(10)]``:
type V = Vec[int]
```

`Annotated` cannot be used with an unpacked *TypeVarTuple*:

```
type Variadic[*Ts] = Annotated[*Ts, Ann1] = Annotated[T1, T2, T3, ..., Ann1]
↪# NOT valid
```

where `T1`, `T2`, ⋯ are *TypeVars*. This is invalid as only one type should be passed to Annotated.

By default, *get_type_hints()* strips the metadata from annotations. Pass `include_extras=True` to have the metadata preserved:

```
>>> from typing import Annotated, get_type_hints
>>> def func(x: Annotated[int, "metadata"]) -> None: pass
...
>>> get_type_hints(func)
{'x': <class 'int'>, 'return': <class 'NoneType'>}
>>> get_type_hints(func, include_extras=True)
{'x': typing.Annotated[int, 'metadata'], 'return': <class 'NoneType'>}
```

At runtime, the metadata associated with an `Annotated` type can be retrieved via the `__metadata__` attribute:

```
>>> from typing import Annotated
>>> X = Annotated[int, "very", "important", "metadata"]
>>> X
typing.Annotated[int, 'very', 'important', 'metadata']
>>> X.__metadata__
('very', 'important', 'metadata')
```

If you want to retrieve the original type wrapped by `Annotated`, use the `__origin__` attribute:

```
>>> from typing import Annotated, get_origin
>>> Password = Annotated[str, "secret"]
>>> Password.__origin__
<class 'str'>
```

Note that using *get_origin()* will return `Annotated` itself:

```
>>> get_origin(Password)
typing.Annotated
```

더 보기

PEP 593 - Flexible function and variable annotations
The PEP introducing `Annotated` to the standard library.

Added in version 3.9.

`typing.`**`TypeIs`**

Special typing construct for marking user-defined type predicate functions.

`TypeIs` can be used to annotate the return type of a user-defined type predicate function. `TypeIs` only accepts a single type argument. At runtime, functions marked this way should return a boolean and take at least one positional argument.

`TypeIs` aims to benefit *type narrowing* – a technique used by static type checkers to determine a more precise type of an expression within a program' s code flow. Usually type narrowing is done by analyzing conditional code flow and applying the narrowing to a block of code. The conditional expression here is sometimes referred to as a "type predicate":

```
def is_str(val: str | float):
    # "isinstance" type predicate
    if isinstance(val, str):
        # Type of ``val`` is narrowed to ``str``
        ...
    else:
        # Else, type of ``val`` is narrowed to ``float``.
        ...
```

Sometimes it would be convenient to use a user-defined boolean function as a type predicate. Such a function should use `TypeIs[...]` or *TypeGuard* as its return type to alert static type checkers to this intention. `TypeIs` usually has more intuitive behavior than `TypeGuard`, but it cannot be used when the input and output types are incompatible (e.g., `list[object]` to `list[int]`) or when the function does not return `True` for all instances of the narrowed type.

Using `-> TypeIs[NarrowedType]` tells the static type checker that for a given function:

1. The return value is a boolean.
2. If the return value is `True`, the type of its argument is the intersection of the argument' s original type and `NarrowedType`.
3. If the return value is `False`, the type of its argument is narrowed to exclude `NarrowedType`.

예를 들면:

```
from typing import assert_type, final, TypeIs

class Parent: pass
class Child(Parent): pass
@final
class Unrelated: pass

def is_parent(val: object) -> TypeIs[Parent]:
    return isinstance(val, Parent)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def run(arg: Child | Unrelated):
    if is_parent(arg):
        # Type of ``arg`` is narrowed to the intersection
        # of ``Parent`` and ``Child``, which is equivalent to
        # ``Child``.
        assert_type(arg, Child)
    else:
        # Type of ``arg`` is narrowed to exclude ``Parent``,
        # so only ``Unrelated`` is left.
        assert_type(arg, Unrelated)
```

The type inside `TypeIs` must be consistent with the type of the function' s argument; if it is not, static type checkers will raise an error. An incorrectly written `TypeIs` function can lead to unsound behavior in the type system; it is the user' s responsibility to write such functions in a type-safe manner.

If a `TypeIs` function is a class or instance method, then the type in `TypeIs` maps to the type of the second parameter (after `cls` or `self`).

In short, the form `def foo(arg: TypeA) -> TypeIs[TypeB]: ...`, means that if `foo(arg)` returns `True`, then `arg` is an instance of `TypeB`, and if it returns `False`, it is not an instance of `TypeB`.

`TypeIs` also works with type variables. For more information, see **PEP 742** (Narrowing types with `TypeIs`).

Added in version 3.13.

`typing.`**`TypeGuard`**

Special typing construct for marking user-defined type predicate functions.

Type predicate functions are user-defined functions that return whether their argument is an instance of a particular type. `TypeGuard` works similarly to *`TypeIs`*, but has subtly different effects on type checking behavior (see below).

Using `-> TypeGuard` tells the static type checker that for a given function:

1. The return value is a boolean.
2. If the return value is `True`, the type of its argument is the type inside `TypeGuard`.

`TypeGuard` also works with type variables. See **PEP 647** for more details.

예를 들면:

```
def is_str_list(val: list[object]) -> TypeGuard[list[str]]:
    '''Determines whether all objects in the list are strings'''
    return all(isinstance(x, str) for x in val)

def func1(val: list[object]):
    if is_str_list(val):
        # Type of ``val`` is narrowed to ``list[str]``.
        print(" ".join(val))
    else:
        # Type of ``val`` remains as ``list[object]``.
        print("Not a list of strings!")
```

`TypeIs` and `TypeGuard` differ in the following ways:

- `TypeIs` requires the narrowed type to be a subtype of the input type, while `TypeGuard` does not. The main reason is to allow for things like narrowing `list[object]` to `list[str]` even though the latter is not a subtype of the former, since `list` is invariant.
- When a `TypeGuard` function returns `True`, type checkers narrow the type of the variable to exactly the `TypeGuard` type. When a `TypeIs` function returns `True`, type checkers can infer a more precise type

combining the previously known type of the variable with the `TypeIs` type. (Technically, this is known as an intersection type.)

- When a `TypeGuard` function returns `False`, type checkers cannot narrow the type of the variable at all. When a `TypeIs` function returns `False`, type checkers can narrow the type of the variable to exclude the `TypeIs` type.

Added in version 3.10.

`typing.`**`Unpack`**

Typing operator to conceptually mark an object as having been unpacked.

For example, using the unpack operator `*` on a *type variable tuple* is equivalent to using `Unpack` to mark the type variable tuple as having been unpacked:

```
Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]
# Effectively does:
tup: tuple[Unpack[Ts]]
```

In fact, `Unpack` can be used interchangeably with `*` in the context of *typing.TypeVarTuple* and *builtins.tuple* types. You might see `Unpack` being used explicitly in older versions of Python, where `*` couldn' t be used in certain places:

```
# In older versions of Python, TypeVarTuple and Unpack
# are located in the `typing_extensions` backports package.
from typing_extensions import TypeVarTuple, Unpack

Ts = TypeVarTuple('Ts')
tup: tuple[*Ts]         # Syntax error on Python <= 3.10!
tup: tuple[Unpack[Ts]]  # Semantically equivalent, and backwards-compatible
```

`Unpack` can also be used along with *typing.TypedDict* for typing `**kwargs` in a function signature:

```
from typing import TypedDict, Unpack

class Movie(TypedDict):
    name: str
    year: int

# This function expects two keyword arguments - `name` of type `str`
# and `year` of type `int`.
def foo(**kwargs: Unpack[Movie]): ...
```

See **PEP 692** for more details on using `Unpack` for `**kwargs` typing.

Added in version 3.11.

Building generic types and type aliases

The following classes should not be used directly as annotations. Their intended purpose is to be building blocks for creating generic types and type aliases.

These objects can be created through special syntax (type parameter lists and the `type` statement). For compatibility with Python 3.11 and earlier, they can also be created without the dedicated syntax, as documented below.

`class` `typing.`**`Generic`**

제네릭 형을 위한 추상 베이스 클래스.

A generic type is typically declared by adding a list of type parameters after the class name:

```
class Mapping[KT, VT]:
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

Such a class implicitly inherits from `Generic`. The runtime semantics of this syntax are discussed in the Language Reference.

이 클래스는 다음과 같이 사용할 수 있습니다:

```
def lookup_name[X, Y](mapping: Mapping[X, Y], key: X, default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

Here the brackets after the function name indicate a generic function.

For backwards compatibility, generic classes can also be declared by explicitly inheriting from `Generic`. In this case, the type parameters must be declared separately:

```
KT = TypeVar('KT')
VT = TypeVar('VT')

class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

class typing.**TypeVar**(*name*, **constraints*, *bound=None*, *covariant=False*, *contravariant=False*, *infer_variance=False*, *default=typing.NoDefault*)

형 변수.

The preferred way to construct a type variable is via the dedicated syntax for generic functions, generic classes, and generic type aliases:

```
class Sequence[T]:  # T is a TypeVar
    ...
```

This syntax can also be used to create bounded and constrained type variables:

```
class StrSequence[S: str]:  # S is a TypeVar with a `str` upper bound;
    ...                     # we can say that S is "bounded by `str`"


class StrOrBytesSequence[A: (str, bytes)]:  # A is a TypeVar constrained to␣
↪str or bytes
    ...
```

However, if desired, reusable type variables can also be constructed manually, like so:

```
T = TypeVar('T')  # Can be anything
S = TypeVar('S', bound=str)  # Can be any subtype of str
A = TypeVar('A', str, bytes)  # Must be exactly str or bytes
```

Type variables exist primarily for the benefit of static type checkers. They serve as the parameters for generic types as well as for generic function and type alias definitions. See `Generic` for more information on generic types. Generic functions work as follows:

```
def repeat[T](x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n


def print_capitalized[S: str](x: S) -> S:
    """Print x capitalized, and return x."""
    print(x.capitalize())
    return x


def concatenate[A: (str, bytes)](x: A, y: A) -> A:
    """Add two strings or bytes objects together."""
    return x + y
```

Note that type variables can be *bounded*, *constrained*, or neither, but cannot be both bounded *and* constrained.

The variance of type variables is inferred by type checkers when they are created through the type parameter syntax or when `infer_variance=True` is passed. Manually created type variables may be explicitly marked covariant or contravariant by passing `covariant=True` or `contravariant=True`. By default, manually created type variables are invariant. See **PEP 484** and **PEP 695** for more details.

Bounded type variables and constrained type variables have different semantics in several important ways. Using a *bounded* type variable means that the `TypeVar` will be solved using the most specific type possible:

```
x = print_capitalized('a string')
reveal_type(x)  # revealed type is str

class StringSubclass(str):
    pass

y = print_capitalized(StringSubclass('another string'))
reveal_type(y)  # revealed type is StringSubclass

z = print_capitalized(45)  # error: int is not a subtype of str
```

The upper bound of a type variable can be a concrete type, abstract type (ABC or Protocol), or even a union of types:

```
# Can be anything with an __abs__ method
def print_abs[T: SupportsAbs](arg: T) -> None:
    print("Absolute value:", abs(arg))

U = TypeVar('U', bound=str|bytes)  # Can be any subtype of the union str|bytes
V = TypeVar('V', bound=SupportsAbs)  # Can be anything with an __abs__ method
```

Using a *constrained* type variable, however, means that the `TypeVar` can only ever be solved as being exactly one of the constraints given:

```
a = concatenate('one', 'two')
reveal_type(a)  # revealed type is str

b = concatenate(StringSubclass('one'), StringSubclass('two'))
reveal_type(b)  # revealed type is str, despite StringSubclass being passed in

c = concatenate('one', b'two')  # error: type variable 'A' can be either str␣
↪or bytes in a function call, but not both
```

At runtime, `isinstance(x, T)` will raise *`TypeError`*.

__name__

The name of the type variable.

__covariant__

Whether the type var has been explicitly marked as covariant.

__contravariant__

Whether the type var has been explicitly marked as contravariant.

__infer_variance__

Whether the type variable' s variance should be inferred by type checkers.

Added in version 3.12.

__bound__

The upper bound of the type variable, if any.

버전 3.12에서 변경: For type variables created through type parameter syntax, the bound is evaluated only when the attribute is accessed, not when the type variable is created (see lazy-evaluation).

evaluate_bound()

An *evaluate function* corresponding to the `__bound__` attribute. When called directly, this method supports only the `VALUE` format, which is equivalent to accessing the `__bound__` attribute directly, but the method object can be passed to `annotationlib.call_evaluate_function()` to evaluate the value in a different format.

Added in version 3.14.

__constraints__

A tuple containing the constraints of the type variable, if any.

버전 3.12에서 변경: For type variables created through type parameter syntax, the constraints are evaluated only when the attribute is accessed, not when the type variable is created (see lazy-evaluation).

evaluate_constraints()

An *evaluate function* corresponding to the `__constraints__` attribute. When called directly, this method supports only the `VALUE` format, which is equivalent to accessing the `__constraints__` attribute directly, but the method object can be passed to `annotationlib.call_evaluate_function()` to evaluate the value in a different format.

Added in version 3.14.

__default__

The default value of the type variable, or `typing.NoDefault` if it has no default.

Added in version 3.13.

evaluate_default()

An *evaluate function* corresponding to the `__default__` attribute. When called directly, this method supports only the `VALUE` format, which is equivalent to accessing the `__default__` attribute directly, but the method object can be passed to `annotationlib.call_evaluate_function()` to evaluate the value in a different format.

Added in version 3.14.

has_default()

Return whether or not the type variable has a default value. This is equivalent to checking whether `__default__` is not the `typing.NoDefault` singleton, except that it does not force evaluation of the lazily evaluated default value.

Added in version 3.13.

버전 3.12에서 변경: Type variables can now be declared using the type parameter syntax introduced by **PEP 695**. The `infer_variance` parameter was added.

버전 3.13에서 변경: Support for default values was added.

class typing.**TypeVarTuple**(*name*, *, *default=typing.NoDefault*)

Type variable tuple. A specialized form of *type variable* that enables *variadic* generics.

Type variable tuples can be declared in type parameter lists using a single asterisk (*) before the name:

```
def move_first_element_to_last[T, *Ts](tup: tuple[T, *Ts]) -> tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

Or by explicitly invoking the TypeVarTuple constructor:

```
T = TypeVar("T")
Ts = TypeVarTuple("Ts")

def move_first_element_to_last(tup: tuple[T, *Ts]) -> tuple[*Ts, T]:
    return (*tup[1:], tup[0])
```

A normal type variable enables parameterization with a single type. A type variable tuple, in contrast, allows parameterization with an *arbitrary* number of types by acting like an *arbitrary* number of type variables wrapped in a tuple. For example:

```
# T is bound to int, Ts is bound to ()
# Return value is (1,), which has type tuple[int]
move_first_element_to_last(tup=(1,))

# T is bound to int, Ts is bound to (str,)
# Return value is ('spam', 1), which has type tuple[str, int]
move_first_element_to_last(tup=(1, 'spam'))

# T is bound to int, Ts is bound to (str, float)
# Return value is ('spam', 3.0, 1), which has type tuple[str, float, int]
move_first_element_to_last(tup=(1, 'spam', 3.0))

# This fails to type check (and fails at runtime)
# because tuple[()] is not compatible with tuple[T, *Ts]
# (at least one element is required)
move_first_element_to_last(tup=())
```

Note the use of the unpacking operator * in tuple[T, *Ts]. Conceptually, you can think of Ts as a tuple of type variables (T1, T2, ...). tuple[T, *Ts] would then become tuple[T, *(T1, T2, ...)], which is equivalent to tuple[T, T1, T2, ...]. (Note that in older versions of Python, you might see this written using *Unpack* instead, as Unpack[Ts].)

Type variable tuples must *always* be unpacked. This helps distinguish type variable tuples from normal type variables:

```
x: Ts           # Not valid
x: tuple[Ts]    # Not valid
x: tuple[*Ts]   # The correct way to do it
```

Type variable tuples can be used in the same contexts as normal type variables. For example, in class definitions, arguments, and return types:

```
class Array[*Shape]:
    def __getitem__(self, key: tuple[*Shape]) -> float: ...
    def __abs__(self) -> "Array[*Shape]": ...
    def get_shape(self) -> tuple[*Shape]: ...
```

Type variable tuples can be happily combined with normal type variables:

```
class Array[DType, *Shape]:  # This is fine
    pass

class Array2[*Shape, DType]:  # This would also be fine
    pass

class Height: ...
class Width: ...

float_array_1d: Array[float, Height] = Array()      # Totally fine
int_array_2d: Array[int, Height, Width] = Array()   # Yup, fine too
```

However, note that at most one type variable tuple may appear in a single list of type arguments or type parameters:

```
x: tuple[*Ts, *Ts]             # Not valid
class Array[*Shape, *Shape]:   # Not valid
    pass
```

Finally, an unpacked type variable tuple can be used as the type annotation of `*args`:

```
def call_soon[*Ts](
    callback: Callable[[*Ts], None],
    *args: *Ts
) -> None:
    ...
    callback(*args)
```

In contrast to non-unpacked annotations of `*args` - e.g. `*args: int`, which would specify that *all* arguments are `int` - `*args: *Ts` enables reference to the types of the *individual* arguments in `*args`. Here, this allows us to ensure the types of the `*args` passed to `call_soon` match the types of the (positional) arguments of `callback`.

See **PEP 646** for more details on type variable tuples.

__name__

The name of the type variable tuple.

__default__

The default value of the type variable tuple, or `typing.NoDefault` if it has no default.

Added in version 3.13.

evaluate_default()

An *evaluate function* corresponding to the `__default__` attribute. When called directly, this method supports only the `VALUE` format, which is equivalent to accessing the `__default__` attribute directly, but the method object can be passed to `annotationlib.call_evaluate_function()` to evaluate the value in a different format.

Added in version 3.14.

has_default()

Return whether or not the type variable tuple has a default value. This is equivalent to checking whether `__default__` is not the `typing.NoDefault` singleton, except that it does not force evaluation of the lazily evaluated default value.

Added in version 3.13.

Added in version 3.11.

버전 3.12에서 변경: Type variable tuples can now be declared using the type parameter syntax introduced by **PEP 695**.

버전 3.13에서 변경: Support for default values was added.

class typing.**ParamSpec**(*name*, *, *bound=None*, *covariant=False*, *contravariant=False*, *default=typing.NoDefault*)

Parameter specification variable. A specialized version of *type variables*.

In type parameter lists, parameter specifications can be declared with two asterisks (`**`):

```
type IntFunc[**P] = Callable[P, int]
```

For compatibility with Python 3.11 and earlier, `ParamSpec` objects can also be created as follows:

```
P = ParamSpec('P')
```

Parameter specification variables exist primarily for the benefit of static type checkers. They are used to forward the parameter types of one callable to another callable – a pattern commonly found in higher order functions and decorators. They are only valid when used in `Concatenate`, or as the first argument to `Callable`, or as parameters for user-defined Generics. See *`Generic`* for more information on generic types.

For example, to add basic logging to a function, one can create a decorator `add_logging` to log function calls. The parameter specification variable tells the type checker that the callable passed into the decorator and the new callable returned by it have inter-dependent type parameters:

```
from collections.abc import Callable
import logging

def add_logging[T, **P](f: Callable[P, T]) -> Callable[P, T]:
    '''A type-safe decorator to add logging to a function.'''
    def inner(*args: P.args, **kwargs: P.kwargs) -> T:
        logging.info(f'{f.__name__} was called')
        return f(*args, **kwargs)
    return inner

@add_logging
def add_two(x: float, y: float) -> float:
    '''Add two numbers together.'''
    return x + y
```

Without `ParamSpec`, the simplest way to annotate this previously was to use a *`TypeVar`* with upper bound `Callable[..., Any]`. However this causes two problems:

1. The type checker can' t type check the `inner` function because `*args` and `**kwargs` have to be typed *`Any`*.
2. *`cast()`* may be required in the body of the `add_logging` decorator when returning the `inner` function, or the static type checker must be told to ignore the `return inner`.

args

kwargs

Since `ParamSpec` captures both positional and keyword parameters, `P.args` and `P.kwargs` can be used to split a `ParamSpec` into its components. `P.args` represents the tuple of positional parameters in a given call and should only be used to annotate `*args`. `P.kwargs` represents the mapping of keyword parameters to their values in a given call, and should be only be used to annotate `**kwargs`. Both attributes require the annotated parameter to be in scope. At runtime, `P.args` and `P.kwargs` are instances respectively of *`ParamSpecArgs`* and *`ParamSpecKwargs`*.

__name__

The name of the parameter specification.

__default__

The default value of the parameter specification, or *typing.NoDefault* if it has no default.

Added in version 3.13.

evaluate_default()

An *evaluate function* corresponding to the *__default__* attribute. When called directly, this method supports only the *VALUE* format, which is equivalent to accessing the *__default__* attribute directly, but the method object can be passed to *annotationlib.call_evaluate_function()* to evaluate the value in a different format.

Added in version 3.14.

has_default()

Return whether or not the parameter specification has a default value. This is equivalent to checking whether *__default__* is not the *typing.NoDefault* singleton, except that it does not force evaluation of the lazily evaluated default value.

Added in version 3.13.

Parameter specification variables created with `covariant=True` or `contravariant=True` can be used to declare covariant or contravariant generic types. The `bound` argument is also accepted, similar to *TypeVar*. However the actual semantics of these keywords are yet to be decided.

Added in version 3.10.

버전 3.12에서 변경: Parameter specifications can now be declared using the type parameter syntax introduced by **PEP 695**.

버전 3.13에서 변경: Support for default values was added.

참고

Only parameter specification variables defined in global scope can be pickled.

더 보기

- **PEP 612** – Parameter Specification Variables (the PEP which introduced `ParamSpec` and `Concatenate`)
- *Concatenate*
- *Annotating callable objects*

typing.**ParamSpecArgs**

typing.**ParamSpecKwargs**

Arguments and keyword arguments attributes of a *ParamSpec*. The `P.args` attribute of a `ParamSpec` is an instance of `ParamSpecArgs`, and `P.kwargs` is an instance of `ParamSpecKwargs`. They are intended for runtime introspection and have no special meaning to static type checkers.

Calling *get_origin()* on either of these objects will return the original `ParamSpec`:

```
>>> from typing import ParamSpec, get_origin
>>> P = ParamSpec("P")
>>> get_origin(P.args) is P
True
>>> get_origin(P.kwargs) is P
True
```

Added in version 3.10.

class typing.**TypeAliasType**(*name*, *value*, *, *type_params*=())

The type of type aliases created through the `type` statement.

Example:

```
>>> type Alias = int
>>> type(Alias)
<class 'typing.TypeAliasType'>
```

Added in version 3.12.

__name__

The name of the type alias:

```
>>> type Alias = int
>>> Alias.__name__
'Alias'
```

__module__

The module in which the type alias was defined:

```
>>> type Alias = int
>>> Alias.__module__
'__main__'
```

__type_params__

The type parameters of the type alias, or an empty tuple if the alias is not generic:

```
>>> type ListOrSet[T] = list[T] | set[T]
>>> ListOrSet.__type_params__
(T,)
>>> type NotGeneric = int
>>> NotGeneric.__type_params__
()
```

__value__

The type alias' s value. This is lazily evaluated, so names used in the definition of the alias are not resolved until the `__value__` attribute is accessed:

```
>>> type Mutually = Recursive
>>> type Recursive = Mutually
>>> Mutually
Mutually
>>> Recursive
Recursive
>>> Mutually.__value__
Recursive
>>> Recursive.__value__
Mutually
```

evaluate_value()

An *evaluate function* corresponding to the `__value__` attribute. When called directly, this method supports only the `VALUE` format, which is equivalent to accessing the `__value__` attribute directly, but the method object can be passed to `annotationlib.call_evaluate_function()` to evaluate the value in a different format:

```
>>> type Alias = undefined
>>> Alias.__value__
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Traceback (most recent call last):
...
NameError: name 'undefined' is not defined
>>> from annotationlib import Format, call_evaluate_function
>>> Alias.evaluate_value(Format.VALUE)
Traceback (most recent call last):
...
NameError: name 'undefined' is not defined
>>> call_evaluate_function(Alias.evaluate_value, Format.FORWARDREF)
ForwardRef('undefined')
```

Added in version 3.14.

Unpacking

Type aliases support star unpacking using the `*Alias` syntax. This is equivalent to using `Unpack[Alias]` directly:

```
>>> type Alias = tuple[int, str]
>>> type Unpacked = tuple[bool, *Alias]
>>> Unpacked.__value__
tuple[bool, typing.Unpack[Alias]]
```

Added in version 3.14.

기타 특수 지시자

These functions and classes should not be used directly as annotations. Their intended purpose is to be building blocks for creating and declaring types.

class typing.**NamedTuple**

형 지정된 (typed) *collections.namedtuple()* 버전.

용법:

```
class Employee(NamedTuple):
    name: str
    id: int
```

이것은 다음과 동등합니다:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

필드에 기본값을 부여하려면, 클래스 바디에서 그 값을 대입할 수 있습니다:

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

기본값이 있는 필드는 기본값이 없는 모든 필드 뒤에 와야 합니다.

The resulting class has an extra attribute `__annotations__` giving a dict that maps the field names to the field types. (The field names are in the `_fields` attribute and the default values are in the `_field_defaults` attribute, both of which are part of the *namedtuple()* API.)

`NamedTuple` 서브 클래스는 독스트링과 메서드도 가질 수 있습니다:

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

`NamedTuple` subclasses can be generic:

```
class Group[T](NamedTuple):
    key: T
    group: list[T]
```

이전 버전과 호환되는 사용법:

```
# For creating a generic NamedTuple on Python 3.11
T = TypeVar("T")

class Group(NamedTuple, Generic[T]):
    key: T
    group: list[T]

# A functional syntax is also supported
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

버전 3.6에서 변경: **PEP 526** 변수 어노테이션 문법 지원을 추가했습니다.

버전 3.6.1에서 변경: 기본값, 메서드 및 독스트링에 대한 지원을 추가했습니다.

버전 3.8에서 변경: `_field_types`와 `__annotations__` 어트리뷰트는 이제 `OrderedDict` 인스턴스가 아닌 일반 딕셔너리입니다.

버전 3.9에서 변경: `_field_types` 어트리뷰트를 제거하고, 같은 정보를 가지는 더 표준적인 `__annotations__` 어트리뷰트로 대체했습니다.

버전 3.11에서 변경: Added support for generic namedtuples.

버전 3.14에서 변경: Using `super()` (and the `__class__` *closure variable*) in methods of `NamedTuple` subclasses is unsupported and causes a `TypeError`.

Deprecated since version 3.13, will be removed in version 3.15: The undocumented keyword argument syntax for creating NamedTuple classes (`NT = NamedTuple("NT", x=int)`) is deprecated, and will be disallowed in 3.15. Use the class-based syntax or the functional syntax instead.

Deprecated since version 3.13, will be removed in version 3.15: When using the functional syntax to create a NamedTuple class, failing to pass a value to the 'fields' parameter (`NT = NamedTuple("NT")`) is deprecated. Passing `None` to the 'fields' parameter (`NT = NamedTuple("NT", None)`) is also deprecated. Both will be disallowed in Python 3.15. To create a NamedTuple class with 0 fields, use `class NT(NamedTuple): pass` or `NT = NamedTuple("NT", [])`.

class typing.**NewType**(*name*, *tp*)

Helper class to create low-overhead *distinct types*.

A `NewType` is considered a distinct type by a typechecker. At runtime, however, calling a `NewType` returns its argument unchanged.

용법:

```
UserId = NewType('UserId', int)  # Declare the NewType "UserId"
first_user = UserId(1)  # "UserId" returns the argument unchanged at runtime
```

__module__

The module in which the new type is defined.

__name__

The name of the new type.

__supertype__

The type that the new type is based on.

Added in version 3.5.2.

버전 3.10에서 변경: `NewType` is now a class rather than a function.

class typing.**Protocol**(*Generic*)

Base class for protocol classes.

Protocol classes are defined like this:

```
class Proto(Protocol):
    def meth(self) -> int:
        ...
```

이러한 클래스는 주로 구조적 서브 타이핑(정적 덕 타이핑)을 인식하는 정적 형 검사기와 함께 사용됩니다, 예를 들어:

```
class C:
    def meth(self) -> int:
        return 0

def func(x: Proto) -> int:
    return x.meth()

func(C())  # Passes static type check
```

See **PEP 544** for more details. Protocol classes decorated with *runtime_checkable()* (described later) act as simple-minded runtime protocols that check only the presence of given attributes, ignoring their type signatures. Protocol classes without this decorator cannot be used as the second argument to *isinstance()* or *issubclass()*.

프로토콜 클래스는 제네릭일 수 있습니다, 예를 들어:

```
class GenProto[T](Protocol):
    def meth(self) -> T:
        ...
```

In code that needs to be compatible with Python 3.11 or older, generic Protocols can be written as follows:

```
T = TypeVar("T")

class GenProto(Protocol[T]):
    def meth(self) -> T:
        ...
```

Added in version 3.8.

@typing.**runtime_checkable**

프로토콜 클래스를 실행 시간 프로토콜로 표시합니다.

Such a protocol can be used with *isinstance()* and *issubclass()*. This allows a simple-minded structural check, very similar to "one trick ponies" in *collections.abc* such as *Iterable*. For example:

```
@runtime_checkable
class Closable(Protocol):
    def close(self): ...

assert isinstance(open('/some/file'), Closable)

@runtime_checkable
class Named(Protocol):
    name: str

import threading
assert isinstance(threading.Thread(name='Bob'), Named)
```

This decorator raises *TypeError* when applied to a non-protocol class.

> **참고**
>
> `runtime_checkable()` will check only the presence of the required methods or attributes, not their type signatures or types. For example, *ssl.SSLObject* is a class, therefore it passes an *issubclass()* check against *Callable*. However, the `ssl.SSLObject.__init__` method exists only to raise a *TypeError* with a more informative message, therefore making it impossible to call (instantiate) *ssl.SSLObject*.

> **참고**
>
> An *isinstance()* check against a runtime-checkable protocol can be surprisingly slow compared to an `isinstance()` check against a non-protocol class. Consider using alternative idioms such as *hasattr()* calls for structural checks in performance-sensitive code.

Added in version 3.8.

버전 3.12에서 변경: The internal implementation of *isinstance()* checks against runtime-checkable protocols now uses *inspect.getattr_static()* to look up attributes (previously, *hasattr()* was used). As a result, some objects which used to be considered instances of a runtime-checkable protocol may no longer be considered instances of that protocol on Python 3.12+, and vice versa. Most users are unlikely to be affected by this change.

버전 3.12에서 변경: The members of a runtime-checkable protocol are now considered "frozen" at runtime as soon as the class has been created. Monkey-patching attributes onto a runtime-checkable protocol will still work, but will have no impact on *isinstance()* checks comparing objects to the protocol. See What' s new in Python 3.12 for more details.

class typing.**TypedDict**(*dict*)

딕셔너리에 형 힌트를 추가하는 특수 구조. 실행 시간에 일반 *dict*입니다.

`TypedDict`는 모든 인스턴스가 각 키가 일관된 형의 값에 연관되는, 특정한 키 집합을 갖도록 기대되는 딕셔너리 형을 선언합니다. 이 기대는 실행 시간에는 검사되지 않고, 형 검사기에서만 강제됩니다. 사용법:

```
class Point2D(TypedDict):
    x: int
    y: int
    label: str

a: Point2D = {'x': 1, 'y': 2, 'label': 'good'}  # OK
b: Point2D = {'z': 3, 'label': 'bad'}           # Fails type check

assert Point2D(x=1, y=2, label='first') == dict(x=1, y=2, label='first')
```

An alternative way to create a `TypedDict` is by using function-call syntax. The second argument must be a literal *dict*:

```
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': str})
```

This functional syntax allows defining keys which are not valid identifiers, for example because they are keywords or contain hyphens, or when key names must not be mangled like regular private names:

```
# raises SyntaxError
class Point2D(TypedDict):
    in: int  # 'in' is a keyword
    x-y: int  # name with hyphens

class Definition(TypedDict):
    __schema: str  # mangled to `_Definition__schema`

# OK, functional syntax
Point2D = TypedDict('Point2D', {'in': int, 'x-y': int})
Definition = TypedDict('Definition', {'__schema': str})  # not mangled
```

By default, all keys must be present in a `TypedDict`. It is possible to mark individual keys as non-required using *NotRequired*:

```
class Point2D(TypedDict):
    x: int
    y: int
    label: NotRequired[str]

# Alternative syntax
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': NotRequired[str]})
```

This means that a `Point2D TypedDict` can have the `label` key omitted.

It is also possible to mark all keys as non-required by default by specifying a totality of `False`:

```
class Point2D(TypedDict, total=False):
    x: int
    y: int

# Alternative syntax
Point2D = TypedDict('Point2D', {'x': int, 'y': int}, total=False)
```

This means that a `Point2D TypedDict` can have any of the keys omitted. A type checker is only expected to support a literal `False` or `True` as the value of the `total` argument. `True` is the default, and makes all items defined in the class body required.

Individual keys of a `total=False TypedDict` can be marked as required using *Required*:

```
class Point2D(TypedDict, total=False):
    x: Required[int]
    y: Required[int]
    label: str

# Alternative syntax
Point2D = TypedDict('Point2D', {
    'x': Required[int],
    'y': Required[int],
    'label': str
}, total=False)
```

It is possible for a `TypedDict` type to inherit from one or more other `TypedDict` types using the class-based syntax. Usage:

```
class Point3D(Point2D):
    z: int
```

`Point3D` has three items: `x`, `y` and `z`. It is equivalent to this definition:

```
class Point3D(TypedDict):
    x: int
    y: int
    z: int
```

A `TypedDict` cannot inherit from a non-`TypedDict` class, except for *Generic*. For example:

```
class X(TypedDict):
    x: int

class Y(TypedDict):
    y: int

class Z(object): pass  # A non-TypedDict class

class XY(X, Y): pass  # OK

class XZ(X, Z): pass  # raises TypeError
```

A `TypedDict` can be generic:

```
class Group[T](TypedDict):
    key: T
    group: list[T]
```

To create a generic `TypedDict` that is compatible with Python 3.11 or lower, inherit from *Generic* explicitly:

```
T = TypeVar("T")

class Group(TypedDict, Generic[T]):
    key: T
    group: list[T]
```

A `TypedDict` can be introspected via annotations dicts (see annotations-howto for more information on annotations best practices), *__total__*, *__required_keys__*, and *__optional_keys__*.

__total__

`Point2D.__total__` gives the value of the `total` argument. Example:

```
>>> from typing import TypedDict
>>> class Point2D(TypedDict): pass
>>> Point2D.__total__
True
>>> class Point2D(TypedDict, total=False): pass
>>> Point2D.__total__
False
>>> class Point3D(Point2D): pass
>>> Point3D.__total__
True
```

This attribute reflects *only* the value of the `total` argument to the current `TypedDict` class, not whether the class is semantically total. For example, a `TypedDict` with `__total__` set to `True` may have keys

marked with *NotRequired*, or it may inherit from another `TypedDict` with `total=False`. Therefore, it is generally better to use *__required_keys__* and *__optional_keys__* for introspection.

__required_keys__

Added in version 3.9.

__optional_keys__

`Point2D.__required_keys__` and `Point2D.__optional_keys__` return *frozenset* objects containing required and non-required keys, respectively.

Keys marked with *Required* will always appear in `__required_keys__` and keys marked with *NotRequired* will always appear in `__optional_keys__`.

For backwards compatibility with Python 3.10 and below, it is also possible to use inheritance to declare both required and non-required keys in the same `TypedDict` . This is done by declaring a `TypedDict` with one value for the `total` argument and then inheriting from it in another `TypedDict` with a different value for `total`:

```
>>> class Point2D(TypedDict, total=False):
...     x: int
...     y: int
...
>>> class Point3D(Point2D):
...     z: int
...
>>> Point3D.__required_keys__ == frozenset({'z'})
True
>>> Point3D.__optional_keys__ == frozenset({'x', 'y'})
True
```

Added in version 3.9.

> **참고**
>
> If `from __future__ import annotations` is used or if annotations are given as strings, annotations are not evaluated when the `TypedDict` is defined. Therefore, the runtime introspection that `__required_keys__` and `__optional_keys__` rely on may not work properly, and the values of the attributes may be incorrect.

Support for *ReadOnly* is reflected in the following attributes:

__readonly_keys__

A *frozenset* containing the names of all read-only keys. Keys are read-only if they carry the *ReadOnly* qualifier.

Added in version 3.13.

__mutable_keys__

A *frozenset* containing the names of all mutable keys. Keys are mutable if they do not carry the *ReadOnly* qualifier.

Added in version 3.13.

See the TypedDict section in the typing documentation for more examples and detailed rules.

Added in version 3.8.

버전 3.11에서 변경: Added support for marking individual keys as *Required* or *NotRequired*. See **PEP 655**.

버전 3.11에서 변경: Added support for generic `TypedDict`s.

버전 3.13에서 변경: Removed support for the keyword-argument method of creating `TypedDict`s.

버전 3.13에서 변경: Support for the *ReadOnly* qualifier was added.

Deprecated since version 3.13, will be removed in version 3.15: When using the functional syntax to create a TypedDict class, failing to pass a value to the 'fields' parameter (`TD = TypedDict("TD")`) is deprecated. Passing `None` to the 'fields' parameter (`TD = TypedDict("TD", None)`) is also deprecated. Both will be disallowed in Python 3.15. To create a TypedDict class with 0 fields, use `class TD(TypedDict): pass` or `TD = TypedDict("TD", {})`.

프로토콜

The following protocols are provided by the `typing` module. All are decorated with *@runtime_checkable*.

class typing.**SupportsAbs**

반환형이 공변적(covariant)인 하나의 추상 메서드 `__abs__`를 가진 ABC.

class typing.**SupportsBytes**

하나의 추상 메서드 `__bytes__`를 가진 ABC.

class typing.**SupportsComplex**

하나의 추상 메서드 `__complex__`를 가진 ABC.

class typing.**SupportsFloat**

하나의 추상 메서드 `__float__`를 가진 ABC.

class typing.**SupportsIndex**

하나의 추상 메서드 `__index__`를 가진 ABC.

Added in version 3.8.

class typing.**SupportsInt**

하나의 추상 메서드 `__int__`를 가진 ABC.

class typing.**SupportsRound**

반환형이 공변적(covariant)인 하나의 추상 메서드 `__round__`를 가진 ABC.

ABCs and Protocols for working with I/O

class typing.**IO**[*AnyStr*]()

class typing.**TextIO**[*AnyStr*]()

class typing.**BinaryIO**[*AnyStr*]()

Generic class `IO[AnyStr]` and its subclasses `TextIO(IO[str])` and `BinaryIO(IO[bytes])` represent the types of I/O streams such as returned by *open()*. Please note that these classes are not protocols, and their interface is fairly broad.

The protocols *io.Reader* and *io.Writer* offer a simpler alternative for argument types, when only the `read()` or `write()` methods are accessed, respectively:

```
def read_and_write(reader: Reader[str], writer: Writer[bytes]):
    data = reader.read()
    writer.write(data.encode())
```

Also consider using *collections.abc.Iterable* for iterating over the lines of an input stream:

```
def read_config(stream: Iterable[str]):
    for line in stream:
        ...
```

함수와 데코레이터

typing.**cast**(*typ*, *val*)

값을 형으로 변환합니다.

값을 변경하지 않고 반환합니다. 형 검사기에서는 반환 값이 지정된 형임을 나타내지만, 실행 시간에는 의도적으로 아무것도 확인하지 않습니다 (우리는 이것이 가능한 한 빠르기를 원합니다).

typing.**assert_type**(*val*, *typ*, /)

Ask a static type checker to confirm that *val* has an inferred type of *typ*.

At runtime this does nothing: it returns the first argument unchanged with no checks or side effects, no matter the actual type of the argument.

When a static type checker encounters a call to `assert_type()`, it emits an error if the value is not of the specified type:

```
def greet(name: str) -> None:
    assert_type(name, str)  # OK, inferred type of `name` is `str`
    assert_type(name, int)  # type checker error
```

This function is useful for ensuring the type checker' s understanding of a script is in line with the developer' s intentions:

```
def complex_function(arg: object):
    # Do some complex type-narrowing logic,
    # after which we hope the inferred type will be `int`
    ...
    # Test whether the type checker correctly understands our function
    assert_type(arg, int)
```

Added in version 3.11.

typing.**assert_never**(*arg*, /)

Ask a static type checker to confirm that a line of code is unreachable.

Example:

```
def int_or_str(arg: int | str) -> None:
    match arg:
        case int():
            print("It's an int")
        case str():
            print("It's a str")
        case _ as unreachable:
            assert_never(unreachable)
```

Here, the annotations allow the type checker to infer that the last case can never execute, because `arg` is either an *`int`* or a *`str`*, and both options are covered by earlier cases.

If a type checker finds that a call to `assert_never()` is reachable, it will emit an error. For example, if the type annotation for `arg` was instead `int | str | float`, the type checker would emit an error pointing out that `unreachable` is of type *`float`*. For a call to `assert_never` to pass type checking, the inferred type of the argument passed in must be the bottom type, *`Never`*, and nothing else.

At runtime, this throws an exception when called.

> **더 보기**
>
> Unreachable Code and Exhaustiveness Checking has more information about exhaustiveness checking with static typing.

Added in version 3.11.

typing.**reveal_type**(*obj*, /)

Ask a static type checker to reveal the inferred type of an expression.

When a static type checker encounters a call to this function, it emits a diagnostic with the inferred type of the argument. For example:

```
x: int = 1
reveal_type(x)  # Revealed type is "builtins.int"
```

This can be useful when you want to debug how your type checker handles a particular piece of code.

At runtime, this function prints the runtime type of its argument to `sys.stderr` and returns the argument unchanged (allowing the call to be used within an expression):

```
x = reveal_type(1)  # prints "Runtime type is int"
print(x)  # prints "1"
```

Note that the runtime type may be different from (more or less specific than) the type statically inferred by a type checker.

Most type checkers support `reveal_type()` anywhere, even if the name is not imported from `typing`. Importing the name from `typing`, however, allows your code to run without runtime errors and communicates intent more clearly.

Added in version 3.11.

@typing.**dataclass_transform**(*, *eq_default=True*, *order_default=False*, *kw_only_default=False*, *frozen_default=False*, *field_specifiers=()*, ***kwargs*)

Decorator to mark an object as providing `dataclass`-like behavior.

`dataclass_transform` may be used to decorate a class, metaclass, or a function that is itself a decorator. The presence of `@dataclass_transform()` tells a static type checker that the decorated object performs runtime "magic" that transforms a class in a similar way to `@dataclasses.dataclass`.

Example usage with a decorator function:

```
@dataclass_transform()
def create_model[T](cls: type[T]) -> type[T]:
    ...
    return cls

@create_model
class CustomerModel:
    id: int
    name: str
```

On a base class:

```
@dataclass_transform()
class ModelBase: ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

On a metaclass:

```
@dataclass_transform()
class ModelMeta(type): ...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
class ModelBase(metaclass=ModelMeta): ...

class CustomerModel(ModelBase):
    id: int
    name: str
```

The `CustomerModel` classes defined above will be treated by type checkers similarly to classes created with *@dataclasses.dataclass*. For example, type checkers will assume these classes have `__init__` methods that accept `id` and `name`.

The decorated class, metaclass, or function may accept the following bool arguments which type checkers will assume have the same effect as they would have on the *@dataclasses.dataclass* decorator: `init`, `eq`, `order`, `unsafe_hash`, `frozen`, `match_args`, `kw_only`, and `slots`. It must be possible for the value of these arguments (`True` or `False`) to be statically evaluated.

The arguments to the `dataclass_transform` decorator can be used to customize the default behaviors of the decorated class, metaclass, or function:

매개변수

- **eq_default** (bool) – Indicates whether the `eq` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `True`.
- **order_default** (bool) – Indicates whether the `order` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.
- **kw_only_default** (bool) – Indicates whether the `kw_only` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.
- **frozen_default** (bool) – Indicates whether the `frozen` parameter is assumed to be `True` or `False` if it is omitted by the caller. Defaults to `False`.

  Added in version 3.12.
- **field_specifiers** (tuple[Callable[..., Any], ...]) – Specifies a static list of supported classes or functions that describe fields, similar to *dataclasses.field()*. Defaults to `()`.
- ****kwargs** (*Any*) – Arbitrary other keyword arguments are accepted in order to allow for possible future extensions.

Type checkers recognize the following optional parameters on field specifiers:

표 1: **Recognised parameters for field specifiers**

Parameter name	Description
`init`	Indicates whether the field should be included in the synthesized `__init__` method. If unspecified, `init` defaults to `True`.
`default`	Provides the default value for the field.
`default_factory`	Provides a runtime callback that returns the default value for the field. If neither `default` nor `default_factory` are specified, the field is assumed to have no default value and must be provided a value when the class is instantiated.
`factory`	An alias for the `default_factory` parameter on field specifiers.
`kw_only`	Indicates whether the field should be marked as keyword-only. If `True`, the field will be keyword-only. If `False`, it will not be keyword-only. If unspecified, the value of the `kw_only` parameter on the object decorated with `dataclass_transform` will be used, or if that is unspecified, the value of `kw_only_default` on `dataclass_transform` will be used.
`alias`	Provides an alternative name for the field. This alternative name is used in the synthesized `__init__` method.

At runtime, this decorator records its arguments in the `__dataclass_transform__` attribute on the decorated object. It has no other runtime effect.

See **PEP 681** for more details.

Added in version 3.11.

`@typing.overload`

Decorator for creating overloaded functions and methods.

The `@overload` decorator allows describing functions and methods that support multiple different combinations of argument types. A series of `@overload`-decorated definitions must be followed by exactly one non-`@overload`-decorated definition (for the same function/method).

`@overload`-decorated definitions are for the benefit of the type checker only, since they will be overwritten by the non-`@overload`-decorated definition. The non-`@overload`-decorated definition, meanwhile, will be used at runtime but should be ignored by a type checker. At runtime, calling an `@overload`-decorated function directly will raise `NotImplementedError`.

An example of overload that gives a more precise type than can be expressed using a union or a type variable:

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    ...  # actual implementation goes here
```

See **PEP 484** for more details and comparison with other typing semantics.

버전 3.11에서 변경: Overloaded functions can now be introspected at runtime using `get_overloads()`.

`typing.get_overloads`(*func*)

Return a sequence of `@overload`-decorated definitions for *func*.

func is the function object for the implementation of the overloaded function. For example, given the definition of `process` in the documentation for `@overload`, `get_overloads(process)` will return a sequence of three function objects for the three defined overloads. If called on a function with no overloads, `get_overloads()` returns an empty sequence.

`get_overloads()` can be used for introspecting an overloaded function at runtime.

Added in version 3.11.

`typing.clear_overloads`()

Clear all registered overloads in the internal registry.

This can be used to reclaim the memory used by the registry.

Added in version 3.11.

`@typing.final`

Decorator to indicate final methods and final classes.

Decorating a method with `@final` indicates to a type checker that the method cannot be overridden in a subclass. Decorating a class with `@final` indicates that it cannot be subclassed.

예를 들면:

```
class Base:
    @final
    def done(self) -> None:
        ...
class Sub(Base):
    def done(self) -> None:  # Error reported by type checker
        ...

@final
class Leaf:
    ...
class Other(Leaf):  # Error reported by type checker
    ...
```

이러한 속성에 대한 실행 시간 검사는 없습니다. 자세한 내용은 **PEP 591**을 참조하십시오.

Added in version 3.8.

버전 3.11에서 변경: The decorator will now attempt to set a `__final__` attribute to `True` on the decorated object. Thus, a check like `if getattr(obj, "__final__", False)` can be used at runtime to determine whether an object `obj` has been marked as final. If the decorated object does not support setting attributes, the decorator returns the object unchanged without raising an exception.

`@typing.`**`no_type_check`**

어노테이션이 형 힌트가 아님을 나타내는 데코레이터.

This works as a class or function *decorator*. With a class, it applies recursively to all methods and classes defined in that class (but not to methods defined in its superclasses or subclasses). Type checkers will ignore all annotations in a function or class with this decorator.

`@no_type_check` mutates the decorated object in place.

`@typing.`**`no_type_check_decorator`**

다른 데코레이터에 *`no_type_check()`* 효과를 주는 데코레이터.

이것은 데코레이트 된 함수를 *`no_type_check()`*로 감싸는 무언가로 데코레이터를 감쌉니다.

Deprecated since version 3.13, will be removed in version 3.15: No type checker ever added support for `@no_type_check_decorator`. It is therefore deprecated, and will be removed in Python 3.15.

`@typing.`**`override`**

Decorator to indicate that a method in a subclass is intended to override a method or attribute in a superclass.

Type checkers should emit an error if a method decorated with `@override` does not, in fact, override anything. This helps prevent bugs that may occur when a base class is changed without an equivalent change to a child class.

For example:

```
class Base:
    def log_status(self) -> None:
        ...

class Sub(Base):
    @override
    def log_status(self) -> None:  # Okay: overrides Base.log_status
        ...

    @override
    def done(self) -> None:  # Error reported by type checker
        ...
```

There is no runtime checking of this property.

The decorator will attempt to set an `__override__` attribute to `True` on the decorated object. Thus, a check like `if getattr(obj, "__override__", False)` can be used at runtime to determine whether an object `obj` has been marked as an override. If the decorated object does not support setting attributes, the decorator returns the object unchanged without raising an exception.

See **PEP 698** for more details.

Added in version 3.12.

`@typing.type_check_only`

Decorator to mark a class or function as unavailable at runtime.

이 데코레이터 자체는 실행 시간에 사용할 수 없습니다. 주로, 구현이 비공개 클래스의 인스턴스를 반환할 때, 형 스텁 파일에 정의된 클래스를 표시하기 위한 용도입니다:

```
@type_check_only
class Response:  # private or not available at runtime
    code: int
    def get_header(self, name: str) -> str: ...

def fetch_response() -> Response: ...
```

비공개 클래스의 인스턴스를 반환하는 것은 좋지 않음에 유의하십시오. 일반적으로 그러한 클래스를 공개로 만드는 것이 바람직합니다.

인트로스펙션 도우미

`typing.get_type_hints`(*obj*, *globalns=None*, *localns=None*, *include_extras=False*)

함수, 메서드, 모듈 또는 클래스 객체에 대한 형 힌트가 포함된 딕셔너리를 반환합니다.

This is often the same as `obj.__annotations__`, but this function makes the following changes to the annotations dictionary:

- Forward references encoded as string literals or `ForwardRef` objects are handled by evaluating them in *globalns*, *localns*, and (where applicable) *obj*' s type parameter namespace. If *globalns* or *localns* is not given, appropriate namespace dictionaries are inferred from *obj*.
- `None` is replaced with `types.NoneType`.
- If `@no_type_check` has been applied to *obj*, an empty dictionary is returned.
- If *obj* is a class `C`, the function returns a dictionary that merges annotations from `C`' s base classes with those on `C` directly. This is done by traversing `C.__mro__` and iteratively combining `__annotations__` dictionaries. Annotations on classes appearing earlier in the *method resolution order* always take precedence over annotations on classes appearing later in the method resolution order.
- The function recursively replaces all occurrences of `Annotated[T, ...]` with `T`, unless *include_extras* is set to `True` (see `Annotated` for more information).

See also `annotationlib.get_annotations()`, a lower-level function that returns annotations more directly.

> **조심**
>
> This function may execute arbitrary code contained in annotations. See *Security implications of introspecting annotations* for more information.

> **참고**
>
> If any forward references in the annotations of *obj* are not resolvable or are not valid Python code, this function will raise an exception such as `NameError`. For example, this can happen with imported *type aliases* that include forward references, or with names imported under `if TYPE_CHECKING`.

버전 3.9에서 변경: Added `include_extras` parameter as part of **PEP 593**. See the documentation on `Annotated` for more information.

버전 3.11에서 변경: Previously, `Optional[t]` was added for function and method annotations if a default value equal to `None` was set. Now the annotation is returned unchanged.

typing.**get_origin**(*tp*)

Get the unsubscripted version of a type: for a typing object of the form `X[Y, Z, ...]` return `X`.

If `X` is a typing-module alias for a builtin or `collections` class, it will be normalized to the original class. If `X` is an instance of `ParamSpecArgs` or `ParamSpecKwargs`, return the underlying `ParamSpec`. Return `None` for unsupported objects.

Examples:

```
assert get_origin(str) is None
assert get_origin(Dict[str, int]) is dict
assert get_origin(Union[int, str]) is Union
assert get_origin(Annotated[str, "metadata"]) is Annotated
P = ParamSpec('P')
assert get_origin(P.args) is P
assert get_origin(P.kwargs) is P
```

Added in version 3.8.

typing.**get_args**(*tp*)

Get type arguments with all substitutions performed: for a typing object of the form `X[Y, Z, ...]` return `(Y, Z, ...)`.

If `X` is a union or `Literal` contained in another generic type, the order of `(Y, Z, ...)` may be different from the order of the original arguments `[Y, Z, ...]` due to type caching. Return `()` for unsupported objects.

Examples:

```
assert get_args(int) == ()
assert get_args(Dict[int, str]) == (int, str)
assert get_args(Union[int, str]) == (int, str)
```

Added in version 3.8.

typing.**get_protocol_members**(*tp*)

Return the set of members defined in a `Protocol`.

```
>>> from typing import Protocol, get_protocol_members
>>> class P(Protocol):
...     def a(self) -> str: ...
...     b: int
>>> get_protocol_members(P) == frozenset({'a', 'b'})
True
```

Raise `TypeError` for arguments that are not Protocols.

Added in version 3.13.

typing.**is_protocol**(*tp*)

Determine if a type is a `Protocol`.

예를 들면:

```
class P(Protocol):
    def a(self) -> str: ...
    b: int

is_protocol(P)     # => True
is_protocol(int)   # => False
```

Added in version 3.13.

typing.**is_typeddict**(*tp*)

Check if a type is a `TypedDict`.

For example:

```
class Film(TypedDict):
    title: str
    year: int

assert is_typeddict(Film)
assert not is_typeddict(list | str)

# TypedDict is a factory for creating typed dicts,
# not a typed dict itself
assert not is_typeddict(TypedDict)
```

Added in version 3.10.

class typing.**ForwardRef**

Class used for internal typing representation of string forward references.

For example, `List["SomeClass"]` is implicitly transformed into `List[ForwardRef("SomeClass")]`. `ForwardRef` should not be instantiated by a user, but may be used by introspection tools.

> **참고**
>
> **PEP 585** generic types such as `list["SomeClass"]` will not be implicitly transformed into `list[ForwardRef("SomeClass")]` and thus will not automatically resolve to `list[SomeClass]`.

Added in version 3.7.4.

버전 3.14에서 변경: This is now an alias for `annotationlib.ForwardRef`. Several undocumented behaviors of this class have been changed; for example, after a `ForwardRef` has been evaluated, the evaluated value is no longer cached.

typing.**evaluate_forward_ref**(*forward_ref*, *, *owner=None*, *globals=None*, *locals=None*, *type_params=None*, *format=annotationlib.Format.VALUE*)

Evaluate an `annotationlib.ForwardRef` as a *type hint*.

This is similar to calling `annotationlib.ForwardRef.evaluate()`, but unlike that method, `evaluate_forward_ref()` also recursively evaluates forward references nested within the type hint.

See the documentation for `annotationlib.ForwardRef.evaluate()` for the meaning of the *owner*, *globals*, *locals*, *type_params*, and *format* parameters.

> **조심**
>
> This function may execute arbitrary code contained in annotations. See *Security implications of introspecting annotations* for more information.

Added in version 3.14.

`typing.NoDefault`

A sentinel object used to indicate that a type parameter has no default value. For example:

```
>>> T = TypeVar("T")
>>> T.__default__ is typing.NoDefault
True
>>> S = TypeVar("S", default=None)
>>> S.__default__ is None
True
```

Added in version 3.13.

상수

`typing.TYPE_CHECKING`

A special constant that is assumed to be `True` by 3rd party static type checkers. It' s `False` at runtime.

A module which is expensive to import, and which only contain types used for typing annotations, can be safely imported inside an `if TYPE_CHECKING:` block. This prevents the module from actually being imported at runtime; annotations aren' t eagerly evaluated (see **PEP 649**) so using undefined symbols in annotations is harmless–as long as you don' t later examine them. Your static type analysis tool will set `TYPE_CHECKING` to `True` during static type analysis, which means the module will be imported and the types will be checked properly during such analysis.

용법:

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: expensive_mod.SomeType) -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

If you occasionally need to examine type annotations at runtime which may contain undefined symbols, use *`annotationlib.get_annotations()`* with a `format` parameter of *`annotationlib.Format.STRING`* or *`annotationlib.Format.FORWARDREF`* to safely retrieve the annotations without raising *`NameError`*.

Added in version 3.5.2.

Deprecated aliases

This module defines several deprecated aliases to pre-existing standard library classes. These were originally included in the `typing` module in order to support parameterizing these generic classes using `[]`. However, the aliases became redundant in Python 3.9 when the corresponding pre-existing classes were enhanced to support `[]` (see **PEP 585**).

The redundant types are deprecated as of Python 3.9. However, while the aliases may be removed at some point, removal of these aliases is not currently planned. As such, no deprecation warnings are currently issued by the interpreter for these aliases.

If at some point it is decided to remove these deprecated aliases, a deprecation warning will be issued by the interpreter for at least two releases prior to removal. The aliases are guaranteed to remain in the `typing` module without deprecation warnings until at least Python 3.14.

Type checkers are encouraged to flag uses of the deprecated types if the program they are checking targets a minimum Python version of 3.9 or newer.

Aliases to built-in types

class typing.**Dict**(*dict, MutableMapping[KT, VT]*)

Deprecated alias to `dict`.

Note that to annotate arguments, it is preferred to use an abstract collection type such as `Mapping` rather than to use `dict` or `typing.Dict`.

버전 3.9부터 폐지됨: `builtins.dict` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**List**(*list, MutableSequence[T]*)

Deprecated alias to `list`.

Note that to annotate arguments, it is preferred to use an abstract collection type such as `Sequence` or `Iterable` rather than to use `list` or `typing.List`.

버전 3.9부터 폐지됨: `builtins.list` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**Set**(*set, MutableSet[T]*)

Deprecated alias to `builtins.set`.

Note that to annotate arguments, it is preferred to use an abstract collection type such as `collections.abc.Set` rather than to use `set` or `typing.Set`.

버전 3.9부터 폐지됨: `builtins.set` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**FrozenSet**(*frozenset, AbstractSet[T_co]*)

Deprecated alias to `builtins.frozenset`.

버전 3.9부터 폐지됨: `builtins.frozenset` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

typing.**Tuple**

Deprecated alias for `tuple`.

`tuple` and `Tuple` are special-cased in the type system; see *Annotating tuples* for more details.

버전 3.9부터 폐지됨: `builtins.tuple` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**Type**(*Generic[CT_co]*)

Deprecated alias to `type`.

See *The type of class objects* for details on using `type` or `typing.Type` in type annotations.

Added in version 3.5.2.

버전 3.9부터 폐지됨: `builtins.type` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

Aliases to types in `collections`

class typing.**DefaultDict**(*collections.defaultdict, MutableMapping[KT, VT]*)

Deprecated alias to `collections.defaultdict`.

Added in version 3.5.2.

버전 3.9부터 폐지됨: `collections.defaultdict` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**OrderedDict**(*collections.OrderedDict, MutableMapping[KT, VT]*)

Deprecated alias to `collections.OrderedDict`.

Added in version 3.7.2.

버전 3.9부터 폐지됨: `collections.OrderedDict` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**ChainMap**(*collections.ChainMap, MutableMapping[KT, VT]*)

Deprecated alias to `collections.ChainMap`.

Added in version 3.6.1.

버전 3.9부터 폐지됨: `collections.ChainMap` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**Counter**(*collections.Counter, Dict[T, int]*)

Deprecated alias to `collections.Counter`.

Added in version 3.6.1.

버전 3.9부터 폐지됨: `collections.Counter` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**Deque**(*deque, MutableSequence[T]*)

Deprecated alias to `collections.deque`.

Added in version 3.6.1.

버전 3.9부터 폐지됨: `collections.deque` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

Aliases to other concrete types

class typing.**Pattern**

class typing.**Match**

Deprecated aliases corresponding to the return types from `re.compile()` and `re.match()`.

These types (and the corresponding functions) are generic over `AnyStr`. `Pattern` can be specialised as `Pattern[str]` or `Pattern[bytes]`; `Match` can be specialised as `Match[str]` or `Match[bytes]`.

버전 3.9부터 폐지됨: `re`의 클래스 `Pattern`과 `Match`는 이제 `[]`를 지원합니다. **PEP 585**와 제네릭 에일리어스 형을 참조하십시오.

class typing.**Text**

Deprecated alias for `str`.

`Text` is provided to supply a forward compatible path for Python 2 code: in Python 2, `Text` is an alias for `unicode`.

`Text`를 사용하여 값이 파이썬 2와 파이썬 3 모두와 호환되는 방식으로 유니코드 문자열을 포함해야 함을 나타내십시오:

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

Added in version 3.5.2.

버전 3.11부터 폐지됨: Python 2 is no longer supported, and most type checkers also no longer support type checking Python 2 code. Removal of the alias is not currently planned, but users are encouraged to use `str` instead of `Text`.

Aliases to container ABCs in `collections.abc`

class `typing.`**`AbstractSet`**(*Collection*[*T_co*])

Deprecated alias to `collections.abc.Set`.

버전 3.9부터 폐지됨: `collections.abc.Set` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class `typing.`**`ByteString`**(*Sequence*[*int*])

Deprecated alias to `collections.abc.ByteString`.

Use `isinstance(obj, collections.abc.Buffer)` to test if `obj` implements the buffer protocol at runtime. For use in type annotations, either use `Buffer` or a union that explicitly specifies the types your code supports (e.g., `bytes | bytearray | memoryview`).

`ByteString` was originally intended to be an abstract class that would serve as a supertype of both `bytes` and `bytearray`. However, since the ABC never had any methods, knowing that an object was an instance of `ByteString` never actually told you anything useful about the object. Other common buffer types such as `memoryview` were also never understood as subtypes of `ByteString` (either at runtime or by static type checkers).

See **PEP 688** for more details.

Deprecated since version 3.9, will be removed in version 3.17.

class `typing.`**`Collection`**(*Sized, Iterable*[*T_co*], *Container*[*T_co*])

Deprecated alias to `collections.abc.Collection`.

Added in version 3.6.

버전 3.9부터 폐지됨: `collections.abc.Collection` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class `typing.`**`Container`**(*Generic*[*T_co*])

Deprecated alias to `collections.abc.Container`.

버전 3.9부터 폐지됨: `collections.abc.Container` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class `typing.`**`ItemsView`**(*MappingView, AbstractSet*[*tuple*[*KT_co, VT_co*]])

Deprecated alias to `collections.abc.ItemsView`.

버전 3.9부터 폐지됨: `collections.abc.ItemsView` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class `typing.`**`KeysView`**(*MappingView, AbstractSet*[*KT_co*])

Deprecated alias to `collections.abc.KeysView`.

버전 3.9부터 폐지됨: `collections.abc.KeysView` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class `typing.`**`Mapping`**(*Collection*[*KT*], *Generic*[*KT, VT_co*])

Deprecated alias to `collections.abc.Mapping`.

버전 3.9부터 폐지됨: `collections.abc.Mapping` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class `typing.`**`MappingView`**(*Sized*)

Deprecated alias to `collections.abc.MappingView`.

버전 3.9부터 폐지됨: `collections.abc.MappingView` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**MutableMapping**(*Mapping*[*KT, VT*])

Deprecated alias to `collections.abc.MutableMapping`.

버전 3.9부터 폐지됨: `collections.abc.MutableMapping` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**MutableSequence**(*Sequence*[*T*])

Deprecated alias to `collections.abc.MutableSequence`.

버전 3.9부터 폐지됨: `collections.abc.MutableSequence` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**MutableSet**(*AbstractSet*[*T*])

Deprecated alias to `collections.abc.MutableSet`.

버전 3.9부터 폐지됨: `collections.abc.MutableSet` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**Sequence**(*Reversible*[*T_co*], *Collection*[*T_co*])

Deprecated alias to `collections.abc.Sequence`.

버전 3.9부터 폐지됨: `collections.abc.Sequence` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**ValuesView**(*MappingView, Collection*[*_VT_co*])

Deprecated alias to `collections.abc.ValuesView`.

버전 3.9부터 폐지됨: `collections.abc.ValuesView` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

Aliases to asynchronous ABCs in `collections.abc`

class typing.**Coroutine**(*Awaitable*[*ReturnType*], *Generic*[*YieldType, SendType, ReturnType*])

Deprecated alias to `collections.abc.Coroutine`.

See *Annotating generators and coroutines* for details on using `collections.abc.Coroutine` and `typing.Coroutine` in type annotations.

Added in version 3.5.3.

버전 3.9부터 폐지됨: `collections.abc.Coroutine` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**AsyncGenerator**(*AsyncIterator*[*YieldType*], *Generic*[*YieldType, SendType*])

Deprecated alias to `collections.abc.AsyncGenerator`.

See *Annotating generators and coroutines* for details on using `collections.abc.AsyncGenerator` and `typing.AsyncGenerator` in type annotations.

Added in version 3.6.1.

버전 3.9부터 폐지됨: `collections.abc.AsyncGenerator` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

버전 3.13에서 변경: The `SendType` parameter now has a default.

class typing.**AsyncIterable**(*Generic*[*T_co*])

Deprecated alias to `collections.abc.AsyncIterable`.

Added in version 3.5.2.

버전 3.9부터 폐지됨: `collections.abc.AsyncIterable` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**AsyncIterator**(*AsyncIterable*[*T_co*])

Deprecated alias to `collections.abc.AsyncIterator`.

Added in version 3.5.2.

버전 3.9부터 폐지됨: `collections.abc.AsyncIterator` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**Awaitable**(*Generic*[*T_co*])

Deprecated alias to `collections.abc.Awaitable`.

Added in version 3.5.2.

버전 3.9부터 폐지됨: `collections.abc.Awaitable` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

Aliases to other ABCs in `collections.abc`

class typing.**Iterable**(*Generic*[*T_co*])

Deprecated alias to `collections.abc.Iterable`.

버전 3.9부터 폐지됨: `collections.abc.Iterable` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**Iterator**(*Iterable*[*T_co*])

Deprecated alias to `collections.abc.Iterator`.

버전 3.9부터 폐지됨: `collections.abc.Iterator` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

typing.**Callable**

Deprecated alias to `collections.abc.Callable`.

See *Annotating callable objects* for details on how to use `collections.abc.Callable` and `typing.Callable` in type annotations.

버전 3.9부터 폐지됨: `collections.abc.Callable` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

버전 3.10에서 변경: `Callable` now supports `ParamSpec` and `Concatenate`. See **PEP 612** for more details.

class typing.**Generator**(*Iterator*[*YieldType*], *Generic*[*YieldType, SendType, ReturnType*])

Deprecated alias to `collections.abc.Generator`.

See *Annotating generators and coroutines* for details on using `collections.abc.Generator` and `typing.Generator` in type annotations.

버전 3.9부터 폐지됨: `collections.abc.Generator` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

버전 3.13에서 변경: Default values for the send and return types were added.

class typing.**Hashable**

Deprecated alias to `collections.abc.Hashable`.

버전 3.12부터 폐지됨: Use `collections.abc.Hashable` directly instead.

class typing.**Reversible**(*Iterable*[*T_co*])

Deprecated alias to `collections.abc.Reversible`.

버전 3.9부터 폐지됨: `collections.abc.Reversible` now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

class typing.**Sized**

Deprecated alias to *collections.abc.Sized*.

버전 3.12부터 폐지됨: Use *collections.abc.Sized* directly instead.

Aliases to contextlib ABCs

class typing.**ContextManager**(*Generic*[*T_co, ExitT_co*])

Deprecated alias to *contextlib.AbstractContextManager*.

The first type parameter, `T_co`, represents the type returned by the `__enter__()` method. The optional second type parameter, `ExitT_co`, which defaults to `bool | None`, represents the type returned by the `__exit__()` method.

Added in version 3.5.4.

버전 3.9부터 폐지됨: *contextlib.AbstractContextManager* now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

버전 3.13에서 변경: Added the optional second type parameter, `ExitT_co`.

class typing.**AsyncContextManager**(*Generic*[*T_co, AExitT_co*])

Deprecated alias to *contextlib.AbstractAsyncContextManager*.

The first type parameter, `T_co`, represents the type returned by the `__aenter__()` method. The optional second type parameter, `AExitT_co`, which defaults to `bool | None`, represents the type returned by the `__aexit__()` method.

Added in version 3.6.2.

버전 3.9부터 폐지됨: *contextlib.AbstractAsyncContextManager* now supports subscripting (`[]`). See **PEP 585** and 제네릭 에일리어스 형.

버전 3.13에서 변경: Added the optional second type parameter, `AExitT_co`.

26.1.13 Deprecation Timeline of Major Features

Certain features in `typing` are deprecated and may be removed in a future version of Python. The following table summarizes major deprecations for your convenience. This is subject to change, and not all deprecations are listed.

Feature	Deprecated in	Projected removal	PEP/ issue
`typing` versions of standard collections	3.9	Undecided (see *Deprecated aliases* for more information)	**PEP 585**
typing.ByteString	3.9	3.17	gh-91896
typing.Text	3.11	Undecided	gh-92332
typing.Hashable and *typing.Sized*	3.12	Undecided	gh-94309
typing.TypeAlias	3.12	Undecided	**PEP 695**
@typing.no_type_check_decorator	3.13	3.15	gh-106309
typing.AnyStr	3.13	3.18	gh-105578

26.2 pydoc — 설명서 생성과 온라인 도움말 시스템

소스 코드: Lib/pydoc.py

---

pydoc 모듈은 파이썬 모듈로부터 자동으로 설명서를 생성합니다. 설명서는 콘솔에 텍스트 페이지로 표시하거나, 웹 브라우저로 제공하거나, HTML 파일에 저장할 수 있습니다.

모듈, 클래스, 함수 및 메서드의 경우, 표시된 설명서는 객체와 재귀적으로 그것의 설명 가능한 멤버들의 독스트링(즉, *__doc__* 어트리뷰트)에서 파생됩니다. 독스트링이 없으면, pydoc은 소스 파일의 클래스, 함수 또는 메서드의 정의 바로 위나 모듈의 맨 위에 있는 주석 줄 블록에서 설명을 얻으려고 합니다(*inspect.getcomments()*를 참조하십시오).

내장 함수 *help()*는 대화형 인터프리터에서 온라인 도움말 시스템을 호출합니다. 이 시스템은 pydoc을 사용하여 설명서를 콘솔에 텍스트로 생성합니다. 같은 텍스트 설명서는 운영 체제의 명령 프롬프트에서 **pydoc**을 스크립트로 실행하여 파이썬 인터프리터 외부에서도 볼 수 있습니다. 예를 들어,

```
python -m pydoc sys
```

를 셸 프롬프트에서 실행하면, *sys* 모듈의 설명서를 유닉스 **man** 명령으로 표시되는 매뉴얼 페이지와 비슷한 스타일로 표시합니다. **pydoc**의 인자는 함수, 모듈 또는 패키지의 이름이거나 모듈이나 패키지의 모듈 내의 클래스, 메서드 또는 함수에 대한 점으로 구분된 참조일 수 있습니다. **pydoc**에 대한 인자가 경로처럼 보이고 (즉, 유닉스의 슬래시와 같이 운영 체제의 경로 구분 기호가 포함되어 있으면), 기존 파이썬 소스 파일을 가리키면, 해당 파일에 대한 설명서가 생성됩니다.

> **참고**
>
> 객체와 그들의 설명을 찾기 위해, pydoc은 설명할 모듈을 임포트 합니다. 따라서, 모듈 수준의 모든 코드는 이때 실행됩니다. 파일이 임포트될 때가 아니라 스크립트로 호출할 때만 실행되도록 코드를 보호하려면 `if __name__ == '__main__':`를 사용하십시오.

출력을 콘솔에 인쇄할 때, **pydoc**은 읽기 쉽도록 출력을 페이지로 나누려고 합니다. MANPAGER나 PAGER 환경 변수가 설정되면, **pydoc**은 그 값을 페이지 매김 프로그램으로 사용합니다. 둘 다 설정되면 MANPAGER가 사용됩니다.

인자 앞에 `-w` 플래그를 지정하면 콘솔에 텍스트를 표시하는 대신, HTML 설명서가 현재 디렉터리에 파일로 기록됩니다.

인자 앞에 `-k` 플래그를 지정하면 인자로 주어진 키워드에 대해 사용 가능한 모든 모듈의 개요 행을 검색할 수 있습니다. 역시 유닉스 **man** 명령과 유사한 방식입니다. 모듈의 개요 행은 설명서 문자열의 첫 행입니다.

pydoc을 사용하여 방문하는 웹 브라우저에 설명서를 제공하는 HTTP 서버를 로컬 시스템에서 시작할 수도 있습니다. **python -m pydoc -p 1234**는 포트 1234에서 HTTP 서버를 시작해서, 여러분이 선호하는 웹 브라우저에서 `http://localhost:1234/`의 설명서를 볼 수 있도록 합니다. 포트 번호로 `0`을 지정하면 사용되지 않는 임의의 포트가 선택됩니다.

python -m pydoc -n <hostname>은 주어진 호스트 이름에서 리스닝하는 서버를 시작합니다. 기본적으로 호스트 이름은 'localhost' 이지만 다른 컴퓨터에서 서버에 도달하도록 하고 싶으면 서버가 응답하는 호스트 이름을 변경해야 할 수 있습니다. 개발 중에 컨테이너 내에서 pydoc을 실행하려는 경우 특히 유용합니다.

python -m pydoc -b는 서버를 시작하고 모듈 색인 페이지로 웹 브라우저를 추가로 엽니다. 제공되는 각 페이지에는 상단에 탐색 바가 있어, 개별 항목에 대한 도움말을 조회(*Get*)하고, 개요 행에 키워드가 있는 모든 모듈을 검색(*Search*)하고, 모듈 색인(*Module index*), 주제(*Topics*) 및 키워드(*Keywords*) 페이지로 이동할 수 있습니다.

pydoc이 설명서를 생성할 때, 현재 환경과 경로를 사용하여 모듈을 찾습니다. 따라서, **pydoc spam**을 호출하면 정확히 파이썬 인터프리터를 시작하고 `import spam`을 입력할 때 얻는 모듈의 버전을 설명합니다.

코어 모듈에 대한 모듈 설명은 `https://docs.python.org/X.Y/library/`에 있다고 가정합니다. 여기서 `X`와 `Y`는 파이썬 인터프리터의 주 버전 번호와 부 버전 번호입니다. 이는 PYTHONDOCS 환경 변수를 다른

URL로 설정하거나 라이브러리 레퍼런스 매뉴얼 페이지가 있는 로컬 디렉터리로 설정하여 재정의할 수 있습니다.

버전 3.2에서 변경: `-b` 옵션이 추가되었습니다.

버전 3.3에서 변경: `-g` 명령 줄 옵션이 제거되었습니다.

버전 3.4에서 변경: 이제 `pydoc`은 *inspect.getfullargspec()* 대신 *inspect.signature()*를 사용하여 콜러블에서 서명 정보를 추출합니다.

버전 3.7에서 변경: `-n` 옵션이 추가되었습니다.

26.3 파이썬 개발 모드

Added in version 3.7.

파이썬 개발 모드에는 기본적으로 활성화하기에 너무 비싼 추가 실행 시간 검사를 도입합니다. 코드가 올바르면 기본값보다 더 상세하지(verbose) 않아야 합니다; 새로운 경고는 문제가 감지될 때만 발생합니다.

`-X dev` 명령 줄 옵션을 사용하거나 `PYTHONDEVMODE` 환경 변수를 `1`로 설정하여 활성화할 수 있습니다.

See also Python debug build.

26.3.1 파이썬 개발 모드의 효과

파이썬 개발 모드를 활성화하는 것은 다음 명령과 유사하지만, 아래에 설명된 추가 효과가 있습니다:

```
PYTHONMALLOC=debug PYTHONASYNCIODEBUG=1 python -W default -X faulthandler
```

파이썬 개발 모드의 효과:

- `default` 경고 필터를 추가합니다. 다음과 같은 경고가 표시됩니다:
  - *DeprecationWarning*
  - *ImportWarning*
  - *PendingDeprecationWarning*
  - *ResourceWarning*

  일반적으로, 위의 경고는 기본 경고 필터가 필터링합니다.

  `-W default` 명령 줄 옵션이 사용된 것처럼 작동합니다.

  경고를 에러로 처리하려면 `-W error` 명령 줄 옵션을 사용하거나 `PYTHONWARNINGS` 환경 변수를 `error`로 설정하십시오.
- 메모리 할당자에 디버그 훅을 설치하여 다음을 확인합니다:
  - 버퍼 언더플로
  - 버퍼 오버플로
  - 메모리 할당자 API 위반
  - GIL의 안전하지 않은 사용

  `PyMem_SetupDebugHooks()` C 함수를 참조하십시오.

  `PYTHONMALLOC` 환경 변수가 `debug`로 설정된 것처럼 동작합니다.

  메모리 할당자에 디버그 훅을 설치하지 않고 파이썬 개발 모드를 사용하려면, `PYTHONMALLOC` 환경 변수를 `default`로 설정하십시오.
- 파이썬 시작 시 *faulthandler.enable()*을 호출하여 *SIGSEGV*, *SIGFPE*, *SIGABRT*, *SIGBUS* 및 *SIGILL* 시그널에 대한 처리기를 설치하여 충돌 시 파이썬 트레이스백을 덤프합니다.

  `-X faulthandler` 명령 줄 옵션이 사용되거나 `PYTHONFAULTHANDLER` 환경 변수가 `1`로 설정된 것처럼 작동합니다.

- *asyncio* 디버그 모드를 활성화합니다. 예를 들어, `asyncio`는 어웨이트 하지 않은 코루틴을 확인하고 이를 로그 합니다.

  `PYTHONASYNCIODEBUG` 환경 변수가 `1`로 설정된 것처럼 동작합니다.
- 문자열 인코딩과 디코딩 연산에 대해 *encoding*과 *errors* 인자를 확인합니다. 예: `open()`, `str.encode()` 및 `bytes.decode()`.

  기본적으로, 최상의 성능을 위해, *errors* 인자는 첫 번째 인코딩/디코딩 에러에서만 검사되며 빈 문자열에 대해서는 *encoding* 인자가 무시되는 경우가 있습니다.
- `io.IOBase` 파괴자는 `close()` 예외를 로그 합니다.
- `sys.flags`의 `dev_mode` 어트리뷰트를 `True`로 설정합니다.

파이썬 개발 모드는 (성능과 메모리에 대한) 오버헤드 비용이 너무 비싸서, 기본적으로 `tracemalloc` 모듈을 활성화하지 않습니다. `tracemalloc` 모듈을 활성화하면 일부 에러의 원인에 대한 추가 정보가 제공됩니다. 예를 들어, `ResourceWarning`은 자원이 할당된 곳의 트레이스백을 로그하고, 버퍼 오버플로 에러는 메모리 블록이 할당된 곳의 트레이스백을 로그 합니다.

파이썬 개발 모드는 `-O` 명령 줄 옵션이 `assert` 문을 제거하거나 `__debug__`를 `False`로 설정하는 것을 막지 않습니다.

The Python Development Mode can only be enabled at the Python startup. Its value can be read from `sys.flags.dev_mode`.

버전 3.8에서 변경: `io.IOBase` 파괴자는 이제 `close()` 예외를 로그 합니다.

버전 3.9에서 변경: *encoding*과 *errors* 인자는 이제 문자열 인코딩과 디코딩 연산을 검사합니다.

26.3.2 ResourceWarning 예

명령 줄에 지정된 텍스트 파일의 줄 수를 세는 스크립트의 예:

```
import sys

def main():
    fp = open(sys.argv[1])
    nlines = len(fp.readlines())
    print(nlines)
    # 파일이 묵시적으로 닫힙니다

if __name__ == "__main__":
    main()
```

스크립트는 파일을 명시적으로 닫지 않습니다. 기본적으로, 파이썬은 아무런 경고도 하지 않습니다. 269 줄이 있는 README.txt를 사용하는 예:

```
$ python script.py README.txt
269
```

파이썬 개발 모드를 사용하면 `ResourceWarning` 경고가 표시됩니다:

```
$ python -X dev script.py README.txt
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='README.rst'␣
↪mode='r' encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object allocation traceback
```

또한, `tracemalloc`을 활성화하면 파일이 열린 줄이 표시됩니다:

```
$ python -X dev -X tracemalloc=5 script.py README.rst
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='README.rst'␣
↪mode='r' encoding='UTF-8'>
  main()
Object allocated at (most recent call last):
  File "script.py", lineno 10
    main()
  File "script.py", lineno 4
    fp = open(sys.argv[1])
```

수선은 파일을 명시적으로 닫는 것입니다. 컨텍스트 관리자를 사용하는 예:

```
def main():
    # 블록에서 빠져나갈 때 파일을 명시적으로 닫습니다
    with open(sys.argv[1]) as fp:
        nlines = len(fp.readlines())
    print(nlines)
```

자원을 명시적으로 닫지 않으면 예상보다 오래 자원을 열어둘 수 있습니다; 파이썬을 종료할 때 심각한 문제가 발생할 수 있습니다. CPython에서도 나쁘지만, PyPy에서는 더 나쁩니다. 리소스를 명시적으로 닫으면 응용 프로그램을 더 결정적이고 안정적으로 만들 수 있습니다.

26.3.3 잘못된 파일 기술자 에러 예

자신의 첫 줄을 표시하는 스크립트:

```
import os

def main():
    fp = open(__file__)
    firstline = fp.readline()
    print(firstline.rstrip())
    os.close(fp.fileno())
    # 파일이 묵시적으로 닫힙니다

main()
```

기본적으로, 파이썬은 아무런 경고도 하지 않습니다:

```
$ python script.py
import os
```

파이썬 개발 모드는 *ResourceWarning*을 표시하고 파일 객체를 파이널라이즈 할 때 "잘못된 파일 기술자 (Bad file descriptor)" 에러를 로그 합니다:

```
$ python -X dev script.py
import os
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='script.py'␣
↪mode='r' encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object allocation traceback
Exception ignored in: <_io.TextIOWrapper name='script.py' mode='r' encoding='UTF-8
↪'>
Traceback (most recent call last):
  File "script.py", line 10, in <module>
    main()
OSError: [Errno 9] Bad file descriptor
```

os.close(fp.fileno()) 는 파일 기술자를 닫습니다. 파일 객체 파이널라이저가 파일 기술자를 다시 닫으려고 하면, Bad file descriptor 에러로 실패합니다. 파일 기술자는 한 번만 닫아야 합니다. 최악의 시나리오에서는, 두 번 닫을 때 충돌이 발생할 수 있습니다 (예는 bpo-18748을 참조하십시오).

수선은 os.close(fp.fileno()) 줄을 제거하거나, closefd=False로 파일을 여는 것입니다.

26.4 doctest — 대화형 파이썬 예제 테스트

소스 코드: Lib/doctest.py

---

doctest 모듈은 대화형 파이썬 세션처럼 보이는 텍스트를 검색한 다음, 해당 세션을 실행하여 표시된 대로 정확하게 작동하는지 검증합니다. doctest를 사용하는 몇 가지 일반적인 방법이 있습니다:

- 모든 대화식 예제가 설명된 대로 작동하는지 확인하여 모듈의 독스트링이 최신인지 확인합니다.
- 테스트 파일이나 테스트 객체의 대화형 예제가 예상대로 작동하는지 확인하여 회귀 테스트를 수행합니다.
- 입/출력 예제를 그대로 보여줌으로써 패키지에 대한 자습서를 작성합니다. 예제나 설명문 중 어느 것이 강조되는지에 따라, "문학적 테스트(literate testing)" 나 "실행 가능한 설명서(executable documentation)" 의 느낌을 줍니다.

여기에 완전하지만 작은 예제 모듈이 있습니다:

```
"""
이것은 "예제" 모듈입니다.

예제 모듈은 factorial() 함수 하나를 제공합니다. 예를 들어,

>>> factorial(5)
120
"""

def factorial(n):
    """n의 계승을 반환합니다, 정확한 정수 >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    float의 계승도 좋습니다만, float는 정확한 정수여야 합니다:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    또한 터무니없이 크지 않아야 합니다:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # 1e300과 같은 값을 잡아냅니다
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

example.py를 명령 줄에서 직접 실행하면, *doctest*가 마술을 부리기 시작합니다:

```
$ python example.py
$
```

출력이 없습니다! 이것이 정상이며, 모든 예제가 작동했다는 것을 뜻합니다. -v를 스크립트에 전달하면, *doctest*는 시도하고 있는 내용에 대한 자세한 로그를 출력하고 끝에 요약을 인쇄합니다.:

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

결국, 다음과 같이 끝납니다:

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 test in __main__
   6 tests in __main__.factorial
7 tests in 2 items.
7 passed.
Test passed.
$
```

이것이 *doctest*를 생산적으로 사용하기 위해서 알아야 할 모든 것입니다! 시도해 보세요. 다음 절에서는 자세한 내용을 제공합니다. 표준 파이썬 테스트 스위트와 라이브러리에는 doctest 예제가 많습니다. 특히 유용한 예제는 표준 테스트 파일 `Lib/test/test_doctest/test_doctest.py`에서 찾을 수 있습니다.

Added in version 3.13: Output is colorized by default and can be controlled using environment variables.

26.4.1 간단한 사용법: 독스트링에 있는 예제 확인하기

doctest를 사용하는 가장 간단한 방법은 (하지만 계속 이렇게 할 필요는 없습니다) 각 모듈 `M`을 다음과 같이 끝내는 것입니다:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

그러면 `doctest`는 모듈 `M`의 독스트링을 검사합니다.

모듈을 스크립트로 실행하면 독스트링의 예제가 실행되고 검증됩니다:

```
python M.py
```

예제가 실패하지 않는 한 아무것도 표시되지 않습니다, 실패하면 실패한 예제와 실패 원인이 stdout으로 출력되고, 마지막 출력 줄은 `***Test Failed*** N failures.` 입니다. 여기서 *N*은 실패한 예제의 수입니다.

대신 `-v` 스위치로 실행해 보십시오:

```
python M.py -v
```

그러면 시도한 모든 예제에 대한 자세한 보고서가 표준 출력으로 출력되고, 끝에 정돈된 요약이 붙습니다.

You can force verbose mode by passing `verbose=True` to *testmod()*, or prohibit it by passing `verbose=False`. In either of those cases, *sys.argv* is not examined by *testmod()* (so passing `-v` or not has no effect).

*testmod()*를 실행하는 명령줄 바로 가기도 있습니다, 명령줄 사용법 절을 참조하십시오.

*testmod()*에 대한 자세한 내용은, 기본 *API* 절을 참조하십시오.

26.4.2 간단한 사용법: 텍스트 파일에 있는 예제 확인하기

doctest의 또 다른 간단한 활용은 텍스트 파일에 있는 대화형 예제를 테스트하는 것입니다. 이것은 *testfile()* 함수로 수행할 수 있습니다:

```
import doctest
doctest.testfile("example.txt")
```

이 짧은 스크립트는 `example.txt` 파일에 들어있는 대화형 파이썬 예제를 실행하고 검증합니다. 파일 내용은 하나의 거대한 독스트링인 것처럼 취급됩니다; 파일이 파이썬 프로그램일 필요가 없습니다! 예를 들어, `example.txt`에 다음과 같은 것이 들어있습니다:

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from example import factorial
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Now use it:

    >>> factorial(6)
    120
```

doctest.testfile("example.txt") 를 실행하면 이 문서에 있는 에러를 찾습니다:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

As with *testmod()*, *testfile()* won' t display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to stdout, using the same format as testmod().

기본적으로, *testfile()* 은 호출하는 모듈의 디렉터리에서 파일을 찾습니다. 다른 위치에서 파일을 찾도록 지시하는 데 사용할 수 있는 선택적 인자에 대한 설명은 기본 *API* 절을 참조하십시오.

testmod() 와 마찬가지로, *testfile()* 의 상세도는 -v 명령 줄 스위치나 선택적 키워드 인자 *verbose* 를 사용하여 설정할 수 있습니다.

testfile() 을 실행하는 명령줄 바로 가기도 있습니다, 명령줄 사용법 절을 참조하십시오.

testfile() 에 대한 자세한 내용은, 기본 *API* 절을 참조하십시오.

26.4.3 명령줄 사용법

The *doctest* module can be invoked as a script from the command line:

```
python -m doctest [-v] [-o OPTION] [-f] file [file ...]
```

-v, --verbose

시도한 모든 예제에 대한 자세한 보고서가 표준 출력으로 출력되고, 끝에 정돈된 요약이 붙습니다:

```
python -m doctest -v example.py
```

이렇게 하면 example.py 를 독립 실행형 모듈로 임포트하고, *testmod()* 를 실행합니다. 파일이 패키지 일부이고 그 패키지에서 다른 서브 모듈을 임포트하면, 올바르게 작동하지 않을 수 있음에 유의하십시오.

파일 이름이 .py 로 끝나지 않으면, doctest 는 대신 *testfile()* 로 실행되어야 한다고 추론합니다:

```
python -m doctest -v example.txt
```

-o, --option <option>

Option flags control various aspects of doctest' s behavior, see section 옵션 플래그.

Added in version 3.4.

-f, --fail-fast

-o FAIL_FAST 의 축약입니다.

Added in version 3.4.

26.4.4 작동 방법

이 절에서는 doctest가 어떻게 작동하는지 자세히 설명합니다: 어떤 독스트링을 살피는지, 대화형 예제를 어떻게 찾는지, 사용하는 실행 컨텍스트는 무엇인지, 예외를 어떻게 처리하는지, 어떻게 옵션 플래그를 사용하여 동작을 제어하는지. 이것은 doctest 예제를 작성하기 위해 알아야 할 정보입니다; 이러한 예제에 대해 실제로 doctest를 실행하는 방법에 대한 자세한 내용은 다음 절을 참조하십시오.

어떤 독스트링을 검사합니까?

모듈 독스트링과 모든 함수, 클래스 및 메서드 독스트링이 검색됩니다. 모듈로 임포트 된 객체는 검색되지 않습니다. 또한, 테스트를 모듈의 일부로 포함시키고 싶지만 도움말 텍스트에는 포함시키고 싶지 않은 경우가 있습니다. 이 경우 테스트가 독스트링에 포함되지 않도록 해야 합니다. Doctest는 모듈 수준 변수인 `__test__`를 보고 이를 통해 다른 테스트를 찾아냅니다. `M.__test__`가 존재하면, 딕셔너리이어야 하고 각 항목은 (문자열) 이름을 함수 객체, 클래스 객체 또는 문자열에 매핑합니다. `M.__test__`에서 발견된 함수와 클래스 객체 독스트링이 검색되고, 문자열은 독스트링인 것처럼 처리됩니다. 출력에서, `M.__test__`의 키 `K`가 이름 `M.__test__.K`로 나타납니다.

For example, place this block of code at the top of `example.py`:

```
__test__ = {
    'numbers': """
>>> factorial(6)
720

>>> [factorial(n) for n in range(6)]
[1, 1, 2, 6, 24, 120]
"""
}
```

The value of `example.__test__["numbers"]` will be treated as a docstring and all the tests inside it will be run. It is important to note that the value can be mapped to a function, class object, or module; if so, `doctest` searches them recursively for docstrings, which are then scanned for tests.

발견된 모든 클래스는 포함된 메서드와 중첩된 클래스의 독스트링을 테스트하기 위해 유사하게 재귀적으로 검색됩니다.

> **참고**
>
> `doctest` can only automatically discover classes and functions that are defined at the module level or inside other classes.
>
> Since nested classes and functions only exist when an outer function is called, they cannot be discovered. Define them outside to make them visible.

독스트링 예제는 어떻게 인식됩니까?

대부분 대화형 콘솔 세션의 복사하여 붙여넣기가 잘 작동하지만, doctest는 특정 파이썬 셸의 정확한 에뮬레이션을 시도하지 않습니다.

```
>>> # 주석은 무시됩니다
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
no
NO
NO!!!
>>>
```

모든 예상 출력은 코드가 포함된 마지막 '>>> ' 또는 '... ' 줄 바로 다음에 나와야 하며, (있다면) 예상 출력은 다음 '>>> ' 나 전체 공백 줄까지 확장됩니다.

세부 사항:

- 예상 출력은 전체 공백 줄을 포함할 수 없습니다. 그러한 줄은 예상 출력의 끝으로 인식되기 때문입니다. 예상 출력이 빈 줄을 포함하면, doctest 예제에서 빈 줄이 나타나는 곳에 <BLANKLINE>을 넣으십시오.
- 모든 하드 탭 문자는 8열 탭 정지를 사용하여 스페이스로 확장됩니다. 테스트 된 코드에 의해 생성된 출력의 탭은 수정되지 않습니다. 샘플 출력의 모든 하드 탭이 확장되므로, 이것은 코드 출력에 하드 탭이 포함될 때 doctest가 통과할 수 있는 유일한 방법은, *NORMALIZE_WHITESPACE* 옵션이나 지시자가 유효한 경우뿐임을 의미합니다. 또는, 출력을 캡처하여 테스트 일부로 예상값과 비교하도록 테스트를 다시 작성할 수 있습니다. 이러한 소스의 탭 처리는 시행착오를 거쳐 얻어진 것이며, 가장 에러가 발생하지 않는 방법으로 입증되었습니다. 사용자 정의 *DocTestParser* 클래스를 작성하여 탭 처리에 다른 알고리즘을 사용하는 것도 가능합니다.
- stdout으로의 출력은 캡처되지만, stderr로의 출력은 그렇지 않습니다 (예외 트레이스백은 다른 수단을 통해 캡처됩니다).
- 대화식 세션에서 역 슬래시를 통해 줄을 계속하거나, 다른 이유로 백 슬래시를 사용하면, 날 독스트링 (raw docstring)을 사용해서 역 슬래시를 입력한 그대로 유지해야 합니다:

  ```
  >>> def f(x):
  ...     r'''Backslashes in a raw docstring: m\n'''
  ...
  >>> print(f.__doc__)
  Backslashes in a raw docstring: m\n
  ```

  그렇지 않으면, 백 슬래시가 문자열 일부로 해석됩니다. 예를 들어, 위의 \n은 개행 문자로 해석됩니다. 또는, doctest 버전에서 각 백 슬래시를 중복시킬 수 있습니다 (그리고 날 문자열은 사용하지 않습니다):

  ```
  >>> def f(x):
  ...     '''Backslashes in a raw docstring: m\\n'''
  ...
  >>> print(f.__doc__)
  Backslashes in a raw docstring: m\n
  ```

- 시작 열은 중요하지 않습니다:

  ```
  >>> assert "Easy!"
        >>> import math
            >>> math.floor(1.9)
            1
  ```

  그리고 예제를 시작한 초기 '>>> ' 줄에 나타나는 것만큼의 선행 공백을 예상 출력에서 제거합니다.

실행 컨텍스트란 무엇입니까?

기본적으로, *doctest*가 테스트할 독스트링을 찾을 때마다, M의 전역 이름 공간(globals)의 얕은 복사를 사용하므로, 실행 중인 테스트는 모듈의 실제 전역을 변경하지 않고, M의 한 테스트가 실수로 다른 테스트가 작동하도록 만드는 부스러기를 남기지 않습니다. 이는 예제가 M에서 최상위 수준에 정의된 이름과 실행

중인 독스트링에서 앞서 정의한 이름을 자유롭게 사용할 수 있음을 의미합니다. 예제는 다른 독스트링에 정의된 이름을 볼 수 없습니다.

대신 `globs=your_dict`를 *testmod()*나 *testfile()*로 전달하여 실행 컨텍스트로 여러분 자신의 딕셔너리를 사용하도록 할 수 있습니다.

예외는 어떻게 됩니까?

문제없습니다, 트레이스백이 예제에 의해 생성된 유일한 출력이기만 하면 됩니다: 그냥 트레이스백을 붙여넣으십시오.[1] 트레이스백에는 빠르게 변할 가능성이 있는 세부 사항(예를 들어, 정확한 파일 경로와 줄 번호)이 포함되어 있으므로, 이것은 doctest가 수락할 내용에 유연하도록 신경 써야 하는 한 가지 사례입니다.

간단한 예:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

이 doctest는 `list.remove(x): x not in list` 세부 정보를 포함하는 *ValueError*가 발생하면 성공합니다.

예외의 예상 출력은 다음 두 줄 중 한 가지가 예제의 첫 번째 줄과 같게 들여쓰기 된 트레이스백 헤더로 시작해야 합니다:

```
Traceback (most recent call last):
Traceback (innermost last):
```

트레이스백 헤더 다음에는 선택적인 트레이스백 스택이 오며, 그 내용은 doctest가 무시합니다. 보통 트레이스백 스택은 생략되거나, 대화형 세션에서 그대로 복사됩니다.

트레이스백 스택 다음에는 가장 흥미로운 부분이 옵니다: 예외 형과 세부 사항이 있는 줄. 대개 이것은 트레이스백의 마지막 줄이지만, 예외에 여러 줄로 구성된 세부 사항이 있으면 여러 줄로 확장될 수 있습니다:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

(*ValueError*로 시작하는) 마지막 세 줄이 예외의 형 및 세부 사항과 비교되고, 나머지는 무시됩니다.

모범 사례는 예제에 중요한 설명으로서의 가치를 추가하지 않는 한 트레이스백 스택을 생략하는 것입니다. 따라서 마지막 예제는 이렇게 하는 것이 더 좋습니다:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
    line
detail
```

트레이스백이 매우 특별하게 취급된다는 점에 유의하십시오. 특히, 다시 작성된 예제에서, ... 의 사용은 doctest의 *ELLIPSIS* 옵션과 무관합니다. 이 예제의 줄임표는 생략하거나, 3개(혹은 300개)의 쉼표나 숫자 또는 몬티 파이썬 쇼의 들여쓰기 된 대본이어도 똑같이 잘 동작합니다.

한 번쯤 읽어야 할 세부 정보이지만, 기억할 필요는 없습니다:

[1] 예상 출력과 예외를 모두 포함하는 예제는 지원되지 않습니다. 어디에서 하나가 끝나고 다른 하나가 시작되는지 추측하는 것은 너무 에러가 발생하기 쉽고, 이것은 또한 혼란스러운 테스트를 만들게 됩니다.

- Doctest는 예상 출력이 예외 트레이스백에서 온 것인지 일반 인쇄에서 온 것인지 추측할 수 없습니다. 그래서, 예를 들어, `ValueError: 42 is prime`을 예상하는 예제는 *ValueError*가 실제로 발생해도 통과하지만, 예제가 단지 그 트레이스백 텍스트를 출력해도 통과합니다. 실제로는, 일반 출력은 거의 트레이스백 헤더 줄로 시작하지 않으므로, 실제 문제가 되지는 않습니다.
- (있다면) 트레이스백 스택의 각 줄은 예제의 첫 번째 줄보다 더 들여쓰기 되거나, 또는 영숫자(alphanumeric)가 아닌 문자로 시작해야 합니다. 트레이스백 헤더 뒤에 같은 정도로 들여쓰기 되고, 영숫자로 시작하는 첫 번째 줄은 예외 세부 사항의 시작으로 간주합니다. 물론 이것은 진짜 트레이스백에 잘 들어맞습니다.
- *IGNORE_EXCEPTION_DETAIL* doctest 옵션을 지정하면, 가장 왼쪽 콜론 다음에 오는 모든 것과 예외 이름의 모듈 정보가 무시됩니다.
- The interactive shell omits the traceback header line for some *SyntaxError*s. But doctest uses the traceback header line to distinguish exceptions from non-exceptions. So in the rare case where you need to test a `SyntaxError` that omits the traceback header, you will need to manually add the traceback header line to your test example.
- 일부 예외의 경우, 파이썬은 ^ 마커와 물결표를 사용하여 에러의 위치를 표시합니다:

```
>>> 1 + None
  File "<stdin>", line 1
    1 + None
    ~~^~~~~~
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

  에러의 위치를 나타내는 줄은 예외 형과 세부 사항 앞에 오므로, doctest가 점검하지 않습니다. 예를 들어, ^ 마커를 잘못된 위치에 넣어도, 다음 테스트가 통과합니다:

```
>>> 1 + None
  File "<stdin>", line 1
    1 + None
    ^~~~~~~
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

옵션 플래그

많은 옵션 플래그가 doctest의 다양한 동작을 제어합니다. 플래그의 기호 이름은 모듈 상수로 제공되며, 함께 비트별 OR되어 다양한 함수로 전달될 수 있습니다. 이 이름은 *doctest 지시자*에서도 사용될 수 있으며, `-o` 옵션을 통해 doctest 명령 줄 인터페이스로 전달될 수 있습니다.

첫 번째 옵션 그룹은 테스트의 의미를 정의하는데, doctest가 실제 출력이 예제의 예상 출력과 일치하는지를 결정하는 측면을 제어합니다:

`doctest.`**DONT_ACCEPT_TRUE_FOR_1**

기본적으로, 예상 출력 블록에 `1` 만 있으면, 단지 `1`이나 `True` 만 포함된 실제 출력 블록을 일치하는 것으로 간주하며, `0`과 `False`도 유사하게 다룹니다. *DONT_ACCEPT_TRUE_FOR_1*이 지정되면, 두 치환 모두 허용되지 않습니다. 기본 동작은 파이썬이 많은 함수의 반환형을 정수에서 논릿값으로 변경했다는 것을 반영합니다; "작은 정수" 출력을 예상하는 doctest가 이러한 경우에 여전히 작동합니다. 아마도 이 옵션은 사라지게 되겠지만, 몇 년 동안은 남아있을 겁니다.

`doctest.`**DONT_ACCEPT_BLANKLINE**

기본적으로, 예상 출력 블록에 `<BLANKLINE>` 문자열만 포함된 줄이 있으면, 해당하는 줄은 실제 출력의 빈 줄과 일치합니다. 진짜 빈 줄은 예상 출력을 끝내므로, 이것이 빈 줄을 예상하는 유일한 방법입니다. *DONT_ACCEPT_BLANKLINE*이 지정되면, 이 치환은 허용되지 않습니다.

`doctest.`**NORMALIZE_WHITESPACE**

지정되면, 모든 공백(빈칸과 개행) 시퀀스는 같게 취급됩니다. 예상 출력 내의 모든 공백 시퀀스는 실제 출력 내의 모든 공백 시퀀스와 일치합니다. 기본적으로, 공백은 정확히 일치해야 합니다. *NORMALIZE_WHITESPACE*는 예상 출력 줄이 매우 길고 소스의 여러 줄에 걸쳐 줄넘김하려는 경우에 특히 유용합니다.

doctest.**ELLIPSIS**

지정되면, 예상 출력의 줄임표(...)가 실제 출력의 모든 부분 문자열과 일치 할 수 있습니다. 여기에는 줄 경계를 넘는 부분 문자열과 빈 부분 문자열이 포함되므로, 사용을 간단하게 유지하는 것이 가장 좋습니다. 복잡한 사용은 정규식에서 .*를 쓸 때처럼 "이런, 너무 많이 일치하는군!" 과 같은 상황을 만들 수 있습니다.

doctest.**IGNORE_EXCEPTION_DETAIL**

When specified, doctests expecting exceptions pass so long as an exception of the expected type is raised, even if the details (message and fully qualified exception name) don' t match.

For example, an example expecting `ValueError: 42` will pass if the actual exception raised is `ValueError: 3*14`, but will fail if, say, a *TypeError* is raised instead. It will also ignore any fully qualified name included before the exception class, which can vary between implementations and versions of Python and the code/libraries in use. Hence, all three of these variations will work with the flag specified:

```
>>> raise Exception('message')
Traceback (most recent call last):
Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
builtins.Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
__main__.Exception: message
```

Note that *ELLIPSIS* can also be used to ignore the details of the exception message, but such a test may still fail based on whether the module name is present or matches exactly.

버전 3.2에서 변경: *IGNORE_EXCEPTION_DETAIL*은 이제 테스트 중인 예외를 포함하는 모듈과 관련된 정보도 무시합니다.

doctest.**SKIP**

지정되면, 예제를 전혀 실행하지 않습니다. 이것은 doctest 예제가 설명서와 테스트 케이스의 두 가지 역할을 하는 문맥에서, 설명을 위해 예제를 포함해야 하지만 검사하지는 않아야 할 때 유용할 수 있습니다. 예를 들어, 예제의 출력이 임의적일 수 있습니다; 또는 예제가 테스트 구동기에서 사용할 수 없는 자원에 의존할 수 있습니다.

SKIP 플래그는 임시로 "주석 처리한" 예제를 위해 사용될 수도 있습니다.

doctest.**COMPARISON_FLAGS**

위의 모든 비교 플래그를 함께 OR 한 비트 마스크.

두 번째 옵션 그룹은 테스트 실패가 보고되는 방식을 제어합니다:

doctest.**REPORT_UDIFF**

지정되면, 여러 줄의 예상 및 실제 출력을 수반하는 실패가 통합(unified) diff를 사용하여 표시됩니다.

doctest.**REPORT_CDIFF**

지정되면, 여러 줄의 예상 및 실제 출력을 수반하는 실패가 문맥(context) diff를 사용하여 표시됩니다.

doctest.**REPORT_NDIFF**

지정되면, 차이점은 널리 사용되는 `ndiff.py` 유틸리티와 같은 알고리즘을 사용하여, `difflib.Differ`로 계산됩니다. 이 방법은 줄 간의 차이뿐만 아니라 줄 안에서의 차이점을 표시하는 유일한 방법입니다. 예를 들어, 예상 출력 줄에 숫자 `1`이 포함된 줄에 실제 출력이 문자 `l`을 포함하고 있으면, 일치하지 않는 열 위치를 나타내는 캐럿(caret)이 들어간 줄이 삽입됩니다.

doctest.**REPORT_ONLY_FIRST_FAILURE**

지정되면, 각 doctest에서 실패한 첫 번째 예제를 표시하지만, 나머지 모든 예제에 대해서는 출력을 억제합니다. 이렇게 하면 doctest가 이전의 실패로 인해 망가진 올바른 예제를 보고하지 않게 되지만, 첫

번째 실패와 무관하게 실패한 잘못된 예제를 숨길 수도 있습니다. *REPORT_ONLY_FIRST_FAILURE*가 지정될 때, 나머지 예제는 여전히 실행되며, 보고된 총실패 수에 포함됩니다; 출력만 억제됩니다.

doctest.**FAIL_FAST**

지정되면, 첫 번째 실패 예제 후에 종료하고, 나머지 예제를 실행하지 않습니다. 따라서, 보고되는 실패 횟수는 최대 1입니다. 이 플래그는 디버깅 중에 유용할 수 있습니다, 첫 번째 실패 이후의 예제는 디버깅 출력조차 생성하지 않기 때문입니다.

doctest.**REPORTING_FLAGS**

위의 모든 보고(reporting) 플래그를 함께 OR 한 비트 마스크.

새로운 옵션 플래그 이름을 등록하는 방법도 있습니다만, 서브 클래스를 통해 *doctest* 내부를 확장하려고 하지 않는 한 유용하지는 않습니다:

doctest.**register_optionflag**(*name*)

지정된 이름으로 새로운 옵션 플래그를 만들고, 새로운 플래그의 정숫값을 반환합니다. *register_optionflag()*은 *OutputChecker*나 *DocTestRunner*를 서브 클래싱할 때 서브 클래스가 지원하는 새 옵션을 만들 때 사용할 수 있습니다. *register_optionflag()*는 항상 다음의 관용구를 사용하여 호출해야 합니다:

```
MY_FLAG = register_optionflag('MY_FLAG')
```

지시자

Doctest 지시자를 사용하면 개별 예제의 옵션 플래그를 수정할 수 있습니다. Doctest 지시자는 예제의 소스 코드 뒤에 오는 특수한 파이썬 주석입니다:

```
directive:             "#" "doctest:" directive_options
directive_options:     directive_option (",," directive_option)*
directive_option:      on_or_off directive_option_name
on_or_off:             "+" | "-"
directive_option_name: "DONT_ACCEPT_BLANKLINE" | "NORMALIZE_WHITESPACE" | ...
```

+나 -와 지시자 옵션 이름 사이의 공백은 허용되지 않습니다. 지시자 옵션 이름은 위에 설명된 옵션 플래그 이름 중 하나일 수 있습니다.

예제의 doctest 지시자는 그 단일 예제에 대한 doctest의 동작을 수정합니다. 이름 붙인 동작을 활성화하려면 +를 사용하고, 비활성화하려면 -를 사용하십시오.

예를 들어, 이 테스트는 통과합니다:

```
>>> print(list(range(20)))  # doctest: +NORMALIZE_WHITESPACE
[0,   1,  2,  3,  4,  5,  6,  7,  8,  9,
10,  11, 12, 13, 14, 15, 16, 17, 18, 19]
```

지시자가 없으면 실패하는데, 실제 출력에는 한 자리 숫자 리스트 요소 앞에 두 개의 공백이 없기도 하고, 실제 출력은 한 줄이기 때문입니다. 이 테스트도 통과하는데, 그러기 위해서 역시 지시자가 필요합니다:

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS
[0, 1, ..., 18, 19]
```

하나의 물리적 줄에 여러 개의 지시자를 쉼표로 구분하여 사용할 수 있습니다:

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

하나의 예제에 여러 개의 지시자 주석이 사용되면, 모두 결합합니다:

```
>>> print(list(range(20)))  # doctest: +ELLIPSIS
...                         # doctest: +NORMALIZE_WHITESPACE
[0,    1, ...,   18,    19]
```

앞의 예가 보여주듯이, 여러분의 예제에 지시자만 포함된 ... 줄을 추가할 수 있습니다. 예가 너무 길어서 지시자가 같은 줄에 편안하게 들어갈 수 없을 때 유용할 수 있습니다:

```
>>> print(list(range(5)) + list(range(10, 20)) + list(range(30, 40)))
... # doctest: +ELLIPSIS
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

모든 옵션은 기본적으로 비활성화되고, 지시자가 표시된 예제에만 적용되므로, (지시자에 +를 통해) 옵션을 활성화하는 것이 일반적으로 유일한 의미 있는 선택입니다. 하지만, doctest를 실행하는 함수에 옵션 플래그를 전달하여 다른 기본값을 설정할 수도 있습니다. 이럴 때, 지시자에서 -를 통해 옵션을 비활성화하는 것이 유용할 수 있습니다.

경고

*doctest*는 예상 출력에서 정확한 일치를 요구하는 것에 심각합니다. 단일 문자가 일치하지 않아도 테스트가 실패합니다. 여러분이 출력에 있어서 파이썬이 정확히 무엇을 보장하고 무엇을 보장하지 않는지 배워감에 따라, 이것은 아마도 여러분을 몇 번 놀라게 할 것입니다. 예를 들어, 집합을 인쇄할 때, 파이썬은 원소가 특정 순서로 인쇄되는 것을 보장하지 않으므로, 다음과 같은 테스트는

```
>>> foo()
{"spam", "eggs"}
```

취약합니다! 한 가지 해결 방법은 다음과 같습니다

```
>>> foo() == {"spam", "eggs"}
True
```

대신에. 또 다른 방법은

```
>>> d = sorted(foo())
>>> d
['eggs', 'spam']
```

다른 것들도 있지만, 아마 아이디어를 얻었을 겁니다.

또 다른 나쁜 생각은 객체 주소를 포함하는 것들을 출력하는 것입니다

```
>>> id(1.0)  # 때로 실패할 것이 확실합니다
7948648
>>> class C: pass
>>> C()  # 인스턴스에 대한 기본 repr()은 주소를 포함합니다
<C object at 0x00AC18F0>
```

ELLIPSIS 지시자는 마지막 예제를 다루는 좋은 접근법을 제공합니다:

```
>>> C()  # doctest: +ELLIPSIS
<C object at 0x...>
```

Floating-point numbers are also subject to small output variations across platforms, because Python defers to the platform C library for some floating-point calculations, and C libraries vary widely in quality here.

```
>>> 1000**0.1  # risky
1.9952623149688797
>>> round(1000**0.1, 9) # safer
1.995262315
>>> print(f'{1000**0.1:.4f}') # much safer
1.9953
```

`I/2.**J` 형식의 숫자는 모든 플랫폼에서 안전하며, 저는 종종 이런 형식의 숫자를 만들도록 doctest 예제를 꾸밉니다:

```
>>> 3./4  # 완전히 안전합니다
0.75
```

간단한 분수는 또한 사람들이 이해하기가 더 쉬우므로, 더 좋은 설명서가 되도록 합니다.

26.4.5 기본 API

`testmod()`와 `testfile()` 함수는 대부분 기본 사용에 충분한 doctest에 대한 간단한 인터페이스를 제공합니다. 이 두 함수에 대한 덜 형식적인 소개는 섹션 간단한 사용법: 독스트링에 있는 예제 확인하기와 간단한 사용법: 텍스트 파일에 있는 예제 확인하기를 참조하십시오.

doctest.**testfile**(*filename*, *module_relative=True*, *name=None*, *package=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *parser=DocTestParser()*, *encoding=None*)

*filename*를 제외한 모든 인자는 선택적이며 키워드 형식으로 지정해야 합니다.

filename 파일에 있는 예제를 테스트합니다. `(failure_count, test_count)`를 반환합니다.

선택적 인자 *module_relative*는 filename을 해석하는 방법을 지정합니다:

- *module_relative*가 `True`(기본값)이면, *filename*는 OS 독립적 모듈 상대 경로를 지정합니다. 기본적으로, 이 경로는 호출하는 모듈의 디렉터리에 상대적입니다; 그러나 *package* 인자가 지정되면, 해당 패키지에 상대적입니다. OS 독립성을 보장하기 위해, *filename*은 `/` 문자를 사용하여 경로 세그먼트를 분리해야 하며, 절대 경로일 수 없습니다 (즉, `/`로 시작할 수 없습니다).
- *module_relative*가 `False`이면, *filename*은 OS 특정 경로를 지정합니다. 경로는 절대나 상대일 수 있습니다; 상대 경로는 현재 작업 디렉터리를 기준으로 해석됩니다.

선택적 인자 *name*은 테스트의 이름을 제공합니다; 기본적으로, 또는 `None`이면, `os.path.basename(filename)`이 사용됩니다.

선택적 인자 *package*는 디렉터리가 모듈 상대 filename의 기본 디렉터리로 사용될 파이썬 패키지나 파이썬 패키지의 이름입니다. 패키지를 지정하지 않으면, 호출하는 모듈의 디렉터리가 모듈 상대 filename의 기본 디렉터리로 사용됩니다. *module_relative*가 `False`일 때 *package*를 지정하는 것은 에러입니다.

선택적 인자 *globs*는 예제를 실행할 때 전역으로 사용될 딕셔너리를 제공합니다. doctest를 위해 이 딕셔너리의 새 얕은 사본이 만들어지므로, 예제는 깨끗한 서판으로 시작합니다. 기본적으로, 또는 `None`이면, 새 빈 딕셔너리가 사용됩니다.

선택적 인자 *extraglobs*는 예제를 실행하는 데 사용되는 전역에 병합될 딕셔너리를 제공합니다. 이것은 `dict.update()`처럼 작동합니다: *globs*와 *extraglobs*에 공통 키가 있으면, *extraglobs*의 연관된 값이 병합된 딕셔너리에 나타납니다. 기본적으로, 또는 `None`이면, 추가 전역은 사용되지 않습니다. doctest의 매개 변수화를 허용하는 고급 기능입니다. 예를 들어, doctest는 클래스의 일반 이름을 사용하여 베이스 클래스용으로 작성할 수 있습니다, 그런 다음 일반 이름을 테스트할 서브 클래스에 매핑하는 *extraglobs* 딕셔너리를 전달하여 임의의 수의 서브 클래스를 테스트하는데 재사용할 수 있습니다.

Optional argument *verbose* prints lots of stuff if true, and prints only failures if false; by default, or if `None`, it' s true if and only if `'-v'` is in `sys.argv`.

선택적 인자 *report*가 참이면 끝에 요약을 인쇄하고, 그렇지 않으면 끝에 아무것도 인쇄하지 않습니다. verbose 모드에서는 요약 정보가 상세히 표시되며, 그렇지 않으면 요약 정보는 매우 간단합니다 (사실, 모든 테스트가 통과되면 비어 있습니다).

Optional argument *optionflags* (default value `0`) takes the bitwise OR of option flags. See section 옵션 플래그.

선택적 인자 *raise_on_error*의 기본값은 거짓입니다. 참이면, 예제에서 첫 번째 실패나 예기치 않은 예외가 발생할 때 예외가 발생합니다. 이것은 실패를 사후(post-mortem) 디버깅할 수 있도록 합니다. 기본 동작은 예제를 계속 실행하는 것입니다.

선택적 인자 *parser*는 파일에서 테스트를 추출하는 데 사용할 `DocTestParser`(또는 서브 클래스)를 지정합니다. 기본값은 일반 파서(즉, `DocTestParser()`)입니다.

선택적 인자 *encoding*은 파일을 유니코드로 변환하는 데 사용할 인코딩을 지정합니다.

doctest.**testmod**(*m=None, name=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, exclude_empty=False*)

모든 인자는 선택적이며, *m*을 제외한 모든 인자는 키워드 형식으로 지정해야 합니다.

모듈 *m*(또는 *m*가 제공되지 않았거나 `None`이면 모듈 `__main__`)에서 도달할 수 있는 함수와 클래스의 독스트링에 있는 예제를 테스트합니다. `m.__doc__`으로 시작합니다.

딕셔너리 `m.__test__`이 존재하면, 여기에서 도달할 수 있는 예제도 테스트합니다. `m.__test__`는 이름(문자열)을 함수, 클래스 및 문자열에 매핑합니다; 함수와 클래스 독스트링에서 예제를 검색합니다; 문자열은 그것이 독스트링인 것처럼 직접 검색합니다.

모듈 *m*에 속하는 객체에 연결된 독스트링 만 검색합니다.

`(failure_count, test_count)`를 반환합니다.

선택적 인자 *name*은 모듈의 이름을 제공합니다; 기본적으로, 또는 `None`이면, `m.__name__`이 사용됩니다.

선택적 인자 *exclude_empty*의 기본값은 거짓입니다. 참이면, doctest가 발견되지 않은 객체는 고려 대상에서 제외됩니다. 기본값은 이전 버전과의 호환성을 위한 해킹입니다, 여전히 `testmod()`와 함께 `doctest.master.summarize`를 사용하는 코드는 테스트가 없는 객체에 대해 계속 출력합니다. 새로운 `DocTestFinder` 생성자에 대한 *exclude_empty* 인자의 기본값은 참입니다.

선택적 인자 *extraglobs*, *verbose*, *report*, *optionflags*, *raise_on_error* 및 *globs*는 위의 함수 `testfile()`와 같습니다만, *globs*의 기본값이 `m.__dict__`인 점이 다릅니다.

doctest.**run_docstring_examples**(*f, globs, verbose=False, name='NoName', compileflags=None, optionflags=0*)

객체 *f*와 관련된 예제를 테스트합니다. 여기서, *f*는 문자열, 모듈, 함수 또는 클래스 객체일 수 있습니다.

딕셔너리 인자 *globs*의 얕은 사본이 실행 컨텍스트에 사용됩니다.

선택적 인자 *name*은 실패 메시지에서 사용되며, 기본값은 `"NoName"`입니다.

선택적 인자 *verbose*가 참이면, 실패가 없어도 출력이 생성됩니다. 기본적으로, 출력은 예제가 실패할 때만 생성됩니다.

선택적 인자 *compileflags*는 예제를 실행할 때 파이썬 컴파일러에서 사용해야 하는 플래그 집합을 제공합니다. 기본적으로, 또는 `None`이면, *globs*에서 발견되는 퓨처 기능 집합에 해당하는 플래그가 추론됩니다.

선택적 인자 *optionflags*는 위의 함수 `testfile()`에서 처럼 작동합니다.

26.4.6 Unittest API

doctest된 모듈 모음이 늘어남에 따라, 모든 doctest를 체계적으로 실행하는 방법이 필요합니다. `doctest`는 doctest가 포함된 모듈과 텍스트 파일로부터 `unittest` 테스트 스위트를 만드는 데 사용할 수 있는 두 가지 함수를 제공합니다. `unittest` 테스트 탐색과 통합하려면, 테스트 모듈에 *load_tests* 함수를 포함하십시오:

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

Doctest가 있는 텍스트 파일과 모듈로부터 `unittest.TestSuite` 인스턴스를 만드는 두 가지 주요 함수가 있습니다:

doctest.**DocFileSuite**(**paths, module_relative=True, package=None, setUp=None, tearDown=None, globs=None, optionflags=0, parser=DocTestParser(), encoding=None*)

하나 이상의 텍스트 파일로부터 doctest 테스트를 `unittest.TestSuite`로 변환합니다.

The returned `unittest.TestSuite` is to be run by the unittest framework and runs the interactive examples in each file. If an example in any file fails, then the synthesized unit test fails, and a `failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number. If all the examples in a file are skipped, then the synthesized unit test is also marked as skipped.

검사할 텍스트 파일을 하나 이상의 paths(문자열)로 전달합니다.

옵션은 키워드 인자로 제공될 수 있습니다:

선택적 인자 *module_relative*는 *paths*에 있는 파일명을 해석하는 방법을 지정합니다:

- *module_relative*가 `True`(기본값)이면, *paths*의 각 파일명은 OS 독립적 모듈 상대 경로를 지정합니다. 기본적으로, 이 경로는 호출하는 모듈의 디렉터리에 상대적입니다; 그러나 *package* 인자가 지정되면, 해당 패키지에 상대적입니다. OS 독립성을 보장하기 위해, 각 파일명은 `/` 문자를 사용하여 경로 세그먼트를 분리해야 하며, 절대 경로일 수 없습니다 (즉, `/`로 시작할 수 없습니다).
- *module_relative*가 `False`이면, *paths*의 각 파일명은 OS 특정 경로를 지정합니다. 경로는 절대나 상대일 수 있습니다; 상대 경로는 현재 작업 디렉터리를 기준으로 해석됩니다.

선택적 인자 *package*는 디렉터리가 *paths*의 모듈 상대 파일명의 기본 디렉터리로 사용될 파이썬 패키지나 파이썬 패키지의 이름입니다. 패키지를 지정하지 않으면, 호출하는 모듈의 디렉터리가 모듈 상대 파일명의 기본 디렉터리로 사용됩니다. *module_relative*가 `False`일 때 *package*를 지정하는 것은 에러입니다.

Optional argument *setUp* specifies a set-up function for the test suite. This is called before running the tests in each file. The *setUp* function will be passed a `DocTest` object. The *setUp* function can access the test globals as the `globs` attribute of the test passed.

Optional argument *tearDown* specifies a tear-down function for the test suite. This is called after running the tests in each file. The *tearDown* function will be passed a `DocTest` object. The *tearDown* function can access the test globals as the `globs` attribute of the test passed.

선택적 인자 *globs*는 테스트의 초기 전역 변수를 포함하는 딕셔너리입니다. 이 딕셔너리의 새 사본이 테스트마다 만들어집니다. 기본적으로, *globs*는 새로운 빈 딕셔너리입니다.

선택적 인자 *optionflags*는 테스트에 대한 기본 doctest 옵션을 지정하는데, 개별 옵션 플래그를 함께 OR 해서 만들어집니다. 옵션 플래그 절을 참조하십시오. 보고(reporting) 옵션을 설정하는 더 좋은 방법은 아래 함수 `set_unittest_reportflags()`를 참조하십시오.

선택적 인자 *parser*는 파일에서 테스트를 추출하는 데 사용할 `DocTestParser`(또는 서브 클래스)를 지정합니다. 기본값은 일반 파서(즉, `DocTestParser()`)입니다.

선택적 인자 *encoding*은 파일을 유니코드로 변환하는 데 사용할 인코딩을 지정합니다.

전역 `__file__`이 `DocFileSuite()`를 사용하여 텍스트 파일에서 로드된 doctest에 제공된 전역에 추가됩니다.

`doctest.`**`DocTestSuite`**(*module=None, globs=None, extraglobs=None, test_finder=None, setUp=None, tearDown=None, optionflags=0, checker=None*)

모듈에 대한 doctest 테스트를 `unittest.TestSuite`로 변환합니다.

The returned `unittest.TestSuite` is to be run by the unittest framework and runs each doctest in the module. Each docstring is run as a separate unit test. If any of the doctests fail, then the synthesized unit test fails, and a `unittest.TestCase.failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number. If all the examples in a docstring are skipped, then the

선택적 인자 *module*은 테스트할 모듈을 제공합니다. 모듈 객체나 (점으로 구분될 수 있는) 모듈 이름일 수 있습니다. 지정하지 않으면, 이 함수를 호출하는 모듈이 사용됩니다.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is the module' s `__dict__`.

선택적 인자 *extraglobs*는 *globs*에 병합되는 전역 변수의 추가 집합을 지정합니다. 기본적으로, 추가 전역 변수는 사용되지 않습니다.

선택적 인자 *test_finder*는 모듈에서 doctest를 추출하는 데 사용되는 `DocTestFinder` 객체(또는 드롭인 대체)입니다.

Optional arguments *setUp*, *tearDown*, and *optionflags* are the same as for function `DocFileSuite()` above, but they are called for each docstring.

이 함수는 `testmod()`와 같은 검색 기법을 사용합니다.

버전 3.5에서 변경: *module*에 독스트링이 없으면 `DocTestSuite()`는 `ValueError`를 발생시키는 대신 빈 `unittest.TestSuite`를 반환합니다.

수면 아래에서, `DocTestSuite()`는 `doctest.DocTestCase` 인스턴스에서 `unittest.TestSuite`를 만들고, `DocTestCase`는 `unittest.TestCase`의 서브 클래스입니다. `DocTestCase`는 여기에서 설명되지는 않지만 (내부 세부 사항입니다), 그것의 코드를 살펴보면 `unittest` 통합의 정확한 세부 사항에 대한 질문에 대한 답을 얻을 수 있습니다.

마찬가지로, `DocFileSuite()`는 `doctest.DocFileCase` 인스턴스에서 `unittest.TestSuite`를 만들고, `DocFileCase`는 `DocTestCase`의 서브 클래스입니다.

So both ways of creating a `unittest.TestSuite` run instances of `DocTestCase`. This is important for a subtle reason: when you run `doctest` functions yourself, you can control the `doctest` options in use directly, by passing option flags to `doctest` functions. However, if you' re writing a `unittest` framework, `unittest` ultimately controls when and how tests get run. The framework author typically wants to control `doctest` reporting options (perhaps, e.g., specified by command line options), but there' s no way to pass options through `unittest` to `doctest` test runners.

For this reason, `doctest` also supports a notion of `doctest` reporting flags specific to `unittest` support, via this function:

`doctest.`**`set_unittest_reportflags`**(*flags*)

사용할 `doctest` 보고 플래그를 설정합니다.

인자 *flags*는 옵션 플래그의 비트별 OR를 취합니다. 옵션 플래그 절을 참조하십시오. "보고 플래그"만 사용할 수 있습니다.

This is a module-global setting, and affects all future doctests run by module `unittest`: the `runTest()` method of `DocTestCase` looks at the option flags specified for the test case when the `DocTestCase` instance was constructed. If no reporting flags were specified (which is the typical and expected case), `doctest`' s `unittest` reporting flags are bitwise ORed into the option flags, and the option flags so augmented are passed to the `DocTestRunner` instance created to run the doctest. If any reporting flags were specified when the `DocTestCase` instance was constructed, `doctest`' s `unittest` reporting flags are ignored.

함수가 호출되기 전에 유효했던 `unittest` 보고 플래그의 값이 함수에 의해 반환됩니다.

26.4.7 고급 API

기본 API는 doctest를 사용하기 쉽게 하기 위한 간단한 래퍼입니다. 그것은 매우 유연하며, 대부분 사용자의 요구를 충족시켜야 합니다; 그러나, 테스팅에 대한 세밀한 제어가 필요하거나, doctest의 기능을 확장하려면, 고급 API를 사용해야 합니다.

고급 API는 doctest 케이스에서 추출한 대화식 예제를 저장하는 데 사용되는 두 개의 컨테이너 클래스를 중심으로 돌아갑니다:

- `Example`: 예상 출력과 쌍을 이루는 단일 파이썬 문장.
- `DocTest`: 일반적으로 단일 독스트링이나 텍스트 파일에서 추출된 `Example`의 모음.

doctest 예제를 찾고, 구문 분석하고, 실행하고, 검사하기 위해 추가 처리 클래스가 정의됩니다:

- `DocTestFinder`: 주어진 모듈에서 모든 독스트링을 찾고, `DocTestParser`를 사용하여 대화식 예제가 들어있는 모든 독스트링에서 `DocTest`를 만듭니다.
- `DocTestParser`: 문자열(가령 객체의 독스트링)에서 `DocTest` 객체를 만듭니다.
- `DocTestRunner`: `DocTest`에 있는 예제를 실행하고, `OutputChecker`를 사용하여 출력을 확인합니다.
- `OutputChecker`: doctest 예제의 실제 출력을 예상 출력과 비교하고, 그들이 일치하는지 결정합니다.

이러한 처리 클래스 간의 관계는 다음 도표에 요약되어 있습니다:

```
list of:
+------+                    +---------+
|module| --DocTestFinder-> | DocTest | --DocTestRunner-> results
+------+    |         ^     +---------+     |       ^     (printed)
            |         |     | Example |     |       |
            v         |     |   ...   |     v       |
          DocTestParser     | Example |   OutputChecker
                            +---------+
```

DocTest 객체

class doctest.**DocTest**(*examples, globs, name, filename, lineno, docstring*)

단일 이름 공간에서 실행되어야 하는 doctest 예제의 모음. 생성자 인자는 같은 이름의 어트리뷰트를 초기화하는 데 사용됩니다.

*DocTest*는 다음 어트리뷰트를 정의합니다. 이들은 생성자에 의해 초기화되며, 직접 수정하면 안 됩니다.

examples

이 테스트가 실행해야 하는 개별 대화형 파이썬 예제를 인코딩하는 *Example* 객체의 리스트.

globs

예제가 실행되어야 하는 이름 공간 (일명 전역). 이름을 값에 매핑하는 딕셔너리입니다. 예제가 만든 이름 공간의 모든 변경 사항(가령 새 변수 바인딩)은 테스트가 실행된 후 *globs*에 반영됩니다.

name

*DocTest*를 식별하는 문자열 이름. 일반적으로, 테스트가 추출된 객체나 파일의 이름입니다.

filename

The name of the file that this *DocTest* was extracted from; or `None` if the filename is unknown, or if the `DocTest` was not extracted from a file.

lineno

이 *DocTest*가 시작되는 *filename* 내의 줄 번호, 또는 줄 번호가 없으면 `None`. 이 줄 번호는 파일의 시작 부분을 기준으로 0에서 시작합니다.

docstring

테스트가 추출된 문자열, 또는 문자열이 없거나 테스트가 문자열에서 추출되지 않았으면 `None`.

Example 객체

class doctest.**Example**(*source, want, exc_msg=None, lineno=0, indent=0, options=None*)

파이썬 문장과 예상 출력으로 구성된 단일 대화형 예제. 생성자 인자는 같은 이름의 어트리뷰트를 초기화하는 데 사용됩니다.

*Example*는 다음 어트리뷰트를 정의합니다. 이들은 생성자에 의해 초기화되며, 직접 수정하면 안 됩니다.

source

예제의 소스 코드가 포함된 문자열. 이 소스 코드는 단일 파이썬 문으로 구성되며 항상 개행으로 끝납니다; 생성자는 필요하면 개행 문자를 추가합니다.

want

예제의 소스 코드를 실행할 때 (stdout이나 예외 발생 시 트레이스백으로부터) 예상되는 출력. *want*는 출력이 예상되지 않으면 빈 문자열이고, 그렇지 않으면 개행으로 끝납니다. 생성자는 필요하면 개행을 추가합니다.

exc_msg

예제가 예외를 생성할 것으로 예상되면, 예제에서 생성된 예외 메시지; 또는 예외를 생성할 것으로 예상되지 않으면 None. 이 예외 메시지는 `traceback.format_exception_only()`의 반환값과 비교됩니다. `exc_msg`는 None이 아니면 개행으로 끝납니다. 생성자는 필요하면 개행을 추가합니다.

lineno

이 예제가 시작하는 이 예제를 포함하는 문자열 내의 줄 번호. 이 줄 번호는 포함하는 문자열의 시작 부분을 기준으로 0에서 시작합니다.

indent

포함하는 문자열 내에서의 이 예제의 들여쓰기, 즉 예제의 첫 번째 프롬프트 앞에 오는 스페이스 문자의 수.

options

옵션 플래그에서 True나 False로 매핑하는 딕셔너리, 이 예제의 기본 옵션을 재정의하는 데 사용됩니다. 이 딕셔너리에 포함되지 않은 옵션 플래그는 (`DocTestRunner`의 *optionflags*에 지정된 대로) 기본값으로 남습니다. 기본적으로, 아무런 옵션도 설정되지 않습니다.

DocTestFinder 객체

class doctest.**DocTestFinder**(*verbose=False*, *parser=DocTestParser()*, *recurse=True*, *exclude_empty=True*)

주어진 객체에 관련된 `DocTest`를 그것의 독스트링과 그것이 포함하는 객체의 독스트링으로부터 추출하기 위해서 사용되는 처리 클래스. `DocTest`는 모듈, 클래스, 함수, 메서드, 정적 메서드, 클래스 메서드 및 프로퍼티에서 추출할 수 있습니다.

선택적 인자 *verbose*는 파인더가 검색한 객체를 표시하는 데 사용될 수 있습니다. 기본값은 False (출력 없음)입니다.

선택적 인자 *parser*는 독스트링에서 doctest를 추출하는 데 사용되는 `DocTestParser` 객체 (또는 드롭인 대체)를 지정합니다.

선택적 인자 *recurse*가 거짓이면, `DocTestFinder.find()`는 오직 주어진 객체만을 검사 할 뿐, 포함된 객체는 검사하지 않습니다.

선택적 인자 *exclude_empty*가 거짓이면, `DocTestFinder.find()`는 빈 독스트링을 가진 객체에 대한 테스트를 포함합니다.

`DocTestFinder`는 다음 메서드를 정의합니다:

find(*obj*[, *name*][, *module*][, *globs*][, *extraglobs*])

*obj*의 독스트링이나 포함된 객체의 독스트링으로 정의된 `DocTest`의 리스트를 반환합니다.

선택적 인자 *name*은 객체의 이름을 지정합니다. 이 이름은 반환된 `DocTest`의 이름을 구성하는 데 사용됩니다. *name*이 지정되지 않으면, obj.__name__이 사용됩니다.

선택적 매개 변수 *module*은 주어진 객체를 포함하는 모듈입니다. 모듈이 지정되지 않거나 None이면, 테스트 파인더는 자동으로 올바른 모듈을 판별하려고 시도합니다. 객체의 모듈은 다음과 같이 사용됩니다:

- *globs*가 지정되지 않으면, 기본 이름 공간으로.
- DocTestFinder가 다른 모듈에서 임포트 된 객체에서 DocTest를 추출하지 못하도록 하려고. (*module*이 아닌 다른 모듈을 가진 포함 된 객체는 무시됩니다.)
- 객체를 포함하는 파일의 이름을 찾으려고.
- 해당 파일 내에서 객체의 줄 번호를 찾는 데 도움이 됩니다.

*module*이 False면, 모듈을 찾으려고 시도하지 않습니다. 이것은 눈에 띄지 않는데, 대부분 doctest 자체를 테스트할 때 사용합니다: *module*이 False이거나, None이지만 자동으로 찾을 수 없으면, 모든 객체는 (존재하지 않는) 모듈에 속한 것으로 간주하므로, 포함된 모든 객체에서 (재귀적으로) doctest를 검색합니다.

The globals for each `DocTest` is formed by combining *globs* and *extraglobs* (bindings in *extraglobs* override bindings in *globs*). A new shallow copy of the globals dictionary is created for each DocTest.

If *globs* is not specified, then it defaults to the module' s `__dict__`, if specified, or `{}` otherwise. If *extraglobs* is not specified, then it defaults to `{}`.

DocTestParser 객체

class doctest.**DocTestParser**

문자열에서 대화형 예제를 추출하고, 이를 사용하여 `DocTest` 객체를 만드는 데 사용되는 처리 클래스.

`DocTestParser`는 다음 메서드를 정의합니다:

get_doctest(*string*, *globs*, *name*, *filename*, *lineno*)

주어진 문자열에서 모든 doctest 예제를 추출하고, 이를 `DocTest` 객체로 모읍니다.

globs, *name*, *filename*, and *lineno* are attributes for the new `DocTest` object. See the documentation for `DocTest` for more information.

get_examples(*string*, *name*='<*string*>')

주어진 문자열에서 모든 doctest 예제를 추출하고, 이를 `Example` 객체의 리스트로 반환합니다. 줄 번호는 0부터 시작합니다. 선택적 인자 *name*은, 이 문자열을 식별하는 이름이며, 에러 메시지에만 사용됩니다.

parse(*string*, *name*='<*string*>')

Divide the given string into examples and intervening text, and return them as a list of alternating `Example`s and strings. Line numbers for the `Examples` are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

TestResults 객체

class doctest.**TestResults**(*failed*, *attempted*)

failed

Number of failed tests.

attempted

Number of attempted tests.

skipped

Number of skipped tests.

Added in version 3.13.

DocTestRunner 객체

class doctest.**DocTestRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

`DocTest`에 있는 대화형 예제를 실행하고 검증하는 데 사용되는 처리 클래스.

The comparison between expected outputs and actual outputs is done by an `OutputChecker`. This comparison may be customized with a number of option flags; see section 옵션 플래그 for more information. If the option flags are insufficient, then the comparison may also be customized by passing a subclass of `OutputChecker` to the constructor.

테스트 실행기의 디스플레이 출력은 두 가지 방법으로 제어할 수 있습니다. 첫째, 출력 함수를 `run()`로 전달할 수 있습니다; 이 함수는 표시되어야 하는 문자열로 호출됩니다. 기본값은 `sys.stdout.write`입니다. 출력을 캡처하는 것으로 충분하지 않으면, DocTestRunner를 서브 클래싱하고 `report_start()`, `report_success()`, `report_unexpected_exception()` 및 `report_failure()` 메서드를 재정의하여 디스플레이 출력을 사용자 정의할 수 있습니다.

선택적 키워드 인자 *checker*는 예상 출력을 doctest 예제의 실제 출력과 비교하는 데 사용되는 `OutputChecker` 객체(또는 드롭 인 대체)를 지정합니다.

선택적 키워드 인자 *verbose*는 `DocTestRunner`의 상세도를 제어합니다. *verbose*가 `True`이면, 실행될 때 각 예제에 대한 정보가 인쇄됩니다. *verbose*가 `False`이면, 실패만 인쇄됩니다. *verbose*가 지정되지 않거나 `None`이면, 명령 줄 스위치 `-v`가 사용될 때만 상세 출력이 사용됩니다.

선택적 키워드 인자 *optionflags*는 테스트 실행기가 예상 출력을 실제 출력과 비교하는 방법과 실패를 표시하는 방법을 제어하는 데 사용할 수 있습니다. 자세한 내용은 옵션 플래그 절을 참조하십시오.

The test runner accumulates statistics. The aggregated number of attempted, failed and skipped examples is also available via the `tries`, `failures` and `skips` attributes. The `run()` and `summarize()` methods return a `TestResults` instance.

`DocTestRunner`는 다음 메서드를 정의합니다:

report_start(*out*, *test*, *example*)

테스트 러너가 주어진 예제를 처리하려고 한다고 보고합니다. 이 메서드는 `DocTestRunner`의 서브 클래스가 출력을 사용자 정의할 수 있도록 제공됩니다; 직접 호출해서는 안 됩니다.

example is the example about to be processed. *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run()`.

report_success(*out*, *test*, *example*, *got*)

주어진 예제가 성공적으로 실행되었음을 보고합니다. 이 메서드는 `DocTestRunner`의 서브 클래스가 출력을 사용자 정의할 수 있도록 제공됩니다; 직접 호출해서는 안 됩니다.

*example*은 처리될 예제입니다. *got*은 예제의 실제 출력입니다. *test*는 *example*을 포함하는 테스트입니다. *out*은 `DocTestRunner.run()`에 전달된 출력 함수입니다.

report_failure(*out*, *test*, *example*, *got*)

주어진 예제가 실패했음을 보고합니다. 이 메서드는 `DocTestRunner`의 서브 클래스가 출력을 사용자 정의할 수 있도록 제공됩니다; 직접 호출해서는 안 됩니다.

*example*은 처리될 예제입니다. *got*은 예제의 실제 출력입니다. *test*는 *example*을 포함하는 테스트입니다. *out*은 `DocTestRunner.run()`에 전달된 출력 함수입니다.

report_unexpected_exception(*out*, *test*, *example*, *exc_info*)

주어진 예제가 예기치 않은 예외를 발생시켰다고 보고합니다. 이 메서드는 `DocTestRunner`의 서브 클래스가 출력을 사용자 정의할 수 있도록 제공됩니다; 직접 호출해서는 안 됩니다.

*example*은 처리될 예제입니다. *exc_info*는 예기치 않은 예외에 대한 정보를 포함하는 튜플입니다 (`sys.exc_info()`에 의해 반환되는 것). *test*는 *example*을 포함하는 테스트입니다. *out*은 `DocTestRunner.run()`에 전달된 출력 함수입니다.

run(*test*, *compileflags=None*, *out=None*, *clear_globs=True*)

test(`DocTest` 객체)에 있는 예제를 실행하고, 출력 함수 *out*을 사용하여 결과를 표시합니다. `TestResults` 인스턴스를 반환합니다.

예제는 이름 공간 `test.globs`에서 실행됩니다. *clear_globs*가 참(기본값)이면, 가비지 수집을 돕기 위해 테스트가 실행된 후 이 이름 공간이 지워집니다. 테스트가 완료된 후에 이름 공간을 검사하려면 *clear_globs=False*를 사용하십시오.

*compileflags*는 예제를 실행할 때 파이썬 컴파일러에서 사용해야 하는 플래그 집합을 제공합니다. 지정되지 않으면, *globs*에 적용되는 퓨처-임포트 플래그 집합이 기본값이 됩니다.

각 예제의 출력은 `DocTestRunner`의 출력 검사기를 사용하여 검사되며, 결과는 `DocTestRunner.report_*()` 메서드로 포맷됩니다.

summarize(*verbose=None*)

이 DocTestRunner가 실행 한 모든 테스트 케이스의 요약을 인쇄하고, `TestResults` 인스턴스를 반환합니다.

선택적 *verbose* 인자는 요약이 얼마나 상세할지를 제어합니다. 상세도가 지정되지 않으면, `DocTestRunner`의 상세도가 사용됩니다.

`DocTestParser`는 다음과 같은 어트리뷰트를 갖습니다:

tries

Number of attempted examples.

failures

Number of failed examples.

skips

Number of skipped examples.

Added in version 3.13.

OutputChecker 객체

class doctest.**OutputChecker**

doctest 예제의 실제 출력이 예상 출력과 일치하는지를 확인하는 데 사용되는 클래스. *OutputChecker*는 두 가지 메서드를 정의합니다: *check_output()*은 주어진 출력 쌍을 비교하고 일치하면 `True`를 반환합니다; *output_difference()*는 두 출력 간의 차이를 설명하는 문자열을 반환합니다.

*OutputChecker*는 다음 메서드를 정의합니다:

check_output(*want*, *got*, *optionflags*)

예제의 실제 출력(*got*)이 예상 출력(*want*)과 일치할 때만 `True`를 반환합니다. 이 문자열은 같으면 항상 일치하는 것으로 간주합니다; 그러나 테스트 실행기가 사용하는 옵션 플래그에 따라 몇 가지 정확하지 않은 일치 유형도 가능합니다. 옵션 플래그에 대한 자세한 정보는 옵션 플래그 절을 참조하십시오.

output_difference(*example*, *got*, *optionflags*)

주어진 예제(*example*)에 대한 예상 출력과 실제 출력(*got*)의 차이를 설명하는 문자열을 반환합니다. *optionflags*는 *want*와 *got*을 비교하는 데 사용되는 옵션 플래그 집합입니다.

26.4.8 디버깅

Doctest는 doctest 예제를 디버깅하기 위한 몇 가지 메커니즘을 제공합니다:

- 몇몇 함수는 doctest를 파이썬 디버거 *pdb*에서 실행할 수 있는 실행 가능한 파이썬 프로그램으로 변환합니다.
- *DebugRunner* 클래스는 첫 번째 실패한 예제에 대한 예외를 발생시키는 *DocTestRunner*의 서브 클래스이며 해당 예제에 대한 정보가 들어 있습니다. 이 정보는 예제에서 사후 디버깅을 수행하는 데 사용될 수 있습니다.
- *DocTestSuite()*에 의해 생성된 *unittest* 케이스는 *unittest.TestCase*에 의해 정의된 *debug()* 메서드를 지원합니다.
- doctest 예제에 *pdb.set_trace()*에 대한 호출을 추가 할 수 있습니다. 그러면 해당하는 줄이 실행될 때 파이썬 디버거로 들어갑니다. 그런 다음 변수의 현재 값을 검사하는 등의 일을 할 수 있습니다. 예를 들어, `a.py`가 다음과 같은 모듈 독스트링을 포함한다고 가정합시다:

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print(x+3)
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

그러면 대화형 파이썬 세션은 이런 식이 됩니다:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
  2         print(x+3)
  3  ->     import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
[EOF]
(Pdb) p x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

Doctest를 파이썬 코드로 변환하고, 디버거에서 합성 코드를 실행할 수 있는 함수들:

doctest.**script_from_examples**(*s*)

예제가 있는 텍스트를 스크립트로 변환합니다.

인자 *s*는 doctest 예제를 포함하는 문자열입니다. 문자열은 파이썬 스크립트로 변환됩니다. 여기서 *s*의 doctest 예제는 일반 코드로 변환되고, 나머지는 파이썬 주석으로 변환됩니다. 생성된 스크립트는 문자열로 반환됩니다. 예를 들어,

```
import doctest
print(doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print(x+y)
    3
"""))
```

는 다음과 같이 출력합니다:

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print(x+y)
# Expected:
## 3
```

이 함수는 다른 함수(아래 참조)에서 내부적으로 사용되지만, 대화형 파이썬 세션을 파이썬 스크립트로 변환하려고 할 때도 유용합니다.

doctest.**testsource**(*module*, *name*)

객체에 대한 doctest를 스크립트로 변환합니다.

인자 *module*은 doctest가 관심 대상인 객체를 포함하는 모듈 객체나 모듈의 점으로 구분된 이름입니다. 인자 *name*은 doctest가 관심 대상인 객체의 (모듈 내에서의) 이름입니다. 결과는 위의 `script_from_examples()`에서 설명한 대로 파이썬 스크립트로 변환된 객체의 독스트링을 포함하는 문자열입니다. 예를 들어, 모듈 `a.py`에 최상위 함수 `f()`가 포함되었다면,

```
import a, doctest
print(doctest.testsource(a, "a.f"))
```

는 doctest가 코드로 변환되고 나머지는 주석으로 배치된, 함수 `f()`의 독스트링의 스크립트 버전을 인쇄합니다.

doctest.**debug**(*module*, *name*, *pm=False*)

객체의 doctest를 디버그합니다.

*module*과 *name* 인자는 위의 함수 `testsource()`와 같습니다. 명명된 객체의 독스트링에 대한 합성된 파이썬 스크립트가 임시 파일에 기록되고, 그 파일을 파이썬 디버거 `pdb`의 제어하에 실행합니다.

`module.__dict__`의 얕은 사본이 지역과 전역 실행 컨텍스트 모두에 사용됩니다.

선택적 인자 *pm*은 사후 디버깅이 사용되는지를 제어합니다. *pm*이 참값이면, 스크립트 파일은 직접 실행되고, 처리되지 않은 예외를 발생시켜 스크립트가 종료될 때만 디버거가 개입합니다. 그럴 때, 사후 디버깅이 `pdb.post_mortem()`를 통해 호출되어, 처리되지 않은 예외로부터 온 트레이스백 객체를 전달합니다. *pm*이 지정되지 않았거나 거짓이면, 스크립트는 적절한 `exec()` 호출을 `pdb.run()`에 전달하여 시작부터 디버거에서 실행됩니다.

doctest.**debug_src**(*src*, *pm=False*, *globs=None*)

문자열에 있는 doctest를 디버그합니다.

doctest 예제를 포함하는 문자열이 *src* 인자를 통해 직접 지정된다는 점을 제외하면, 위의 함수 `debug()`과 같습니다.

선택적 인자 *pm*은 위의 함수 `debug()`에서와 같은 의미를 가집니다.

선택적 인자 *globs*는 지역과 전역 실행 컨텍스트 모두에 사용할 딕셔너리를 제공합니다. 지정되지 않거나 `None`이면, 빈 딕셔너리가 사용됩니다. 지정되면, 딕셔너리의 얕은 사본이 사용됩니다.

`DebugRunner` 클래스와 이 클래스가 발생시킬 수 있는 특별한 예외는 주로 테스트 프레임워크 작성자가 관심을 가지며, 여기에서는 대략적으로만 다룰 예정입니다. 자세한 내용은 소스 코드, 특히 `DebugRunner`의 독스트링(doctest입니다!)을 참조하십시오:

class doctest.**DebugRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

실패를 만나자마자 예외를 발생시키는 `DocTestRunner`의 서브 클래스. 예기치 않은 예외가 발생하면, 테스트, 예제 및 원래 예외가 포함된 `UnexpectedException` 예외가 발생합니다. 출력이 일치하지 않으면, 테스트, 예제 및 실제 출력을 포함하는 `DocTestFailure` 예외가 발생합니다.

생성자 매개 변수와 메서드에 대한 자세한 내용은 *고급 API* 절의 `DocTestRunner` 설명서를 참조하십시오.

`DebugRunner` 인스턴스가 발생시킬 수 있는 두 가지 예외가 있습니다:

exception doctest.**DocTestFailure**(*test*, *example*, *got*)

doctest 예제의 실제 출력이 예상 출력과 일치하지 않는다는 것을 알리기 위해 `DocTestRunner`가 발생시키는 예외. 생성자 인자는 같은 이름의 어트리뷰트를 초기화하는 데 사용됩니다.

`DocTestFailure`는 다음 어트리뷰트를 정의합니다:

DocTestFailure.**test**

예제가 실패했을 때 실행 중이던 `DocTest` 객체.

DocTestFailure.**example**

실패한 `Example`.

DocTestFailure.**got**

예제의 실제 출력.

exception doctest.**UnexpectedException**(*test*, *example*, *exc_info*)

doctest 예제가 예기치 않은 예외를 발생시켰음을 알리기 위해 *DocTestRunner*가 발생시키는 예외. 생성자 인자는 같은 이름의 어트리뷰트를 초기화하는 데 사용됩니다.

*UnexpectedException*는 다음 어트리뷰트를 정의합니다:

UnexpectedException.**test**

예제가 실패했을 때 실행 중이던 *DocTest* 객체.

UnexpectedException.**example**

실패한 *Example*.

UnexpectedException.**exc_info**

*sys.exc_info()*에 의해 반환되는 것과 같은, 예기치 않은 예외에 대한 정보가 포함된 튜플.

26.4.9 맺음말

소개에서 언급했듯이, *doctest*는 다음 세 가지 주요 용도로 성장했습니다:

1. 독스트링에 있는 예제 검사.
2. 회귀 테스트.
3. 실행 가능한 문서/문학적(literate) 테스트.

이러한 용도들은 다른 요구 사항을 가지며, 이를 구별하는 것이 중요합니다. 특히, 모호한 테스트 케이스로 독스트링을 채우는 것은 나쁜 설명서를 만듭니다.

독스트링을 작성할 때, 독스트링 예제를 주의해서 선택하십시오. 여기에는 배울 필요가 있는 기술이 있습니다—처음에는 자연스럽지 않을 수도 있습니다. 예제는 설명서에 진짜 가치를 부여해야 합니다. 좋은 예제는 종종 많은 단어의 가치가 있습니다. 주의 깊게 작업 된다면, 예제는 사용자에게 매우 가치 있을 것이며, 몇 년이 지나고 변함에 따라 여러 번 수집하는 데 드는 시간을 갚을 것입니다. 제 *doctest* 예제 중 하나가 "해가 없는" 변경 후에 얼마나 자주 작동을 멈추는지 지금도 놀라울 뿐입니다.

Doctest는 회귀 테스트를 위한 훌륭한 도구도 제공합니다. 특히 설명 텍스트를 생략하지 않는다면 더욱더 그렇습니다. 설명과 예제를 번갈아 보여줌으로써, 실제로 무엇이 왜 테스트 되는지를 추적하기가 훨씬 쉬워집니다. 테스트가 실패할 때, 좋은 설명은 문제가 무엇인지, 어떻게 고쳐야 하는지를 쉽게 파악할 수 있게 해줍니다. 코드 기반 테스트에 광범위한 주석을 쓸 수는 있는 것은 사실이지만, 그렇게 하는 프로그래머는 거의 없습니다. 많은 사람이 doctest 접근법을 사용하는 것이 훨씬 더 명확한 테스트를 유도한다는 것을 발견했습니다. 어쩌면 단순히 doctest가 설명을 작성하는 것을 코드를 작성하는 것보다 조금 더 쉽게 만들고, 코드에 주석을 쓰는 것이 조금 더 어렵기 때문일 것입니다. 저는 단지 그것보다는 더 깊이 들어간다고 생각합니다: doctest 기반 테스트를 작성할 때의 자연스러운 태도는 소프트웨어의 미세한 포인트를 설명하고 그것을 예제로 보여주는 것입니다. 이것은 자연스럽게 가장 간단한 기능으로 시작하고, 복잡하고 지엽적인 경우까지 논리적으로 진행되는 테스트 파일로 이어집니다. 무작위로 보이는 격리된 기능 조각을 테스트하는 격리된 함수들의 모음 대신에, 일관된 내러티브가 얻어집니다. 이것은 다른 태도이며, 테스트와 설명의 구별을 모호하게 하면서 다른 결과를 낳습니다.

회귀 테스트는 전용 객체나 파일로 제한하는 것이 가장 좋습니다. 테스트 구성을 위한 몇 가지 옵션이 있습니다.:

- 대화형 예제로 테스트 케이스가 들어있는 텍스트 파일을 작성하고, *testfile()*이나 *DocFileSuite()*를 사용하여 파일을 테스트하십시오. doctest를 처음부터 사용하도록 고안된 새로운 프로젝트에서 가장 쉬운 방법이지만, 이 방법을 권장합니다.
- 명명된 주제에 대한 테스트 케이스를 포함하는 단일 독스트링으로 구성된 _regrtest_topic이라는 함수를 정의하십시오. 이 함수들은 모듈과 같은 파일에 포함되거나 별도의 테스트 파일로 분리될 수 있습니다.
- Define a *__test__* dictionary mapping from regression test topics to docstrings containing test cases.

테스트를 모듈에 배치할 때, 모듈 자체가 테스트 실행기가 될 수 있습니다. 테스트가 실패할 때, 문제를 디버깅하는 동안 실패한 doctest 만 다시 실행하도록 테스트 실행기를 조정할 수 있습니다. 다음은 그러한 테스트 실행기의 최소 예입니다:

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.FAIL_FAST
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print(f"{fail} failures out of {total} tests")
```

26.5 unittest — 단위 테스트 프레임워크

소스 코드: Lib/unittest/__init__.py

(당신이 이미 테스트 기본 개념에 친숙하다면, *assert* 메서드 목록으로 건너뛰어도 좋습니다.)

`unittest` 단위 테스트 프레임워크는 본래 JUnit으로부터 영감을 받고 다른 언어의 주요 단위 테스트 프레임워크와 비슷한 특징을 가지고 있습니다. 이것은 테스트 자동화, 테스트를 위한 사전 설정(setup)과 종료(shutdown) 코드 공유, 테스트를 컬렉션에 종합하기, 테스트와 리포트 프레임워크의 분리 등을 지원합니다.

이를 달성하기 위해 `unittest`는 객체 지향적인 방법으로 몇 가지 중요한 개념을 지원합니다.

테스트 픽스쳐
: 테스트 픽스쳐 (*test fixture*)는 1 개 또는 그 이상의 테스트를 수행할 때 필요한 준비와 그와 관련된 정리 동작에 해당합니다. 예를 들어 이것은 임시 또는 프락시 데이터베이스, 디렉터리를 생성하거나 서버 프로세스를 시작하는 것 등을 포함합니다.

테스트 케이스
: 테스트 케이스(*test case*)는 테스트의 개별 단위입니다. 이것은 특정한 입력 모음에 대해서 특정한 결과를 확인합니다. `unittest`는 베이스 클래스인 `TestCase`를 지원합니다. 이 클래스는 새로운 테스트 케이스를 만드는 데 사용됩니다.

테스트 묶음
: 테스트 묶음(*test suite*)은 여러 테스트 케이스, 테스트 묶음, 또는 둘 다의 모임입니다. 이것은 서로 같이 실행되어야 할 테스트들을 종합하는 데 사용됩니다.

테스트 실행자
: 테스트 실행자(*test runner*)는 테스트 실행을 조율하고 테스트 결과를 사용자에게 제공하는 역할을 하는 컴포넌트입니다. 실행자는 테스트 실행 결과를 보여주기 위해 그래픽 인터페이스, 텍스트 인터페이스를 사용하거나 특별한 값을 반환할 수도 있습니다.

더 보기

***doctest* 모듈**
: 매우 다른 특징을 가지고 있는 또 다른 테스트 지원 모듈

Simple Smalltalk Testing: With Patterns
: `unittest`에 영향을 준 Kent Beck의 패턴을 사용한 테스트 프레임워크 원본 논문

pytest
: 테스트를 작성하기에 간편한 문법을 가지고 있는 제삼자의 단위 테스트 프레임워크. 예시, `assert func(10) == 42`.

파이썬 테스트 도구 분류
: 함수형 테스트 프레임워크와 모의 객체 라이브러리를 포함한 광범위한 파이썬 테스트 도구 목록

Testing in Python 메일링 리스트
: 파이썬에서 테스트하기와 테스트 도구에 대해 논의하는 특정-주제-그룹 (special-interest-group)

파이썬 소스 배포판에 있는 `Tools/unittestgui/unittestgui.py` 스크립트는 테스트 탐색 및 실행을 위한 GUI 도구입니다. 이것은 단위 테스트가 처음인 사람들이 쉽게 사용할 수 있도록 만들어졌습니다. 라이브 환경에서는 Buildbot, Jenkins, GitHub Actions 또는 AppVeyor와 같은 지속적인 통합 시스템을 이용하여 테스트가 이루어지길 추천합니다.

26.5.1 기본 예시

unittest 모듈은 테스트를 구성하고 실행하는 데 풍부한 도구 모음을 제공하고 있습니다. 이 절에서는 대부분 사용자의 요구를 충족시키기 위해 일부 도구 모음만으로도 충분하다는 것을 보여줍니다.

문자열 관련된 3개의 메서드를 테스트하기 위한 짧은 스크립트가 여기에 있습니다:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # 구분 기호가 문자열이 아닐 때 s.split이 실패하는지 검사합니다
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

A test case is created by subclassing *unittest.TestCase*. The three individual tests are defined with methods whose names start with the letters `test`. This naming convention informs the test runner about which methods represent tests.

각 테스트의 핵심은 기대되는 결과를 확인하기 위해 *assertEqual()*를 호출, 조건을 검증하기 위해 *assertTrue()* 또는 *assertFalse()*를 호출, 특정 예외가 발생했는지 검증하기 위해 *assertRaises()*를 호출하는 것입니다. `assert` 문장을 대신하여 이 메서드들을 사용하면 테스트 실행자가 모든 테스트 결과를 취합하여 리포트를 생성할 수 있습니다.

*setUp()*과 *tearDown()* 메서드로 각각의 테스트 메서드 전과 후에 실행될 명령어를 정의할 수 있습니다. 테스트 코드 구조 잡기에서 이것을 더 자세히 다루겠습니다.

마지막 블록은 테스트를 실행하는 간단한 방법을 보여줍니다. *unittest.main()*은 테스트 스크립트에 명령행 인터페이스를 제공합니다. 명령행에서 위 스크립트를 실행하면, 다음과 같은 출력이 나옵니다:

```
...
----------------------------------------------------------------------
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Ran 3 tests in 0.000s

OK
```

-v 옵션을 테스트 스크립트에 넘겨주게 되면 *unittest.main()*은 높은 상세도(verbosity)를 설정하여 그에 따른 출력이 나옵니다:

```
test_isupper (__main__.TestStringMethods.test_isupper) ... ok
test_split (__main__.TestStringMethods.test_split) ... ok
test_upper (__main__.TestStringMethods.test_upper) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

위의 예시는 *unittest*에서 가장 자주 사용되는 기능을 보여주며 이것은 많은 일상적인 테스트 요구 사항을 충족시키기에 충분합니다. 문서의 나머지 부분은 기초부터 시작해서 모든 기능을 살펴봅니다.

버전 3.11에서 변경: The behavior of returning a value from a test method (other than the default `None` value), is now deprecated.

26.5.2 명령행 인터페이스

unittest 모듈은 명령행을 사용하여 모듈, 클래스, 심지어 각 테스트 메서드의 테스트들을 실행할 수 있습니다:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

모듈 이름이나 완전히 정규화된 클래스나 메서드 이름이 포함된 목록을 전달할 수 있습니다.

테스트 모듈은 파일 경로로도 지정할 수 있습니다:

```
python -m unittest tests/test_something.py
```

이것으로 테스트 모듈을 지정할 때 셸(shell)의 파일 이름 완성 기능을 사용할 수 있습니다. 지정된 파일은 반드시 모듈로 임포트 가능해야 합니다. 파일 경로는 '.py' 가 빠지면서 모듈 이름으로 변경되고 경로 구분자도 '.' 로 변경됩니다. 만약 당신이 임포트 가능하지 않은 테스트 파일을 모듈로 사용하고 싶으시다면 이 방법 대신에 그 파일을 직접 실행해야 합니다.

-v 옵션을 사용하여 더 자세한 정보(높은 상세도)로 테스트를 실행할 수 있습니다:

```
python -m unittest -v test_module
```

아무 인자 없이 실행하면 테스트 탐색(*Discovery*)이 실행됩니다:

```
python -m unittest
```

모든 명령행 옵션 목록을 보기:

```
python -m unittest -h
```

버전 3.2에서 변경: 이전 버전에서는 개별 테스트 메서드만 실행이 가능했고, 모듈과 클래스는 불가능했습니다.

Added in version 3.14: Output is colorized by default and can be controlled using environment variables.

명령행 옵션

unittest는 다음과 같은 명령행 옵션을 제공합니다:

-b, --buffer

테스트가 실행될 동안 표준 출력과 표준 에러 스트림이 버퍼링 됩니다. 통과한 테스트 중에 나온 출력은 버려집니다. 보통 테스트 실패나 에러에서 나온 출력은 표시되고 실패 메시지에 추가됩니다.

-c, --catch

테스트 실행 중에 `Control-C`를 누르면 현재 테스트가 끝날 때까지 기다린 다음 지금까지의 모든 결과를 보고합니다. `Control-C`를 다시 누르면 일반적인 *`KeyboardInterrupt`* 예외를 발생합니다.

이 기능과 관련된 함수는 시그널 처리하기를 참고하십시오.

-f, --failfast

첫 번째 에러나 실패가 발생하면 테스트 실행을 중단합니다.

-k

패턴이나 부분 문자열과 일치하는 테스트 메서드나 클래스만 실행합니다. 이 옵션은 여러 번 사용될 수 있습니다. 이 경우 주어진 패턴 중 하나라도 일치하는 모든 테스트 케이스가 포함됩니다.

와일드카드 문자(`*`)를 포함한 패턴은 *`fnmatch.fnmatchcase()`*를 사용하여 그에 일치하는 테스트 이름을 찾고; 그렇지 않은 경우 단순히 대소문자를 구별하는 부분 문자열 일치가 사용됩니다.

패턴을 테스트 로더가 임포트한 완전히 정규화된 테스트 메서드 이름과 대조합니다.

예를 들어 `-k foo`는 `foo_tests.SomeTest.test_something`, `bar_tests.SomeTest.test_foo`에 일치하지만, `bar_tests.FooTest.test_something`에는 일치하지 않습니다.

--locals

트레이스백(traceback)에서 지역 변수를 표시합니다.

--durations `N`

Show the N slowest test cases (N=0 for all).

Added in version 3.2: 명령행 옵션인 `-b`, `-c`, `-f`가 추가되었습니다.

Added in version 3.5: 명령행 옵션 `--locals`.

Added in version 3.7: 명령행 옵션 `-k`.

Added in version 3.12: 명령행 옵션 `--durations`.

명령행은 프로젝트의 모든 테스트 또는 일부분의 테스트 탐색을 위해서도 사용할 수 있습니다.

26.5.3 테스트 탐색(Discovery)

Added in version 3.2.

unittest는 간단한 테스트 탐색을 지원합니다. 테스트 탐색에 호환되기 위해서는 모든 테스트 파일은 반드시 프로젝트의 가장 상위 디렉터리로부터 모듈 또는 패키지로 임포트 가능해야 합니다(이 말은 파일 이름이 반드시 유효한 식별자이어야 한다는 뜻입니다).

테스트 탐색은 *`TestLoader.discover()`*로 구현되어 있습니다, 그러나 명령행으로 사용할 수도 있습니다. 기본적인 명령행 사용법은 다음과 같습니다:

```
cd project_directory
python -m unittest discover
```

> **참고**
>
> 단축형인 `python -m unittest`는 `python -m unittest discover`와 같습니다. 테스트 탐색에 인자를 전달하고 싶을 때는 `discover` 부속 명령어(sub-command)를 명시적으로 사용해야 합니다.

`discover` 부-명령어는 다음과 같은 옵션을 가지고 있습니다:

-v, --verbose
상세한 출력

-s, --start-directory directory
탐색을 시작할 디렉터리(기본값 .)

-p, --pattern pattern
테스트 파일을 검색할 패턴(기본값 test*.py)

-t, --top-level-directory directory
프로젝트의 최상위 디렉터리(기본값 시작 디렉터리)

-s, *-p*, *-t* 옵션은 이 순서대로 위치 인자로서 사용할 수 있습니다. 다음 2개의 명령행은 같습니다:

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

경로가 사용되는 곳에 패키지 이름을 전달하는 것도 가능합니다, 예를 들어 myproject.subpackage.test를 시작 디렉터리로 사용할 수 있습니다. 주어진 패키지 이름은 임포트되어 그것의 파일 시스템상의 위치를 시작 디렉터리로 사용하게 됩니다.

> **조심**
>
> 테스트 탐색은 테스트를 임포트하여 로드합니다. 테스트 탐색이 당신이 지정한 시작 디렉터리로부터 모든 테스트 파일을 찾았다면 임포트하기 위해 그 파일 경로를 패키지 이름으로 바꿉니다. 예를 들어 foo/bar/baz.py는 foo.bar.baz로 임포트될 것입니다.
>
> 만약 당신이 전역적으로 설치된 패키지가 있고 테스트 탐색을 다른 패키지 복사본에 하려고 시도한다면 임포트가 잘못된 위치에서 발생할 수도 있습니다. 만약 이런 일이 발생한다면 테스트 탐색은 경고하고 종료될 것입니다.
>
> 만약 당신이 시작 디렉터리로 경로가 아닌 패키지 이름을 전달했다면 테스트 탐색은 임포트가 어느 경로로부터 되었든 간에 당신이 의도한 경로라고 간주하여 경고를 발생하지 않을 것입니다.

테스트 모듈과 패키지는 *load_tests* 프로토콜을 통하여 테스트 로드와 탐색을 사용자 정의할 수 있습니다.

버전 3.4에서 변경: Test discovery supports *namespace packages*.

버전 3.11에서 변경: Test discovery dropped the *namespace packages* support. It has been broken since Python 3.7. Start directory and its subdirectories containing tests must be regular package that have __init__.py file.

If the start directory is the dotted name of the package, the ancestor packages can be namespace packages.

버전 3.14에서 변경: Test discovery supports *namespace package* as start directory again. To avoid scanning directories unrelated to Python, tests are not searched in subdirectories that do not contain __init__.py.

26.5.4 테스트 코드 구조 잡기

단위 테스트의 기본 구성 블록은 테스트 케이스(*test cases*) — 정확성을 위해 설정되고 확인될 하나의 시나리오입니다. *unittest*에서 테스트 케이스는 *unittest.TestCase* 인스턴스에 해당합니다. 당신만의 테스트 케이스를 만들기 위해서는 *TestCase*의 서브 클래스를 작성하거나 *FunctionTestCase*를 사용해야 합니다.

TestCase 인스턴스의 테스트 코드는 완전히 독립적으로 되어 있어야 합니다, 그래야지 이것을 각각 단독으로 실행하거나 다른 여러 테스트 케이스와 함께 임의의 조합으로 실행할 수 있습니다.

가장 간단한 *TestCase*의 서브 클래스는 특정 테스트 코드를 수행하도록 단순히 테스트 메서드(즉 test로 이름이 시작하는 함수)를 구현하는 것입니다:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

어떤 것을 테스트하기 위해서는 `TestCase` 베이스 클래스에서 제공하는 *assert** 메서드 중 한 개를 사용합니다. 테스트가 실패한다면 그 이유를 설명한 메시지가 포함된 예외가 발생합니다, 그리고 `unittest`는 해당 테스트 케이스를 실패(*failure*)로 취급합니다. 다른 모든 예외는 에러(*errors*)로 취급합니다.

테스트는 매우 많지만, 그것을 위한 사전 설정은 계속 반복될 수 있습니다. 다행히, `setUp()`이란 메서드를 작성하여 사전 설정 코드를 밖으로 분리해낼 수 있습니다. 테스트 프레임워크가 1개의 테스트마다 매번 자동으로 이것을 호출할 것입니다:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

참고

다양한 테스트가 실행될 순서는 테스트 메서드의 이름을 가지고 내장된 문자열 정렬 순서에 의하여 결정될 것입니다.

만약 `setUp()` 메서드가 테스트 실행 중에 예외를 발생시킨다면 프레임워크는 테스트에 오류가 있는 것으로 간주하여 테스트 메서드를 실행하지 않을 것입니다.

마찬가지로 테스트 메서드가 실행되고 나서 정리하기 위해 `tearDown()` 메서드를 제공합니다:

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

만약 `setUp()`이 성공했다면, 테스트가 성공했든 실패했든 상관없이 `tearDown()`이 실행될 것입니다.

이와 같은 테스트를 위한 실행 환경을 테스트 픽스쳐(*test fixture*)라고 부릅니다. 개별 테스트 메서드를 실행하기 위해 고유한 테스트 픽스쳐에 해당하는 새로운 테스트 케이스 인스턴스가 생성됩니다. 따라서 `setUp()`, `tearDown()`, `__init__()`는 테스트 당 1번씩 실행됩니다.

테스트하려는 기능에 따라 테스트들을 같이 모아서 테스트 케이스 구현을 사용하는 것을 추천합니다. 이것을 위해 `unittest`는 메커니즘을 제공합니다: 테스트 묶음(*test suite*), 이것은 `unittest`의 `TestSuite` 클래스에 해당합니다. 대부분의 경우 `unittest.main()`이 테스트를 실행하기 위해 모듈의 모든 테스트 케이스를 수집하여 적절한 행동을 취할 것입니다.

그러나 당신이 테스트 묶음을 사용자 정의하고 싶다면 그것을 직접 만들어야 합니다:

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

당신은 테스트 케이스와 테스트 묶음의 정의를 테스트하려는 코드와 같은 모듈(예를 들어 `widget.py`)에 넣을 수 있습니다, 그러나 테스트 코드를 분리된 모듈(예를 들어 `test_widget.py`)에 넣으면 몇 가지 이점이 있습니다:

- 테스트 모듈이 명령행에서 독립적으로 작동할 수 있습니다.
- 테스트 코드가 배포될 코드와 쉽게 분리될 수 있습니다.
- 충분한 이유 없이 테스트하려는 코드에 맞춰서 테스트 코드를 바꾸려는 유혹이 덜 합니다.
- 테스트 코드가 테스트하려는 코드에 비해 훨씬 덜 빈번하게 수정되어야 합니다.
- 테스트하려는 코드는 더 쉽게 리팩토링할 수 있습니다.
- C 언어로 작성된 모듈의 테스트 코드는 반드시 분리된 모듈에 위치해야 합니다, 따라서 일관성을 지키는 것이 어떨까요?
- 만약 테스트 전략이 바뀌더라도 소스 코드를 바꿀 필요가 없습니다.

26.5.5 이전의 테스트 코드를 다시 사용하기

어떤 사용자들은 이전의 모든 테스트 함수를 *TestCase* 서브 클래스로 변경하는 작업 없이 기존의 테스트 코드를 *unittest*로 실행하고 싶어 할 것입니다.

이러한 이유로 *unittest*는 *FunctionTestCase* 클래스를 제공합니다. 이 *TestCase*의 서브 클래스는 기존 테스트 함수를 감싸는데 사용할 수 있습니다. 사전 설정과 정리 함수 또한 같이 사용할 수 있습니다.

다음과 같은 테스트 함수가 있을 때:

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

다음과 같이 동등한 테스트 케이스 인스턴스를 생성할 수 있습니다, 추가로 사전 설정과 정리 메서드를 함께 설정합니다:

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

> **참고**
>
> *FunctionTestCase*를 사용하여 기존 테스트를 *unittest*-기반 시스템으로 빠르게 변경할 수 있을지라도 이 방법을 추천하지는 않습니다. 시간을 들여서 적절한 *TestCase* 서브 클래스를 설정하는 것이 미래에 있을 테스트 리팩토링을 대단히 쉽게 만들어줄 것입니다.

어떤 경우에는 *doctest* 모듈을 사용하여 기존 테스트가 작성되었을 수도 있습니다. 만약 그렇다면 *doctest*가 제공하는 `DocTestSuite` 클래스를 사용하여 기존의 *doctest*-기반 테스트로부터 *unittest.TestSuite* 인스턴스를 자동으로 만들 수 있습니다.

26.5.6 테스트 건너뛰기와 예상된 실패

Added in version 3.1.

unittest는 테스트 중에서 개별 테스트 메서드나 심지어 전체 클래스를 건너뛸 수 있는 기능을 제공합니다. 게다가 테스트를 "예상된 실패"로 표시하는 기능도 지원합니다, 테스트가 망가져서 실패하더라도 그것을 *TestResult*에 실패라고 기록하지 않습니다.

테스트 건너뛰기는 단순히 *skip()* 데코레이터나 그것의 조건 변형 중 하나를 사용하거나, *setUp()*이나 테스트 메서드 안에서 *TestCase.skipTest()*를 호출하거나, *SkipTest*를 직접 발생시키면 됩니다.

기본적인 건너뛰기는 다음과 같습니다:

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # 특정 버전의 라이브러리에서만 작동하는 테스트.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # 윈도우 특정 테스트 코드
        pass

    def test_maybe_skipped(self):
        if not external_resource_available():
            self.skipTest("external resource not available")
        # 외부 리소스에 의존하는 테스트 코드
        pass
```

아래는 위의 예를 상세 모드로 실행했을 때의 출력입니다:

```
test_format (__main__.MyTestCase.test_format) ... skipped 'not supported in this␣
↪library version'
test_nothing (__main__.MyTestCase.test_nothing) ... skipped 'demonstrating skipping
↪'
test_maybe_skipped (__main__.MyTestCase.test_maybe_skipped) ... skipped 'external␣
↪resource not available'
test_windows_support (__main__.MyTestCase.test_windows_support) ... skipped
↪'requires Windows'

----------------------------------------------------------------------
Ran 4 tests in 0.005s

OK (skipped=4)
```

클래스도 메서드처럼 건너뛰기가 가능합니다:

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

TestCase.setUp() 또한 테스트를 건너뛸 수 있습니다. 이것은 사전 설정해야 할 자원을 사용할 수 없을 때 유용합니다.

예상된 실패는 *expectedFailure()* 데코레이터를 사용합니다.

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

자신만의 건너뛰기 데코레이터를 만들기는 쉽습니다. 테스트를 건너뛰고 싶을 때 *skip()*를 호출하도록 데코레이터를 만들면 됩니다. 다음의 데코레이터는 특정 어트리뷰트가 있는 객체가 전달되지 않으면 테스트를 건너뜁니다:

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

다음 데코레이터들과 예외는 테스트 건너뛰기와 예상된 실패를 구현합니다:

@unittest.**skip**(*reason*)

조건 없이 데코레이트된 테스트를 건너뜁니다. *reason*은 왜 이 테스트가 건너뛰어 졌는지를 설명해야 합니다.

@unittest.**skipIf**(*condition*, *reason*)

*condition*이 참이면 데코레이트된 테스트를 건너뜁니다.

@unittest.**skipUnless**(*condition*, *reason*)

*condition*이 참이 아니면 데코레이트된 테스트를 건너뜁니다.

@unittest.**expectedFailure**

테스트가 예상된 실패나 에러라는 표시를 합니다. 테스트가 (테스트 픽스쳐(*test fixture*) 메서드 중 하나가 아니라) 테스트 함수 자체에서 실패하거나 에러이면 성공으로 간주합니다. 테스트에 통과하면 실패로 간주합니다.

exception unittest.**SkipTest**(*reason*)

이 예외는 테스트를 건너뛰기 위해서 발생합니다.

보통은 이 예외를 직접 발생시키기보다는 *TestCase.skipTest()*나 건너뛰기 데코레이터를 사용할 수 있습니다.

건너뛰는 테스트는 테스트 전후로 *setUp()*이나 *tearDown()*를 실행하지 않을 것입니다. 건너뛰는 클래스는 *setUpClass()*나 *tearDownClass()*를 실행하지 않을 것입니다. 건너뛰는 모듈은 *setUpModule()*이나 *tearDownModule()*을 실행하지 않을 것입니다.

26.5.7 부분 테스트(subtest)를 사용하여 테스트 반복 구별 짓기

Added in version 3.4.

여러분의 테스트들이 아주 작은 부분에서만 다를 때, 예를 들어 몇몇 매개변수, unittest는 *subTest()* 컨텍스트 관리자를 사용하여 테스트 메서드의 바디 안에서 그것들은 구별 짓게 해줍니다.

예를 들어, 다음 테스트는:

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        0과 5 사이의 숫자가 모두 짝수인지 테스트합니다.
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

다음의 출력을 만듭니다:

```
======================================================================
FAIL: test_even (__main__.NumbersTest.test_even) (i=1)
Test that numbers between 0 and 5 are all even.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest.test_even) (i=3)
Test that numbers between 0 and 5 are all even.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest.test_even) (i=5)
Test that numbers between 0 and 5 are all even.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 11, in test_even
    self.assertEqual(i % 2, 0)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 1 != 0
```

부분 테스트를 사용하지 않는다면 테스트 실행은 첫 번째 실패 후에 중단될 것이고 `i` 값이 표시되지 않기 때문에 에러를 진단하는 데 쉽지 않을 것입니다:

```
======================================================================
FAIL: test_even (__main__.NumbersTest.test_even)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

26.5.8 클래스와 함수

이 절은 *unittest*의 API를 심도 있게 설명합니다.

테스트 케이스

class unittest.**TestCase**(*methodName='runTest'*)

TestCase 클래스의 인스턴스는 *unittest* 세계에서 논리적인 테스트 단위에 해당합니다. 이 클래스는 베이스 클래스로 사용되며, 특정 테스트는 구상 클래스로 구현됩니다. 이 클래스는 테스트 실행자가 테스트를 실행할 수 있는 인터페이스를 구현하고 테스트 코드가 검사하고 다양한 실패를 보고할 수 있는 메서드를 구현합니다.

*TestCase*의 각 인스턴스는 하나의 베이스 메서드: *methodName*이 지정하는 이름의 메서드를 실행할 것입니다. 대부분의 *TestCase* 사용에서, 당신은 *methodName*을 바꾸거나 기본 `runTest()` 메서드를 재구현하지 않을 것입니다.

버전 3.2에서 변경: *methodName* 제공 없이도 `TestCase`를 성공적으로 인스턴스화할 수 있습니다. 이것은 대화형 인터프리터에서 `TestCase`로 쉽게 실험을 할 수 있게 합니다.

`TestCase` 인스턴스는 3가지 메서드 그룹을 제공합니다: 한 그룹은 테스트를 실행하는 데 사용되고, 다른 한 그룹은 조건을 확인하고 실패를 보고하는 테스트 구현으로 사용되고, 몇몇 조회 메서드는 테스트 자체에 관한 정보를 수집할 수 있게 해줍니다.

첫 번째 그룹(테스트 실행) 안에 메서드는:

setUp()

테스트 픽스쳐를 준비하기 위해 호출되는 메서드입니다. 이 메서드는 테스트 메서드를 호출하기 바로 직전에 호출됩니다; `AssertionError` 또는 `SkipTest`이외의 이 메서드에서 발생한 모든 예외는 테스트 실패가 아닌 오류로 간주합니다. 기본 구현은 아무것도 하지 않습니다.

tearDown()

테스트 메서드가 불리고 결과가 기록되고 나서 바로 다음에 호출되는 메서드입니다. 테스트 메서드가 예외를 발생했더라도 이 메서드는 불립니다, 따라서 서브 클래스의 구현은 내부 상태를 확인하는 데 특별히 주의를 기울여야 합니다. `AssertionError` 또는 `SkipTest`이외의 이 메서드에서 발생하는 모든 예외는 테스트 실패가 아닌 오류로 간주합니다(따라서 보고된 오류의 총 숫자가 증가합니다). 이 메서드는 테스트 메서드의 결과물에 영향받지 않고 `setUp()`이 성공했을 때만 불립니다. 기본 구현은 아무것도 하지 않습니다.

setUpClass()

개별 클래스의 테스트들이 실행되기 전에 불리는 클래스 메서드입니다. `setUpClass`는 클래스만 인자로 받아 호출되고 `classmethod()`로 데코레이트해야 합니다:

```
@classmethod
def setUpClass(cls):
    ...
```

더 자세한 것은 클래스와 모듈 픽스쳐를 보십시오.

Added in version 3.2.

tearDownClass()

개별 클래스의 테스트들이 실행되고 난 뒤에 불리는 클래스 메서드입니다. `tearDownClass`는 클래스만 인자로 받아 호출되고 `classmethod()`로 데코레이트해야 합니다:

```
@classmethod
def tearDownClass(cls):
    ...
```

더 자세한 것은 클래스와 모듈 픽스쳐를 보십시오.

Added in version 3.2.

run(*result=None*)

테스트를 실행하고, *result* 인자로 전달된 `TestResult`에 결과를 수집합니다. 만약 *result* 인자가 전달 안 되거나 `None`이라면 임시 결과 객체를 (`defaultTestResult()` 메서드를 불러서) 생성하여 사용합니다. `run()` 호출자에게 결과 객체를 반환합니다.

단순히 `TestCase` 인스턴스를 호출하는 것으로 같은 효과를 볼 수 있습니다.

버전 3.3에서 변경: 기존 버전의 `run`은 결과를 반환하지 않았습니다. 인스턴스 호출 또한 그렇지 않았습니다.

skipTest(*reason*)

테스트 메서드나 `setUp()`에서 이것을 호출하면 현재 테스트를 건너뜁니다. 자세한 정보는 테스트 건너뛰기와 예상된 실패를 보십시오.

Added in version 3.1.

subTest(*msg=None*, ***params*)

둘러싼 코드 블록을 부분 테스트로서 실행하는 컨텍스트 관리자를 반환합니다. *msg* 및 *params* 는 선택 사항이며 부분 테스트가 실패 할 때마다 표시되는 임의의 값으로 당신이 명확하게 알아보게 해줍니다.

테스트 케이스는 여러 개의 부분 테스트 선언을 포함할 수 있고, 그것들은 자유롭게 중첩될 수 있습니다.

자세한 정보는 부분 테스트(*subtest*)를 사용하여 테스트 반복 구별 짓기를 보십시오.

Added in version 3.4.

debug()

결과를 수집하지 않고 테스트를 실행합니다. 이것은 테스트에서 발생한 예외가 호출자로 전파될 수 있게 해서, 디버거 환경에서 테스트를 실행할 때 사용될 수 있습니다.

`TestCase` 클래스는 값을 검사하고 실패를 보고하기 위해 몇 개의 assert 메서드를 제공합니다. 다음 표는 보통 많이 사용되는 메서드들입니다(더 많은 assert 메서드는 표 아래를 보십시오):

메서드	검사하는 내용	추가된 버전
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2
`assertIsSubclass(a, b)`	`issubclass(a, b)`	3.14
`assertNotIsSubclass(a, b)`	`not issubclass(a, b)`	3.14

모든 assert 메서드는 *msg* 인자를 받을 수 있습니다, 만약 그것이 전달된다면 실패 시 에러 메시지로 사용됩니다(`longMessage` 도 참고하십시오). `assertRaises()`, `assertRaisesRegex()`, `assertWarns()`, `assertWarnsRegex()`는 컨텍스트 관리자로서 사용될 때만 그들에게 *msg* 키워드 인자를 전달할 수 있다는 점을 주의하십시오.

assertEqual(*first*, *second*, *msg=None*)

*first*와 *second*가 같은지 테스트합니다. 비교한 값이 같지 않으면 테스트는 실패할 것입니다.

추가로, 만약 *first*와 *second*가 정확히 같은 형(type)이고 list, tuple, dict, set, frozenset, str 이거나 `addTypeEqualityFunc()`에 등록된 서브 클래스 형 중 하나일 경우 더 유용한 기본 에러 메시지를 생성하기 위해 형-특화(type-specific) 동등성 함수가 불릴 것입니다(형-특화 메서드 목록을 참고하십시오).

버전 3.1에서 변경: 형-특화 동등성 함수가 자동으로 불리도록 추가

버전 3.2에서 변경: 문자열 비교를 위해서 `assertMultiLineEqual()`를 기본 형-특화 동등성 함수에 추가

assertNotEqual(*first*, *second*, *msg=None*)

*first*와 *second*가 같지 않은지 테스트합니다, 비교한 값이 같으면 테스트는 실패할 것입니다.

assertTrue(*expr*, *msg=None*)

assertFalse(*expr*, *msg=None*)

*expr*이 참(또는 거짓)인지 테스트합니다.

이것은 `bool(expr) is True`와 동등하고 `expr is True`와 동등하지 않다는 것에 주의하십시오(후자를 위해선 `assertIs(expr, True)`를 사용하십시오). 더 구체적인 메서드를 사용할

수 있을 때는 이 메서드를 지양해야 합니다(예, `assertTrue(a == b)` 대신에 `assertEqual(a, b)`), 왜냐하면 실패의 경우에 구체적인 메서드가 더 나은 에러 메시지를 제공하기 때문입니다.

assertIs(*first*, *second*, *msg=None*)

assertIsNot(*first*, *second*, *msg=None*)

*first*와 *second*가 같은 객체인지 (혹은 아닌지) 테스트합니다.

Added in version 3.1.

assertIsNone(*expr*, *msg=None*)

assertIsNotNone(*expr*, *msg=None*)

*expr*이 `None` 인지(아닌지) 테스트합니다.

Added in version 3.1.

assertIn(*member*, *container*, *msg=None*)

assertNotIn(*member*, *container*, *msg=None*)

*member*가 *container* 안에 있는지(아닌지) 테스트합니다.

Added in version 3.1.

assertIsInstance(*obj*, *cls*, *msg=None*)

assertNotIsInstance(*obj*, *cls*, *msg=None*)

*obj*가 *cls*(`isinstance()`가 지원하는 것처럼 클래스 또는 클래스의 튜플)의 인스턴스인지(아닌지) 테스트합니다. 정확한 형 검사를 위해서는 `assertIs(type(obj), cls)`를 사용하십시오.

Added in version 3.2.

assertIsSubclass(*cls*, *superclass*, *msg=None*)

assertNotIsSubclass(*cls*, *superclass*, *msg=None*)

Test that *cls* is (or is not) a subclass of *superclass* (which can be a class or a tuple of classes, as supported by `issubclass()`). To check for the exact type, use `assertIs(cls, superclass)`.

Added in version 3.14.

다음의 메서드를 사용하여 예외, 경고, 로그 메시지의 발생을 검사할 수 있습니다:

메서드	검사하는 내용	추가된 버전
`assertRaises(exc, fun, *args, **kwds)`	`fun(*args, **kwds)` 가 *exc*를 발생	
`assertRaisesRegex(exc, r, fun, *args, **kwds)`	`fun(*args, **kwds)` 가 *exc*를 발생하고 메시지가 정규식 *r*에 일치	3.1
`assertWarns(warn, fun, *args, **kwds)`	`fun(*args, **kwds)` 가 *warn*을 발생	3.2
`assertWarnsRegex(warn, r, fun, *args, **kwds)`	`fun(*args, **kwds)` 가 *warn*을 발생하고 메시지가 정규식 *r*에 일치	3.2
`assertLogs(logger, level)`	`with` 블록이 최소 *level*로 *logger*에 로그를 남김	3.4
`assertNoLogs(logger, level)`	**with 블록이** 최소 *level*로 *logger*에 로그를 남기지 않음	3.10

assertRaises(*exception*, *callable*, **args*, ***kwds*)

assertRaises(*exception*, *, *msg=None*)

`assertRaises()`에 전달된 어떤 위치 또는 키워드 인자와 함께 *callable*이 호출되었을 때 예외가 발생하는지 테스트합니다. *exception*이 발생하면 테스트를 통과하고, 다른 예외가 발생하면 에러이고, 아무 예외도 발생하지 않으면 실패입니다. 여러 예외 모음을 잡기 위해서 예외 클래스를 포함한 튜플을 *exception*으로 전달해도 좋습니다.

만약 선택적인 *msg*와 함께 오직 *exception* 인자만 전달된다면, 테스트할 코드를 함수가 아닌 인라인으로 작성할 수 있도록 컨텍스트 관리자를 반환합니다:

```
with self.assertRaises(SomeException):
    do_something()
```

컨텍스트 관리자로 사용되면, `assertRaises()`는 추가적인 키워드 인자인 *msg*를 받을 수 있습니다.

컨텍스트 관리자는 잡은 예외 객체를 `exception` 어트리뷰트에 저장할 것입니다. 이것은 발생한 예외에 대해서 추가적인 검사를 수행하려는 경우에 유용할 수 있습니다:

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

버전 3.1에서 변경: `assertRaises()`를 컨텍스트 관리자로 사용할 수 있도록 기능 추가.

버전 3.2에서 변경: `exception` 어트리뷰트 추가.

버전 3.3에서 변경: 컨텍스트 관리자로 사용될 때 *msg* 키워드 인자 추가.

assertRaisesRegex(*exception*, *regex*, *callable*, **args*, ***kwds*)

assertRaisesRegex(*exception*, *regex*, *, *msg=None*)

`assertRaises()`와 비슷하지만 발생한 예외의 문자열 표현이 *regex*에 일치하는지 테스트합니다. *regex*는 정규식 객체나 `re.search()`에 사용되기 적합한 정규식 문자열이 될 수 있습니다. 예:

```
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'$",
                       int, 'XYZ')
```

또는:

```
with self.assertRaisesRegex(ValueError, 'literal'):
   int('XYZ')
```

Added in version 3.1: `assertRaisesRegexp` 라는 이름으로 추가되었습니다.

버전 3.2에서 변경: `assertRaisesRegex()`으로 이름 변경.

버전 3.3에서 변경: 컨텍스트 관리자로 사용될 때 *msg* 키워드 인자 추가.

assertWarns(*warning*, *callable*, **args*, ***kwds*)

assertWarns(*warning*, *, *msg=None*)

`assertWarns()`에 전달된 어떤 위치 또는 키워드 인자와 함께 *callable*이 호출되었을 때 경고(warning)가 발생하는지 테스트합니다. *warning*이 발생하면 테스트를 통과하고, 그렇지 않으면 실패입니다. 예외가 발생하면 에러입니다. 여러 경고 모음을 잡기 위해서 경고 클래스를 포함한 튜플을 *warnings*로 전달해도 좋습니다.

만약 선택적인 *msg*와 함께 오직 *warning* 인자만 전달된다면, 테스트할 코드를 함수가 아닌 인라인으로 작성할 수 있도록 컨텍스트 관리자를 반환합니다:

```
with self.assertWarns(SomeWarning):
    do_something()
```

컨텍스트 관리자로 사용되면, `assertWarns()`는 추가적인 키워드 인자인 *msg*를 받을 수 있습니다.

컨텍스트 관리자는 잡은 경고 객체를 `warning` 어트리뷰트에 저장하고, 경고를 발생한 소스코드 줄을 `filename`과 `lineno`에 저장할 것입니다. 이것은 발생한 경고에 대해서 추가적인 검사를 수행하려는 경우에 유용할 수 있습니다:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()

self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

이 메서드는 호출될 때 적용될 경고 필터와 관계없이 작동합니다.

Added in version 3.2.

버전 3.3에서 변경: 컨텍스트 관리자로 사용될 때 *msg* 키워드 인자 추가.

assertWarnsRegex(*warning*, *regex*, *callable*, **args*, ***kwds*)

assertWarnsRegex(*warning*, *regex*, **, msg=None*)

`assertWarns()`와 비슷하지만 발생한 경고의 메시지가 *regex*에 일치하는지 테스트합니다. *regex*는 정규식 객체나 `re.search()`에 사용되기 적합한 정규식 문자열이 될 수 있습니다. 예:

```
self.assertWarnsRegex(DeprecationWarning,
                      r'legacy_function\(\) is deprecated',
                      legacy_function, 'XYZ')
```

또는:

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
    frobnicate('/etc/passwd')
```

Added in version 3.2.

버전 3.3에서 변경: 컨텍스트 관리자로 사용될 때 *msg* 키워드 인자 추가.

assertLogs(*logger=None*, *level=None*)

최소한 *level*로 *logger*나 그 자식들에 최소한 1개의 메시지가 기록되는지 테스트하는 컨텍스트 관리자입니다.

*logger*가 주어졌다면, `logging.Logger` 객체이거나 로거의 이름인 `str`이어야 합니다. 기본값은 전파하지 않는 하위 로거에 의해 차단되지 않은 모든 메시지를 잡을 루트 로거입니다.

*level*이 주어졌다면, 로그 수준의 숫자 값이거나 그에 대응하는 문자열이어야 합니다(예를 들어 `"ERROR"`이거나 `logging.ERROR`). 기본값은 `logging.INFO`입니다.

만약 `with` 블록 안에서 *logger*와 *level* 조건을 만족하는 최소한 1개의 메시지가 나왔다면 테스트는 성공하고, 그렇지 않으면 실패합니다.

컨텍스트 관리자에 의해 반환되는 객체는 조건에 일치하는 로그 메시지를 추적하기 위한 기록 도우미입니다. 이것은 2개의 어트리뷰트를 가지고 있습니다:

records

조건에 일치하는 메시지의 `logging.LogRecord` 객체 목록.

output

조건에 일치하는 메시지의 포맷 출력인 `str` 객체 목록.

예:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

Added in version 3.4.

assertNoLogs (*logger=None, level=None*)

*logger*나 그 자식들에 주어진 *level* 이상으로 메시지가 기록되지 않았음을 테스트하는 컨텍스트 관리자입니다.

*logger*가 주어졌다면, `logging.Logger` 객체이거나 로거의 이름인 `str`이어야 합니다. 기본값은 모든 메시지를 잡을 루트 로거입니다.

*level*이 주어졌다면, 로그 수준의 숫자 값이거나 그에 대응하는 문자열이어야 합니다(예를 들어 `"ERROR"`이거나 `logging.ERROR`). 기본값은 `logging.INFO`입니다.

Unlike `assertLogs()`, nothing will be returned by the context manager.

Added in version 3.10.

더 구체적인 검사를 수행하기 위한 또 다른 메서드가 있습니다, 아래와 같이:

메서드	검사하는 내용	추가된 버전
`assertAlmostEqual(a, b)`	`round(a-b, 7) == 0`	
`assertNotAlmostEqual(a, b)`	`round(a-b, 7) != 0`	
`assertGreater(a, b)`	`a > b`	3.1
`assertGreaterEqual(a, b)`	`a >= b`	3.1
`assertLess(a, b)`	`a < b`	3.1
`assertLessEqual(a, b)`	`a <= b`	3.1
`assertRegex(s, r)`	`r.search(s)`	3.1
`assertNotRegex(s, r)`	`not r.search(s)`	3.2
`assertCountEqual(a, b)`	순서와 상관없이 *a*와 *b*가 같은 개수의 같은 요소를 가졌는지.	3.2
`assertStartsWith(a, b)`	`a.startswith(b)`	3.14
`assertNotStartsWith(a, b)`	`not a.startswith(b)`	3.14
`assertEndsWith(a, b)`	`a.endswith(b)`	3.14
`assertNotEndsWith(a, b)`	`not a.endswith(b)`	3.14
`assertHasAttr(a, b)`	`hastattr(a, b)`	3.14
`assertNotHasAttr(a, b)`	`not hastattr(a, b)`	3.14

assertAlmostEqual (*first, second, places=7, msg=None, delta=None*)

assertNotAlmostEqual (*first, second, places=7, msg=None, delta=None*)

*first*와 *second*가 근사하게 같은지(또는 근사하게 같지 않은지) 테스트합니다. 이는 값 차이를 계산하고, 주어진 소수 자릿(*places*)수(기본값 7)로 반올림한 뒤, 0과 비교하는 것으로 이루어집니다. 이 메서드는 값을 유효 숫자 자릿수(*significant digits*)가 아닌 주어진 소수 자릿수(*decimal places*)(즉, `round()` 함수와 같이)로 반올림합니다.

만약 *places* 대신에 *delta*가 주어진다면 *first*와 *second*의 값 차이는 반드시 *delta*보다 작거나 같아야 (또는 커야) 합니다.

*delta*와 *places*가 동시에 주어지면 `TypeError`가 발생합니다.

버전 3.2에서 변경: `assertAlmostEqual()`은 같다고 비교되는 거의 동등한 객체를 자동으로 고려합니다. `assertNotAlmostEqual()`은 객체가 같다고 비교되면 자동으로 실패합니다. *delta* 키워드 인자를 추가.

assertGreater(*first*, *second*, *msg=None*)

assertGreaterEqual(*first*, *second*, *msg=None*)

assertLess(*first*, *second*, *msg=None*)

assertLessEqual(*first*, *second*, *msg=None*)

*first*를 *second*와 비교해서 각각 메서드 이름에 해당하는 >, >=, <, <= 인지 테스트합니다. 그렇지 않으면 테스트는 실패합니다:

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

Added in version 3.1.

assertRegex(*text*, *regex*, *msg=None*)

assertNotRegex(*text*, *regex*, *msg=None*)

regex 검색이 *text*에 일치하는지(아닌지) 테스트합니다. 실패의 경우, 에러 메시지는 패턴과 *text*(또는 패턴과 예상과 달리 일치한 *text*의 부분)를 포함할 것입니다. *regex*는 정규식 객체나 `re.search()`에 사용되기 적합한 정규식 문자열이 될 수 있습니다.

Added in version 3.1: `assertRegexpMatches` 라는 이름으로 추가되었습니다.

버전 3.2에서 변경: `assertRegexpMatches()` 메서드가 `assertRegex()`로 이름 변경되었습니다.

Added in version 3.2: `assertNotRegex()`.

assertCountEqual(*first*, *second*, *msg=None*)

first 시퀀스가 순서에 상관없이 *second*와 같은 요소를 포함하는지 테스트합니다. 그렇지 않은 경우, 시퀀스들의 차이를 나열한 에러 메시지가 생성됩니다.

*first*와 *second*를 비교할 때 중복된 요소는 무시하지 않습니다. 두 개의 시퀀스에 각 요소가 같은 수 만큼 있는 것을 확인합니다. `assertEqual(Counter(list(first)), Counter(list(second)))`와 같지만 해시 불가능한(unhashable) 시퀀스에도 작동합니다.

Added in version 3.2.

assertStartsWith(*s*, *prefix*, *msg=None*)

assertNotStartsWith(*s*, *prefix*, *msg=None*)

Test that the Unicode or byte string *s* starts (or does not start) with a *prefix*. *prefix* can also be a tuple of strings to try.

Added in version 3.14.

assertEndsWith(*s*, *suffix*, *msg=None*)

assertNotEndsWith(*s*, *suffix*, *msg=None*)

Test that the Unicode or byte string *s* ends (or does not end) with a *suffix*. *suffix* can also be a tuple of strings to try.

Added in version 3.14.

assertHasAttr(*obj*, *name*, *msg=None*)

assertNotHasAttr(*obj*, *name*, *msg=None*)

Test that the object *obj* has (or has not) an attribute *name*.

Added in version 3.14.

`assertEqual()` 메서드는 같은 형의 객체의 동등성 검사를 다른 형-특화 메서드에게로 보냅니다. 이러한 메서드들은 대부분의 내장 형에 대해서 이미 구현되어 있지만, `addTypeEqualityFunc()`을 사용하여 새로운 메서드를 등록하는 것도 가능합니다:

addTypeEqualityFunc(*typeobj*, *function*)

정확히 같은 (서브 클래스가 아닌) *typeobj* 형의 두 객체가 같은지 비교 검사하기 위해 `assertEqual()`한테 불리는 형-특화 메서드를 등록합니다. *function*은 반드시 2개의 위치 인자를 받아야 하고 `assertEqual()`이 그러한 것처럼 msg=None 키워드 인자를 세 번째로 받아야 합니다. 이것은 처음 2개의 매개변수가 같지 않은 것이 확인될 경우 `self.failureException(msg)`을 반드시 발생시켜야 합니다 – 에러 메시지에 유용한 정보를 제공하고 비동등성을 자세히 설명할 수 있을 것입니다.

Added in version 3.1.

`assertEqual()`에서 자동으로 사용하는 형-특화 메서드 목록은 다음 표에 정리되어 있습니다. 보통은 이 메서드를 직접 부를 필요가 없다는 것을 기억하십시오.

메서드	을 비교하기 위해	추가된 버전
`assertMultiLineEqual(a, b)`	문자열	3.1
`assertSequenceEqual(a, b)`	시퀀스	3.1
`assertListEqual(a, b)`	리스트	3.1
`assertTupleEqual(a, b)`	튜플	3.1
`assertSetEqual(a, b)`	집합 또는 불변 집합	3.1
`assertDictEqual(a, b)`	딕셔너리	3.1

assertMultiLineEqual(*first*, *second*, *msg=None*)

여러 줄 문자열인 *first*와 *second*가 같은지 테스트합니다. 같지 않을 경우 에러 메시지에 다른 부분이 강조된 두 문자열의 차이가 포함됩니다. 이 메서드는 `assertEqual()`에서 문자열을 비교할 때 기본적으로 사용됩니다.

Added in version 3.1.

assertSequenceEqual(*first*, *second*, *msg=None*, *seq_type=None*)

2개의 시퀀스가 같은지 테스트합니다. *seq_type*이 전달된 경우, *first*와 *second* 둘 다 *seq_type*의 인스턴스이어야 하고 그렇지 않은 경우 실패가 발생합니다. 시퀀스가 다른 경우, 에러 메시지는 2개 사이의 차이점을 보여주게 됩니다.

이 메서드는 `assertEqual()`에서 직접 호출되진 않지만, `assertListEqual()`와 `assertTupleEqual()`을 구현할 때 사용됩니다.

Added in version 3.1.

assertListEqual(*first*, *second*, *msg=None*)

assertTupleEqual(*first*, *second*, *msg=None*)

2개의 리스트나 튜플이 같은지 테스트합니다. 만약 같지 않다면 에러 메시지는 2개 사이의 차이점만 보여주게 됩니다. 매개변수 중 하나가 잘못된 형인 경우 에러가 발생합니다. 이 메서드는 `assertEqual()`에서 리스트와 튜플을 비교할 때 기본적으로 사용됩니다.

Added in version 3.1.

assertSetEqual(*first*, *second*, *msg=None*)

2개의 집합이 같은지 테스트합니다. 같지 않은 경우 에러 메시지는 집합 사이의 차이를 나열하게 됩니다. 이 메서드는 `assertEqual()`에서 집합이나 불변 집합을 비교할 때 기본적으로 사용됩니다.

*first*와 *second* 중 하나가 `set.difference()` 메서드를 가지고 있지 않으면 실패합니다.

Added in version 3.1.

assertDictEqual(*first*, *second*, *msg=None*)

2개의 딕셔너리가 같은지 테스트합니다. 같지 않은 경우 에러 메시지는 딕셔너리 사이의 차이를 보여주게 됩니다. 이 메서드는 `assertEqual()`에서 딕셔너리를 비교할 때 기본적으로 사용될 것입니다.

Added in version 3.1.

마지막으로 *TestCase*가 다음의 메서드와 어트리뷰트를 제공합니다:

fail(*msg=None*)

무조건 테스트 실패 신호를 보냅니다, 에러 메시지를 위해 *msg*나 `None`을 전달합니다.

failureException

이 클래스 어트리뷰트는 테스트 메서드에서 발생한 예외를 줍니다. 만약 테스트 프레임워크가 추가 정보를 전달하기 위해 특수한 예외를 사용할 필요가 있다면, 프레임워크와 "공정하게 행동하기" 위해서 이 예외를 서브 클래스해야 합니다. 이 어트리뷰트의 초깃값은 *AssertionError* 입니다.

longMessage

이 클래스 어트리뷰트는 실패한 assertXYY 호출에 msg 인자로 전달된 사용자 정의 실패 메시지가 어떻게 동작하는지를 결정합니다. `True`가 기본값입니다. 이 경우, 사용자 정의 메시지가 표준 실패 메시지 끝에 추가됩니다. `False`로 설정할 경우 사용자 정의 메시지가 표준 메시지를 대체합니다.

이 클래스 설정은 인스턴스 어트리뷰트를 설정하여 개별 테스트 메서드에 의해 재정의될 수 있습니다, assert 메서드를 호출하기 전에 self.longMessage를 `True` 또는 `False`로 설정하는 것입니다.

이 클래스 설정은 각 테스트 호출 전에 재설정됩니다.

Added in version 3.1.

maxDiff

이 어트리뷰트는 실패 시 diff를 보고하는 assert 메서드의 최대 diff 출력 길이를 설정합니다. 기본값은 80*8 문자입니다. 이 어트리뷰트에 영향을 받는 assert 메서드는 *assertSequenceEqual()*(이것에 위임하는 모든 시퀀스 비교 메서드를 포함), *assertDictEqual()*, *assertMultiLineEqual()* 입니다.

`maxDiff`를 `None`으로 설정하면 diff의 최대 길이 제한이 없어지는 것을 뜻합니다.

Added in version 3.2.

테스트 프레임워크는 테스트에 관한 정보를 수집하기 위해 다음의 메서드를 사용할 수 있습니다:

countTestCases()

이 테스트 객체에 해당하는 테스트 개수를 반환합니다. *TestCase* 인스턴스에 대해서는 이것은 항상 `1`입니다.

defaultTestResult()

이 테스트 케이스 클래스를 위해서 사용되는 테스트 결과 클래스의 인스턴스를 반환합니다(*run()* 메서드에 다른 결과 인스턴스가 전달되지 않은 경우에).

TestCase 인스턴스에 대해서는 이것은 항상 *TestResult*의 인스턴스입니다; *TestCase*의 서브 클래스는 이것을 필요에 따라 재정의해야 합니다.

id()

특정 테스트 케이스를 식별하는 문자열을 반환합니다. 이것은 보통 모듈과 클래스 이름을 포함한 테스트 메서드의 완전한 이름(full name)입니다.

shortDescription()

테스트의 설명을 반환하거나 설명이 제공되지 않았으면 `None`을 반환합니다. 이 메서드의 기본 구현은 가능하다면 테스트 메서드의 독스트링의 첫 번째 줄을 반환하고 그렇지 않으면 `None`을 반환합니다.

버전 3.1에서 변경: 3.1 버전에서 docstring이 있는 경우에도 짧은 설명에 테스트 이름을 추가하도록 변경되었습니다. 이것은 unittest 확장과 호환성 문제를 일으켰고 테스트 이름 추가는 파이썬 3.2에서 *TextTestResult*로 옮겨졌습니다.

addCleanup(*function, /, *args, **kwargs*)

테스트 중에 사용된 자원을 정리하기 위해 *tearDown()* 이후에 불리는 함수를 추가합니다. 함수들은 추가된 순서의 반대 순서대로 불리게 됩니다(LIFO). 함수가 추가될 때 *addCleanup()*에 같이 전달된 위치 인자나 키워드 인자와 함께 호출됩니다.

만약 `setUp()`이 실패한다면, 즉 `tearDown()`이 불리지 않더라도, 정리 함수들은 여전히 불리게 될 것입니다.

Added in version 3.1.

enterContext(*cm*)

Enter the supplied *context manager*. If successful, also add its `__exit__()` method as a cleanup function by `addCleanup()` and return the result of the `__enter__()` method.

Added in version 3.11.

doCleanups()

이 메서드는 `tearDown()` 이후나, `setUp()`이 예외를 발생시키면 `setUp()`이후에 조건 없이 호출됩니다.

`addCleanup()`에서 추가된 모든 정리 함수들을 호출하는 책임이 있습니다. 만약 정리 함수를 `tearDown()` 이전에 불러야 할 필요가 있다면 `doCleanups()`를 직접 부를 수 있습니다.

`doCleanups()`는 한 번에 하나씩 정리 함수 스택에서 메서드를 꺼내기 때문에 언제든지 호출될 수 있습니다.

Added in version 3.1.

classmethod addClassCleanup(*function*, /, **args*, ***kwargs*)

테스트 클래스 중에 사용된 자원을 정리하기 위해 `tearDownClass()` 이후에 불리는 함수를 추가합니다. 함수들은 추가된 순서의 반대 순서대로 불리게 됩니다(LIFO). 함수가 추가될 때 `addClassCleanup()`에 같이 전달된 위치 인자나 키워드 인자와 함께 호출됩니다.

만약 `setUpClass()`가 실패한다면, 즉 `tearDownClass()`가 불리지 않더라도, 정리 함수들은 여전히 불리게 될 것입니다.

Added in version 3.8.

classmethod enterClassContext(*cm*)

Enter the supplied *context manager*. If successful, also add its `__exit__()` method as a cleanup function by `addClassCleanup()` and return the result of the `__enter__()` method.

Added in version 3.11.

classmethod doClassCleanups()

이 메서드는 `tearDownClass()` 이후나, `setUpClass()`이 예외를 발생시키면 `setUpClass()`이 후에 조건 없이 호출됩니다.

`addClassCleanup()`에서 추가된 모든 정리 함수들을 호출하는 책임이 있습니다. 만약 정리 함수를 `tearDownClass()` 이전에 호출해야 할 필요가 있다면 `doClassCleanups()`를 직접 호출할 수 있습니다.

`doClassCleanups()`는 한 번에 하나씩 정리 함수 스택에서 메서드를 꺼내기 때문에 언제든지 호출될 수 있습니다.

Added in version 3.8.

class unittest.**IsolatedAsyncioTestCase**(*methodName='runTest'*)

이 클래스는 `TestCase`와 유사한 API를 제공하며 코루틴을 테스트 함수로 받아들입니다.

Added in version 3.8.

loop_factory

The *loop_factory* passed to `asyncio.Runner`. Override in subclasses with `asyncio.EventLoop` to avoid using the asyncio policy system.

Added in version 3.13.

async asyncSetUp()

테스트 픽스쳐를 준비하기 위해 호출되는 메서드입니다. 이 메서드는 `setUp()` 이후에 호출됩니다. 이 메서드는 테스트 메서드를 호출하기 바로 직전에 호출됩니다; `AssertionError` 또는

*SkipTest*이외의 이 메서드에서 발생한 모든 예외는 테스트 실패가 아닌 오류로 간주합니다. 기본 구현은 아무것도 하지 않습니다.

async asyncTearDown()

테스트 메서드가 불리고 결과가 기록되고 나서 바로 다음에 호출되는 메서드입니다. 이 메서드는 `tearDown()` 전에 호출됩니다. 테스트 메서드가 예외를 발생했더라도 이 메서드는 불립니다, 따라서 서브 클래스의 구현은 내부 상태를 확인하는 데 특별히 주의를 기울여야 합니다. *AssertionError* 또는 *SkipTest*이외의 이 메서드에서 발생하는 모든 예외는 테스트 실패가 아닌 오류로 간주합니다(따라서 보고된 오류의 총 숫자가 증가합니다). 이 메서드는 테스트 메서드의 결과물에 영향받지 않고 *asyncSetUp()*이 성공했을 때만 불립니다. 기본 구현은 아무것도 하지 않습니다.

addAsyncCleanup(*function*, /, **args*, ***kwargs*)

이 메서드는 정리 함수로 사용할 수 있는 코루틴을 받아들입니다.

async enterAsyncContext(*cm*)

Enter the supplied *asynchronous context manager*. If successful, also add its `__aexit__()` method as a cleanup function by *addAsyncCleanup()* and return the result of the `__aenter__()` method.

Added in version 3.11.

run(*result=None*)

테스트를 실행하기 위한 새 이벤트 루프를 설정하고, *result* 인자로 전달된 *TestResult*에 결과를 수집합니다. 만약 *result* 인자가 전달 안 되거나 `None`이라면 임시 결과 객체를 (`defaultTestResult()` 메서드를 불러서) 생성하여 사용합니다. *run()* 호출자에게 결과 객체를 반환합니다. 테스트의 끝에서 이벤트 루프의 모든 태스크는 취소됩니다.

순서를 보여주는 예:

```
from unittest import IsolatedAsyncioTestCase

events = []


class Test(IsolatedAsyncioTestCase):


    def setUp(self):
        events.append("setUp")

    async def asyncSetUp(self):
        self._async_connection = await AsyncConnection()
        events.append("asyncSetUp")

    async def test_response(self):
        events.append("test_response")
        response = await self._async_connection.get("https://example.com")
        self.assertEqual(response.status_code, 200)
        self.addAsyncCleanup(self.on_cleanup)

    def tearDown(self):
        events.append("tearDown")

    async def asyncTearDown(self):
        await self._async_connection.close()
        events.append("asyncTearDown")

    async def on_cleanup(self):
        events.append("cleanup")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
if __name__ == "__main__":
    unittest.main()
```

테스트를 실행한 후, events 에는 ["setUp", "asyncSetUp", "test_response", "asyncTearDown", "tearDown", "cleanup"] 가 포함됩니다.

class unittest.**FunctionTestCase**(*testFunc, setUp=None, tearDown=None, description=None*)

이 클래스는 테스트 실행자가 테스트를 수행할 수 있게 *TestCase* 인터페이스 일부를 구현합니다, 하지만 테스트 코드가 검사하거나 에러를 보고하는 데 사용하는 메서드를 제공하지는 않습니다. 이것은 레거시 테스트 코드를 사용하여 테스트 케이스를 생성할 때 사용할 수 있습니다, 이것은 레거시 테스트 코드가 *unittest*-기반 테스트 프레임워크에 통합될 수 있게 해줍니다.

테스트 분류

class unittest.**TestSuite**(*tests=()*)

이 클래스는 개별 테스트 케이스와 테스트 묶음의 집합체를 나타냅니다. 이 클래스는 테스트 실행자가 이것을 다른 테스트 케이스처럼 실행할 수 있기 위해 필요한 인터페이스를 제공합니다. *TestSuite* 인스턴스를 실행하는 것은 테스트 묶음을 이터레이션하면서 각 테스트를 개별적으로 실행하는 것과 같습니다.

*tests*가 주어졌다면, 그것은 초기에 이 테스트 묶음을 만들 때 사용될 개별 테스트 케이스이거나 다른 테스트 묶음의 이터러블이어야 합니다. 나중에 컬렉션에 테스트 케이스나 테스트 묶음을 추가할 수 있는 추가 메서드가 제공됩니다.

TestSuite 객체는 *TestCase* 객체와 흡사하게 행동합니다만, 테스트를 실제로 구현하지 않는 것이 다릅니다. 대신에, 이것은 다 같이 실행되어야 하는 테스트 그룹에 테스트들을 모으는 데 사용됩니다. *TestSuite* 인스턴스에 테스트를 추가하기 위해 몇몇 추가적인 메서드를 사용할 수 있습니다.

addTest(*test*)

테스트 묶음에 *TestCase*나 *TestSuite* 추가하기.

addTests(*tests*)

이 테스트 묶음에 *TestCase*와 *TestSuite* 인스턴스의 이터러블에서 나온 모든 테스트를 추가하기.

이것은 *tests*를 이터레이션하면서 각 요소에 대해 *addTest()*를 호출하는 것과 같습니다.

*TestSuite*는 다음 메서드를 *TestCase*와 공유합니다:

run(*result*)

이 테스트 묶음과 연관된 테스트를 실행하고, *result*로 전달된 테스트 결과 객체에 결과를 수집합니다. *TestCase.run()*과 달리 *TestSuite.run()*은 결과 객체가 반드시 전달되어야 합니다.

debug()

결과를 수집하지 않고 이 테스트 묶음과 연관된 테스트를 실행합니다. 이것은 테스트에서 발생한 예외가 호출자로 전파될 수 있게 해서 디버거 환경에서 테스트를 실행할 때 사용될 수 있습니다.

countTestCases()

이 테스트 객체에 해당하는 테스트 개수를 반환합니다, 모든 개별 테스트와 서브-테스트 묶음을 포함합니다.

__iter__()

*TestSuite*로 묶인 테스트들은 항상 이터레이션으로 접근합니다. 서브 클래스는 `__iter__()`를 재정의하여 테스트를 지연해서 제공할 수 있습니다. 이 메서드는 한 개의 테스트 묶음에서 여러 번 불릴 수 있다는 것을 기억하십시오(예를 들어 테스트 개수를 세거나 동등성을 위해 비교할 때), 그러므로 *TestSuite.run()* 전에 수 번의 이터레이션이 반환한 테스트들은 매 이터레이션 호출마다 반드시 같아야 합니다. *TestSuite.run()* 후에는 호출자가 테스트 참조를 보존하기 위해 `TestSuite._removeTestAtIndex()`를 재정의한 서브 클래스를 사용하는 경우가 아니라면 이 메서드에 의해 반환된 테스트에 의존하면 안 됩니다.

버전 3.2에서 변경: 이전 버전에서는 *TestSuite*가 이터레이션을 사용하기보다는 직접 테스트에 접근했습니다, 따라서 `__iter__()`를 재정의하는 것은 테스트를 제공하기에 충분하지 않았습니다.

버전 3.4에서 변경: 이전 버전에서는 *TestSuite*가 *TestSuite.run()* 후에 각 *TestCase* 의 참조를 유지했습니다. 서브 클래스는 `TestSuite._removeTestAtIndex()`를 재정의해서 이 동작을 복구할 수 있습니다.

TestSuite 객체의 전형적인 사용법은 최종 사용자 테스트 장치(harness)보다는 `TestRunner`에 의해 *run()* 메서드가 호출되는 것입니다.

테스트를 로드하고 실행하기

class unittest.**TestLoader**

TestLoader 클래스는 클래스와 모듈로부터 테스트 묶음을 생성하는 데 사용됩니다. 보통, 이 클래스의 인스턴스를 생성할 필요는 없습니다; *unittest* 모듈은 공유 가능한 *unittest.defaultTestLoader* 인스턴스를 제공합니다. 그러나 서브 클래스나 인스턴스를 사용함으로 몇몇 변경 가능한 속성을 사용자 정의할 수 있습니다.

TestLoader 객체는 다음 어트리뷰트를 가집니다:

errors

테스트를 로드하는 동안 발생한 치명적이지 않은(non-fatal) 에러 목록입니다. 어떤 시점에도 로더에 의해 재설정되지 않습니다. 치명적인 에러는 예외를 발생시키는 관련 메서드에 의해 신호가 발생하여 호출자에게 전달됩니다. 치명적이지 않은 에러는 실행 시에 원래 에러를 발생시킬 합성(synthetic) 테스트가 표시하기도 합니다.

Added in version 3.5.

TestLoader 객체는 다음 메서드를 가집니다:

loadTestsFromTestCase(*testCaseClass*)

*TestCase*에서 파생된 `testCaseClass`에 포함된 모든 테스트 케이스의 묶음을 반환합니다.

테스트 케이스 인스턴스는 *getTestCaseNames()*에 의해 이름 지어진 각 메서드를 위해 생성됩니다. 기본값은 `test`로 시작되는 메서드 이름입니다. 만약 *getTestCaseNames()*가 아무 메서드도 반환하지 않지만, `runTest()` 메서드가 구현되었다면 이 메서드를 위해서 1개의 테스트 케이스가 대신 생성됩니다.

loadTestsFromModule(*module*, *, *pattern=None*)

주어진 모듈에 포함된 모든 테스트 케이스 묶음을 반환합니다. 이 메서드는 *TestCase*에서 파생된 클래스를 찾기 위해 *module*을 검색하고 클래스에 정의된 각 테스트 메서드를 위해 클래스의 인스턴스를 생성합니다.

> **참고**
>
> *TestCase*에서 파생된 클래스의 계층 사용이 픽스처와 도우미 함수를 공유하는 데 편리할 수 있지만 직접 인스턴스화를 하도록 의도되지 않은 베이스 클래스에 테스트 메서드를 정의하는 것은 이 메서드와 잘 작동하지 않습니다. 그러나 그렇게 하는 것이 픽스처들이 다르고 서브 클래스에서 정의될 경우에는 유용할 수 있습니다.

만약 모듈이 `load_tests` 함수를 제공한다면 테스트 로드를 위해 이것을 호출할 것입니다. 이것은 모듈이 테스트 로드를 사용자 정의할 수 있도록 해줍니다. 이것은 *load_tests* 프로토콜입니다. *pattern* 인자는 `load_tests`에 세 번째 인자로 전달됩니다.

버전 3.2에서 변경: `load_tests` 지원이 추가됨.

버전 3.5에서 변경: Support for a keyword-only argument *pattern* has been added.

버전 3.12에서 변경: The undocumented and unofficial *use_load_tests* parameter has been removed.

loadTestsFromName(*name*, *module=None*)

문자열 지정자에 맞는 모든 테스트 케이스 묶음을 반환합니다.

지정자 *name*은 모듈, 테스트 케이스 클래스, 테스트 케이스 클래스에 있는 테스트 메서드, `TestSuite` 인스턴스, `TestCase`나 `TestSuite` 인스턴스를 반환하는 콜러블 객체로 해석될 수 있는 "점으로 구분된 이름(dotted name)" 입니다. 이 검사는 여기에 나열된 순서대로 적용됩니다; 즉, 테스트 케이스 클래스에 있는 메서드는 "콜러블 객체" 보다는 "테스트 케이스 클래스에 있는 테스트 메서드" 로 선택될 것입니다.

예를 들어, 만약 당신이 `TestCase`에서 파생된 클래스인 `SampleTestCase`를 포함한 `SampleTests` 모듈을 가지고 있고 그 클래스는 3개의 테스트 메서드(`test_one()`, `test_two()`, `test_three()`)를 가지고 있다면, 지정자 `'SampleTests.SampleTestCase'` 에 대해서 이 메서드는 모든 3개의 테스트 메서드를 실행할 테스트 묶음으로 반환할 것입니다. 지정자가 `'SampleTests.SampleTestCase.test_two'` 라면 이 메서드는 오직 `test_two()` 테스트 메서드를 실행할 테스트 묶음을 반환할 것입니다. 지정자는 아직 임포트되지 않은 모듈이나 패키지를 가리킬 수 있습니다; 부작용(side-effect)으로써 그것들이 임포트될 것입니다.

이 메서드는 주어진 *module*에 상대적인 *name*을 선택적으로 해석할 수 있습니다.

버전 3.5에서 변경: 만약 *name* 순회 중에 `ImportError`나 `AttributeError`가 발생한다면 실행할 때 그 에러를 일으키는 합성 테스트가 반환될 것입니다. 이 에러는 self.errors 에러 모임에 포함될 것입니다.

loadTestsFromNames(*names*, *module=None*)

`loadTestsFromName()`와 비슷하지만, 1개의 이름이 아닌 이름의 시퀀스를 받습니다. 반환값은 각 이름에 정의된 모든 테스트를 지원하는 테스트 묶음입니다.

getTestCaseNames(*testCaseClass*)

testCaseClass 안에서 찾은 메서드 이름을 정렬된 시퀀스로 반환합니다; 이 클래스는 `TestCase`의 서브 클래스이어야 합니다.

discover(*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)

지정된 시작 디렉터리부터 하위 디렉터리를 재귀적으로 순회하여 모든 테스트 모듈을 찾아, 이를 포함하는 TestSuite 객체를 반환합니다. *pattern*에 일치하는 테스트 파일만 로드될 것입니다. (셸 방식의 패턴 일치를 사용합니다.) 임포트 가능한(즉, 유효한 파이썬 식별자인) 모듈 이름만 로드될 것입니다.

모든 테스트 모듈은 반드시 프로젝트의 최상위 수준에서 임포트 가능해야 합니다. 만약 시작 디렉터리가 최상위 디렉터리가 아니라면 *top_level_dir*을 따로 지정해야 합니다.

만약 모듈 임포트가 실패한다면, 예를 들어 문법 에러로 인해, 이것은 1개의 에러로 기록되고 탐색이 계속 진행될 것입니다. 만약 `SkipTest`가 발생해서 임포트가 실패했다면, 이것은 에러가 아닌 건너뛰기로 기록될 것입니다.

만약 패키지(`__init__.py` 라는 이름의 파일을 포함한 디렉터리)를 찾으면, `load_tests` 함수가 있는지 패키지를 검사할 것입니다. 만약 존재한다면 그 패키지에 대해서 `package.load_tests(loader, tests, pattern)`가 불릴 것입니다. 만약 load_tests 함수 자체가 `loader.discover`를 호출할지라도, 테스트 탐색은 실행 중에 패키지에 대한 테스트 검사를 한 번만 실행하도록 보장합니다.

만약 `load_tests`가 존재한다면 탐색은 패키지 안을 재귀 탐색하지 않습니다. `load_tests`가 패키지 안의 모든 테스트를 로드할 책임이 있습니다.

패턴은 의도적으로 로더 어트리뷰트로 저장되지 않아 패키지가 자신에 대한 탐색을 계속할 수 있습니다.

*top_level_dir*은 내부적으로 저장되어, `discover()`의 중첩된 호출 시 기본값으로 사용됩니다. 즉, 패키지의 `load_tests`가 `loader.discover()`를 호출할 경우, 이 인자를 전달할 필요가 없습니다.

*start_dir*는 디렉터리뿐만 아니라 점으로 구분된 모듈 이름이 될 수 있습니다.

Added in version 3.2.

버전 3.4에서 변경: 임포트 할 때 `SkipTest`를 발생시키는 모듈은 에러가 아니라 건너뛰기로 기록됩니다.

*start_dir*은 이름 공간 패키지일 수 있습니다.

임포트되기 전에 경로들이 정렬되어 하부 파일 시스템의 정렬 순서가 파일 이름에 의존하지 않더라도 실행 순서가 같아지도록 합니다.

버전 3.5에서 변경: 이제 발견된 패키지는 그것의 경로가 *pattern*과 일치하는지 여부와 상관없이 `load_tests`를 검사합니다, 왜냐하면 패키지 이름이 기본 패턴과 일치하는 것이 불가능하기 때문입니다.

버전 3.11에서 변경: *start_dir* can not be a *namespace packages*. It has been broken since Python 3.7, and Python 3.11 officially removes it.

버전 3.13에서 변경: *top_level_dir* is only stored for the duration of *discover* call.

버전 3.14에서 변경: *start_dir* can once again be a *namespace package*.

`TestLoader`의 다음 어트리뷰트들은 서브 클래스나 인스턴스에 대입을 통해 구성할 수 있습니다.

testMethodPrefix

테스트 메서드로 해석할 메서드 이름의 접두사에 해당하는 문자열입니다. 기본값은 `'test'` 입니다.

이것은 `getTestCaseNames()` 과 모든 `loadTestsFrom*` 메서드에 영향을 미칩니다.

sortTestMethodsUsing

`getTestCaseNames()`와 모든 `loadTestsFrom*` 메서드에서 메서드 이름 정렬 시에 이름 비교하는 데 사용될 함수입니다.

suiteClass

테스트 목록에서 테스트 묶음을 생성하는 콜러블 객체입니다. 결과 객체에 어떤 메서드도 필요하지 않습니다. 기본값은 `TestSuite` 클래스입니다.

이것은 모든 `loadTestsFrom*` 메서드에 영향을 미칩니다.

testNamePatterns

테스트 묶음에 포함되기 위해서 테스트 메서드가 일치해야 할 유닉스 셸-방식의 와일드카드 테스트 이름 패턴 목록입니다(`-k` 옵션을 보십시오).

만약 이 어트리뷰트가 `None`(기본값)이 아니라면, 테스트 묶음에 포함될 모든 테스트 메서드는 이 목록의 패턴 중 1개와 반드시 일치해야 합니다. 이 패턴 일치는 항상 `fnmatch.fnmatchcase()`를 사용하여 수행된다는 것을 기억하십시오, 그래서 `-k` 옵션에 패턴을 건네주는 것과 달리, 간단한 부분 문자열 패턴은 `*` 와일드카드를 사용하도록 변경되어야 할 것입니다.

이것은 모든 `loadTestsFrom*` 메서드에 영향을 미칩니다.

Added in version 3.7.

class unittest.**TestResult**

어떤 테스트가 성공했고 어떤 테스트가 실패했는지에 관한 정보를 엮는데 사용되는 클래스입니다.

`TestResult` 객체는 여러 테스트의 결과들을 저장합니다. `TestCase`와 `TestSuite` 클래스는 결과가 올바르게 기록되는 것을 보장합니다; 테스트 작성자가 테스트 결과를 기록하는 것에 대해서 걱정할 필요가 없습니다.

`unittest` 위에 만들어진 테스트 프레임워크는 보고 목적으로 여러 테스트가 실행하면서 만들어낸 `TestResult` 객체에 접근하고 싶을 수도 있습니다; `TestRunner.run()` 메서드는 이 목적을 위해 `TestResult` 인스턴스를 반환합니다.

`TestResult` 인스턴스는 테스트 실행 결과를 조사할 때 관심이 생길만한 다음과 같은 어트리뷰트를 가지고 있습니다.

errors

TestCase 인스턴스와 포맷된(formatted) 트레이스백 문자열로 구성된 2-튜플을 포함하는 목록입니다. 각 튜플은 예기치 못한 예외가 발생한 테스트에 해당합니다.

failures

TestCase 인스턴스와 포맷된(formatted) 트레이스백 문자열로 구성된 2-튜플을 포함하는 목록입니다. 각 튜플은 *assert** 메서드를 사용하여 명시적으로 실패가 발생한 테스트에 해당합니다.

skipped

TestCase 인스턴스와 테스트 건너뛰기한 이유 문자열로 구성된 2-튜플을 포함하는 목록입니다.

Added in version 3.1.

expectedFailures

TestCase 인스턴스와 포맷된(formatted) 트레이스백 문자열로 구성된 2-튜플을 포함하는 목록입니다. 각 튜플은 테스트 케이스의 예상된 실패니 에러에 해당합니다.

unexpectedSuccesses

예상된 실패로 표시되었지만 성공한 *TestCase* 인스턴스를 포함하는 목록입니다.

collectedDurations

테스트 케이스 이름과 실행된 각 테스트의 소요 시간을 나타내는 실수로 구성된 2-튜플을 포함하는 목록입니다.

Added in version 3.12.

shouldStop

테스트 실행이 *stop()*에 의해 정지되어야 할 때 `True`로 설정합니다.

testsRun

이제까지 실행된 테스트 총 개수입니다.

buffer

참으로 설정하면 `sys.stdout`와 `sys.stderr`가 *startTest()*와 *stopTest()* 호출 사이에서 버퍼링될 것입니다. 수집된 출력은 테스트가 실패하거나 에러가 발생한 경우에만 실제 `sys.stdout`와 `sys.stderr`에 출력될 것입니다. 모든 출력은 실패 / 에러 메시지에도 첨부됩니다.

Added in version 3.2.

failfast

참으로 설정하면 첫 번째 실패 또는 에러에서 *stop()*이 호출될 것입니다.

Added in version 3.2.

tb_locals

참으로 설정하면 지역 변수가 트레이스백에 보일 것입니다.

Added in version 3.5.

wasSuccessful()

이제까지 실행한 모든 테스트가 성공했다면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

버전 3.4에서 변경: *expectedFailure()* 데코레이터로 표시된 테스트에서 *unexpectedSuccesses*가 있다면 `False`를 반환합니다.

stop()

shouldStop 어트리뷰트를 `True`로 설정하여 현재 실행 중인 테스트 모음을 중단해야 함을 알리기 위한 용도로 이 메서드를 부를 수 있습니다. `TestRunner` 객체는 이 신호를 존중하여 어떠한 추가 테스트 없이 반환해야 합니다.

예를 들어, 사용자가 키보드로 중단 신호를 보낼 때 테스트 프레임워크를 중단하기 위해 *TextTestRunner*가 이 기능을 사용합니다. `TestRunner` 구현을 제공하는 대화형 도구는 비슷한 방법으로 이것을 사용할 수 있습니다.

`TestResult` 클래스의 다음 메서드는 내부 자료 구조를 관리하려고 사용되고, 추가적인 보고 요구 사항을 지원하기 위해 서브 클래스에서 확장할 수도 있습니다. 이것은 테스트가 실행 중에 대화형 보고를 지원하는 도구를 만들 때 특별히 유용합니다.

startTest(*test*)

테스트 케이스 *test*가 막 실행되려 할 때 호출됩니다.

stopTest(*test*)

결과에 상관없이 테스트 케이스 *test*가 실행되고 나서 호출됩니다.

startTestRun()

모든 테스트가 실행되기 전에 1번 호출됩니다.

Added in version 3.1.

stopTestRun()

모든 테스트가 실행되고 나서 1번 호출됩니다.

Added in version 3.1.

addError(*test*, *err*)

테스트 케이스 *test*가 예기치 못한 예외를 발생한 경우 호출됩니다. *err*는 `sys.exc_info()`가 반환한 형식의 튜플입니다: `(type, value, traceback)`.

기본 구현은 `(test, formatted_err)` 튜플을 인스턴스의 `errors` 어트리뷰트에 추가합니다, 여기서 *formatted_err*는 *err*에서 파생된 포맷한 트레이스백입니다.

addFailure(*test*, *err*)

테스트 케이스 *test*가 실패 신호를 보낸 경우 호출됩니다. *err*는 `sys.exc_info()`가 반환한 형식의 튜플입니다: `(type, value, traceback)`.

기본 구현은 `(test, formatted_err)` 튜플을 인스턴스의 `failures` 어트리뷰트에 추가합니다, 여기서 *formatted_err*는 *err*에서 파생된 포맷한 트레이스백입니다.

addSuccess(*test*)

테스트 케이스 *test*가 성공하면 호출됩니다.

기본 구현은 아무것도 하지 않습니다.

addSkip(*test*, *reason*)

테스트 케이스 *test*가 건너뛰어지면 호출됩니다. *reason*은 테스트가 준 건너뛰는 이유입니다.

기본 구현은 `(test, reason)` 튜플을 인스턴스의 `skipped` 어트리뷰트에 추가합니다.

addExpectedFailure(*test*, *err*)

테스트 케이스 *test*가 실패하거나 에러지만, `expectedFailure()` 데코레이터로 표시된 경우 호출됩니다

기본 구현은 `(test, formatted_err)` 튜플을 인스턴스의 `expectedFailures` 어트리뷰트에 추가합니다, 여기서 *formatted_err*는 *err*에서 파생된 포맷한 트레이스백입니다.

addUnexpectedSuccess(*test*)

테스트 케이스 *test*가 `expectedFailure()` 데코레이터로 표시되었지만, 성공한 경우 호출됩니다.

기본 구현은 테스트를 인스턴스의 `unexpectedSuccesses` 어트리뷰트에 추가합니다.

addSubTest(*test*, *subtest*, *outcome*)

부분 테스트가 완료되었을 때 호출됩니다. *test*는 테스트 메서드에 대응하는 테스트 케이스입니다. *subtest*는 부분 테스트를 설명하는 사용자 지정 `TestCase` 인스턴스입니다.

*outcome*이 `None`이면, 부분 테스트가 성공한 것입니다. 그렇지 않으면 예외와 함께 실패한 것인데 *outcome*은 `sys.exc_info()`가 반환한 형식의 튜플입니다: `(type, value, traceback)`.

기본 구현은 결과가 성공인 경우 아무것도 하지 않고 부분 테스트의 실패를 일반적인 실패로 기록합니다.

Added in version 3.4.

addDuration(*test*, *elapsed*)

Called when the test case finishes. *elapsed* is the time represented in seconds, and it includes the execution of cleanup functions.

Added in version 3.12.

class unittest.**TextTestResult**(*stream*, *descriptions*, *verbosity*, **, durations=None*)

*TextTestRunner*에서 사용하는 *TestResult*의 구체적인 구현입니다. 서브 클래스는 인터페이스가 변경될 때 호환성을 보장하기 위해 `**kwargs`를 받아야 합니다.

Added in version 3.2.

버전 3.12에서 변경: *durations* 키워드 매개변수 추가.

unittest.**defaultTestLoader**

공유 목적의 *TestLoader* 클래스의 인스턴스입니다. 만약 *TestLoader*를 사용자 정의할 필요가 없다면, 계속 새로운 인스턴스를 생성하는 것 대신 이 인스턴스를 사용할 수 있습니다.

class unittest.**TextTestRunner**(*stream=None, descriptions=True, verbosity=1, failfast=False, buffer=False, resultclass=None, warnings=None, *, tb_locals=False, durations=None*)

결과를 스트림으로 출력하는 기본 테스트 실행자 구현입니다. 만약 *stream*이 기본값인 `None`이라면, *sys.stderr*가 출력 스트림으로 사용됩니다. 이 클래스는 몇 가지 설정 가능한 매개변수를 가지고 있지만, 본질적으로 매우 간단합니다. 테스트 묶음을 실행하는 그래픽 애플리케이션은 대안 구현을 제공해야 합니다. 이러한 구현은 unittest에 기능이 추가될 때 실행자를 만드는 인터페이스가 변하기 때문에 `**kwargs`를 받아들여야 합니다.

기본적으로 이 실행자는 *DeprecationWarning*, *PendingDeprecationWarning*, *ResourceWarning*, *ImportWarning*이 기본적으로 무시 설정이 되어 있더라도 이것들을 보여줍니다. 파이썬의 `-Wd`이나 `-Wa` 옵션(경고 제어를 보십시오)을 사용하고 *warnings*를 `None`으로 설정하여 이 동작을 오버라이드할 수 있습니다.

버전 3.2에서 변경: *warnings* 매개변수 추가.

버전 3.2에서 변경: 임포트 시간이 아닌 인스턴스화 시간에 기본 스트림이 *sys.stderr*으로 설정됩니다.

버전 3.5에서 변경: *tb_locals* 매개변수 추가.

버전 3.12에서 변경: *durations* 매개변수 추가.

_makeResult()

이 메서드는 *run()*가 사용하는 `TestResult` 인스턴스를 반환합니다. 직접 호출하게 의도되지 않았지만, 사용자 정의 `TestResult`를 제공하기 위해 서브 클래스에서 오버라이드할 수 있습니다.

`_makeResult()`는 `TextTestRunner` 생성자에 `resultclass` 인자로 전달된 클래스나 콜러블을 인스턴스화합니다. 만약 `resultclass`가 제공되지 않았다면 기본값은 *TextTestResult*입니다. 결과 클래스는 다음 인자와 함께 인스턴스화됩니다:

```
stream, descriptions, verbosity
```

run(*test*)

이 메서드는 `TextTestRunner`의 주된 공개 인터페이스입니다. 이 메서드는 *TestSuite*나 *TestCase* 인스턴스를 받습니다. *TestResult*는 *_makeResult()*를 호출하여 생성하고 테스트가 실행되며 결과가 stdout에 출력됩니다.

unittest.**main**(*module='__main__', defaultTest=None, argv=None, testRunner=None, testLoader=unittest.defaultTestLoader, exit=True, verbosity=1, failfast=None, catchbreak=None, buffer=None, warnings=None*)

*module*에서 테스트 모음을 로드하고 실행하는 명령행 프로그램입니다; 이것은 주로 편리하게 실행 가능한 테스트 모듈을 만들기 위한 것입니다. 이 함수의 가장 간단한 사용은 테스트 스크립트 마지막에 다음과 같은 줄을 포함하는 것입니다:

```
if __name__ == '__main__':
    unittest.main()
```

당신은 상세도 인자를 전달하여 좀 더 자세한 정보와 함께 테스트를 실행할 수 있습니다:

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

defaultTest 인자는 *argv*로 테스트 이름이 지정되지 않은 경우 실행될 1개의 테스트 이름이거나 테스트 이름의 이터러블입니다. 만약 이 인자가 지정되지 않거나 `None`이면서 *argv*로 테스트 이름이 지정되지 않으면 *module* 안에서 찾은 모든 테스트가 실행됩니다.

argv 인자는 프로그램에 전달된 옵션 목록이 될 수 있습니다, 첫 번째 요소는 프로그램 이름입니다. 만약 이 인자가 지정되지 않거나 `None`이면, `sys.argv` 값이 사용됩니다.

testRunner 인자는 테스트 실행자 클래스나 이미 생성된 테스트 실행자 인스턴스일 수 있습니다. 기본적으로 `main`은 실행한 테스트가 성공(0)인지 실패(1)인지를 나타내는 종료 코드와 함께 `sys.exit()`을 호출합니다. 종료 코드 5는 테스트가 실행되지 않았거나 건너뛴 경우를 나타냅니다.

testLoader 인자는 `TestLoader` 인스턴스이어야 하고 기본값은 `defaultTestLoader` 입니다.

`main`은 `exit=False` 인자를 전달하여 대화형 인터프리터에서 사용하는 것을 지원합니다. 이것은 `sys.exit()` 호출 없이 결과가 표준 출력에 표시됩니다:

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

failfast, *catchbreak*, *buffer* 매개변수는 명령행 옵션의 같은 이름과 같은 효과를 가지고 있습니다.

warnings 인자는 테스트 실행 중에 사용되어야 할 경고 필터를 지정합니다. 만약 아무 값도 지정되지 않았다면, `-W` 옵션이 **python**으로 전달된 경우(경고 제어를 보십시오)에는 `None`으로 남아 있고, 그렇지 않은 경우에는 `'default'`로 설정됩니다.

`main`을 호출하면 `result` 어트리뷰트를 가진 객체를 반환합니다. 이 어트리뷰트는 테스트 실행 결과를 포함하는 `unittest.TestResult`입니다.

버전 3.1에서 변경: *exit* 매개변수가 추가되었습니다.

버전 3.2에서 변경: *verbosity*, *failfast*, *catchbreak*, *buffer*, *warnings* 매개변수가 추가되었습니다.

버전 3.4에서 변경: *defaultTest* 매개변수가 테스트 이름의 이터러블도 받을 수 있게 바뀌었습니다.

load_tests 프로토콜

Added in version 3.2.

`load_tests` 라 불리는 함수를 구현함으로써 모듈이나 패키지는 일반 테스트 실행이나 테스트 탐색 중에 그것들로부터 테스트가 어떻게 로드될지를 사용자 정의할 수 있습니다.

만약 테스트 모듈이 `load_tests`를 정의했다면 그것은 다음 인자와 함께 `TestLoader.loadTestsFromModule()`의해 호출될 것입니다:

```
load_tests(loader, standard_tests, pattern)
```

여기서 *pattern*은 `loadTestsFromModule`에서 바로 전달된 것입니다. 기본값은 `None` 입니다.

이것은 `TestSuite`를 반환해야 합니다.

*loader*는 로딩을 실행할 `TestLoader` 인스턴스입니다. *standard_tests*는 모듈에서 기본적으로 로드될 테스트입니다. 테스트 모듈이 테스트 기본 모음에서 오직 테스트를 추가하거나 빼기를 원하는 것은 흔한 일입니다. 세 번째 인자는 테스트 탐색의 일부로서 패키지를 로드할 때 사용됩니다.

특정 `TestCase` 클래스 모음에서 테스트를 로드하는 전형적인 `load_tests` 함수는 다음과 같습니다:

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

만약 탐색이 명령행 또는 *TestLoader.discover()*로부터, 패키지가 포함된 디렉터리에서 시작된다면, load_tests를 위해 패키지 __init__.py를 검사합니다. 만약 함수가 존재하지 않으면, 탐색은 그저 다른 디렉터리인 것처럼 패키지 안을 재귀 순회할 것입니다. 그렇지 않다면, 패키지의 테스트를 위한 탐색은 다음 인자와 함께 불리는 load_tests에게 맡겨질 것입니다:

```
load_tests(loader, standard_tests, pattern)
```

이것은 패키지의 모든 테스트에 해당하는 *TestSuite*를 반환해야 합니다. (standard_tests는 오직 __init__.py로부터 수집된 테스트만 포함할 것입니다.

패턴이 load_tests로 전달되기 때문에 패키지는 테스트 검색을 계속 진행(그리고 잠재적으로 수정)할 수 있습니다. 테스트 패키지를 위해서 '아무것도 하지 않는' load_tests 함수는 다음과 같을 것입니다:

```
def load_tests(loader, standard_tests, pattern):
    # 로더 인스턴스에 캐시 된 최상위 디렉터리
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

버전 3.5에서 변경: 패키지 이름이 기본 패턴과 일치하는 것이 불가능하기 때문에 탐색이 더는 *pattern* 일치를 위해서 패키지 이름을 검사하지 않습니다.

26.5.9 클래스와 모듈 픽스쳐

클래스와 모듈 단계의 픽스쳐는 *TestSuite*에 구현되어 있습니다. 테스트 묶음이 새로운 클래스의 테스트를 만나면(만약 존재한다면) 이전 클래스의 tearDownClass() 가 호출되고, 이어 새로운 클래스의 setUpClass() 가 호출됩니다.

마찬가지로 만약 테스트가 이전 테스트와 다른 모듈의 것이라면 이전 모듈의 tearDownModule이 실행되고, 이어 새로운 모듈의 setUpModule이 호출됩니다.

모든 테스트가 실행된 뒤에 마지막으로 tearDownClass와 tearDownModule이 실행됩니다.

공유하는 픽스쳐의 경우 테스트 병렬화와 같은 [잠재적인] 기능과 잘 동작하지 않고 이것은 테스트 분리를 망가트립니다. 이것을 주의 깊게 사용해야 합니다.

unittest의 테스트 로더에 의해 생성된 테스트들의 기본 정렬 순서는 같은 모듈과 클래스의 모든 테스트를 그룹화하는 것입니다. 이것은 setUpClass / setUpModule(등) 이 클래스와 모듈별로 정확하게 1번씩 호출되게 할 것입니다. 만약 당신이 무작위로 순서를 정하여, 그래서 다른 모듈과 클래스의 테스트가 서로 인접한다면, 이 공유 픽스쳐 함수는 1번의 테스트 실행에서 여러 번 호출될 수 있습니다.

공유 픽스쳐는 비표준 정렬 순서를 사용하는 테스트 묶음과 같이 작동하는 것을 의도하지 않습니다. 공유 픽스쳐를 지원하길 원치 않는 프레임워크를 위해서 BaseTestSuite가 여전히 존재합니다.

공유 픽스쳐 함수 중 1개에서 발생한 예외가 있다면, 테스트를 에러로 보고합니다. 해당 테스트 인스턴스가 없기 때문에 에러를 나타내기 위해 _ErrorHolder 객체(*TestCase*와 같은 인터페이스를 가진)가 생성됩니다. 당신이 그저 표준 unittest의 테스트 실행자를 사용한다면 이 세부 항목은 중요하지 않습니다, 그러나 당신이 프레임워크의 저자라면 이것은 관련이 있을 수 있습니다.

setUpClass 와 tearDownClass

이것들은 반드시 클래스 메서드로 구현되어야 합니다:

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

만약 당신이 베이스 클래스의 `setUpClass`와 `tearDownClass`를 호출하고 싶다면 당신이 그것을 직접 호출해야만 합니다. *TestCase*의 구현은 비어있습니다.

만약 `setUpClass` 중에 예외가 발생한다면 클래스의 테스트는 실행되지 않고 `tearDownClass` 는 실행되지 않습니다. 건너뛴 클래스는 `setUpClass` 또는 `tearDownClass`가 실행되지 않을 것입니다. 만약 예외가 *SkipTest* 예외라면 클래스는 에러 대신 건너뛰어졌다고 보고될 것입니다.

setUpModule 과 tearDownModule

이것들은 함수로 구현되어야 합니다:

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

만약 `setUpModule` 중에 예외가 발생한다면 모듈의 테스트는 실행되지 않고 `tearDownModule` 는 실행되지 않습니다. 만약 예외가 *SkipTest* 예외라면 모듈은 에러 대신 건너뛰어졌다고 보고될 것입니다.

예외가 발생하는 경우에도 실행해야 하는 정리 코드를 추가하려면, `addModuleCleanup`을 사용하십시오:

`unittest.`**`addModuleCleanup`**(*function, /, *args, **kwargs*)

테스트 클래스 중에 사용된 자원을 정리하기 위해 *tearDownModule()* 이후에 불리는 함수를 추가합니다. 함수들은 추가된 순서의 반대 순서대로 불리게 됩니다(LIFO). 함수가 추가될 때 *addModuleCleanup()*에 같이 전달된 위치 인자나 키워드 인자와 함께 호출됩니다.

만약 *setUpModule()*이 실패한다면, 즉 *tearDownModule()*이 불리지 않더라도, 정리 함수들은 여전히 불리게 될 것입니다.

Added in version 3.8.

`unittest.`**`enterModuleContext`**(*cm*)

Enter the supplied *context manager*. If successful, also add its `__exit__()` method as a cleanup function by *addModuleCleanup()* and return the result of the `__enter__()` method.

Added in version 3.11.

`unittest.`**`doModuleCleanups`**()

이 함수는 *tearDownModule()* 이후나, *setUpModule()*이 예외를 발생시키면 *setUpModule()*이후에 조건 없이 호출됩니다.

*addModuleCleanup()*에서 추가된 모든 정리 함수들을 호출하는 책임이 있습니다. 만약 정리 함수를 *tearDownModule()* 이전에 불러야 할 필요가 있다면 *doModuleCleanups()*를 직접 부를 수 있습니다.

*doModuleCleanups()*는 한 번에 하나씩 정리 함수 스택에서 메서드를 꺼내기 때문에 언제든지 호출될 수 있습니다.

Added in version 3.8.

26.5.10 시그널 처리하기

Added in version 3.2.

unittest의 *-c/--catch* 명령행 옵션은, *unittest.main()*의 catchbreak 매개 변수와 함께, 테스트 실행 중에 control-C를 사용하기 편하게 처리하도록 합니다. 중단 시그널 잡기를 활성화 하면 control-C는 현재 실행 중인 테스트를 완료하고, 그러면 테스트 실행이 끝나고 이제까지의 모든 결과를 보고할 것입니다. 두 번째 control-c는 평소와 같이 *KeyboardInterrupt*를 발생할 것입니다.

control-c 시그널 처리기는 자체 *signal.SIGINT* 처리기를 설치하는 코드 또는 테스트와의 호환성을 유지하려고 노력합니다. 만약 unittest 처리기가 불리지만 그것이 설치된 *signal.SIGINT* 처리기가 아니면, 즉 그것이 테스트 중에 시스템에 의해 대체되고 위임된다면, 그것은 기본 처리기를 호출합니다. 이것은 설치된 처리기를 대체하고 위임하는 코드에 의해 일반적으로 기대되는 동작입니다. unittest control-c 처리를 개별 테스트 별로 비활성화하고 싶을 때는 *removeHandler()* 데코레이터를 사용할 수 있습니다.

프레임워크 작성자가 테스트 프레임워크에서 control-c 처리 기능을 활성화하기 위해 몇 가지 유틸리티 함수가 있습니다.

unittest.**installHandler**()

control-c 처리기를 설치합니다. *signal.SIGINT*를 받았을 때(보통 사용자가 control-c를 눌렀을 때의 응답으로써) 모든 등록된 결과에 *stop()*이 호출됩니다.

unittest.**registerResult**(*result*)

control-c 처리를 위해서 *TestResult* 객체를 등록합니다. 결과 등록은 그것의 약한 참조를 저장합니다, 그래서 결과가 가비지 수거되는 것을 막지 않습니다.

만약 control-c 처리가 활성화되지 않았다면 *TestResult* 객체 등록은 부작용이 없습니다, 그래서 테스트 프레임워크는 처리가 가능한지 여부와 관계없이 자신이 만든 모든 결과를 무조건 등록할 수 있습니다.

unittest.**removeResult**(*result*)

등록한 결과를 제거합니다. 결과가 제거되고 나면 control-c에 대한 응답으로 결과 객체의 *stop()*을 더는 호출하지 않게 됩니다.

unittest.**removeHandler**(*function=None*)

인자 없이 호출된 경우 이 함수는 만약 control-c 처리기가 설치되었다면 그것을 제거합니다. 또한 이 함수는 테스트 실행 중에 임시로 처리기를 제거하기 위해 테스트 데코레이터로써 사용될 수도 있습니다:

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

26.6 unittest.mock — 모의 객체 라이브러리

Added in version 3.3.

소스 코드: Lib/unittest/mock.py

---

*unittest.mock*은 파이썬에서 테스트하기 위한 라이브러리입니다. 테스트 대상 시스템의 일부를 모의 객체로 교체하고 그들이 사용된 방식에 대해 어서션(assertion)을 할 수 있습니다.

*unittest.mock*은 핵심 *Mock* 클래스를 제공하여 테스트 스위트 전체에 걸쳐 많은 스텁을 만들 필요가 없도록 합니다. 작업을 수행한 후, 사용된 메서드 / 어트리뷰트와 호출에 제공된 인자에 대한 어서션을 할 수 있습니다. 일반적인 방법으로 반환 값을 지정하고 필요한 어트리뷰트를 설정할 수도 있습니다.

또한, mock은 테스트 스코프 내에서 모듈과 클래스 수준 어트리뷰트의 패치를 처리하는 *patch()* 데코레이터를 고유한 객체 생성을 위한 *sentinel*과 함께 제공합니다. *Mock*, *MagicMock* 및 *patch()* 사용 방법에 대한 몇 가지 예는 간략 지침을 참조하십시오.

Mock은 *unittest*와 함께 사용하도록 설계되었고 많은 모킹(mocking) 프레임워크에서 사용하는 '기록(record) -> 재생(replay)' 대신 '액션(action) -> 어서션(assertion)' 패턴을 기반으로 합니다.

이전 버전의 파이썬을 위한 *unittest.mock*의 역 이식이 있습니다, PyPI의 mock에서 제공됩니다.

26.6.1 간략 지침

*Mock*과 *MagicMock* 객체는 그것들을 액세스함에 따라 모든 어트리뷰트와 메서드를 작성하고 사용 방식에 대한 세부 사항을 저장합니다. 반환 값을 지정하거나 사용 가능한 어트리뷰트를 제한하도록 구성한 다음, 사용된 방식에 대한 어서션을 만들 수 있습니다:

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

*side_effect*를 사용하면 모의 객체(mock)가 호출될 때 예외를 발생시키는 것을 포함하여 부작용(side effect)을 수행 할 수 있습니다:

```
>>> from unittest.mock import Mock
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

Mock은 여러 가지 방법으로 구성하고 동작을 제어 할 수 있습니다. 예를 들어 *spec* 인자는 다른 객체에서 사양을 가져오도록 모의 객체를 구성합니다. spec에 존재하지 않는 모의 객체의 어트리뷰트나 메서드에 액세스하려고 시도하면 *AttributeError*로 실패합니다.

patch() 데코레이터 / 컨텍스트 관리자는 테스트 대상 모듈에서 클래스나 객체를 쉽게 모킹할 수 있도록 합니다. 지정한 객체는 테스트 중에 모의 객체(또는 다른 객체)로 치환되고 테스트가 끝나면 복원됩니다:

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

> **참고**
>
> patch 데코레이터를 중첩할 때 모의 객체는 적용한 순서와 같은 순서로 데코레이트 되는 함수로 전달됩니다 (데코레이터가 적용되는 일반적인 파이썬 순서). 이것은 아래에서 위로 감을 뜻하므로, 위의 예에서 `module.ClassName1`에 대한 모의 객체가 먼저 전달됩니다.
>
> *patch()*를 사용할 때 조회되는 이름 공간에서 객체를 패치하는 것이 중요합니다. 이것은 일반적으로 간단하지만, 간략 지침은 패치할 곳을 읽으십시오.

데코레이터 *patch()*는 with 문에서 컨텍스트 관리자로 사용할 수도 있습니다:

```
>>> with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...
>>> mock_method.assert_called_once_with(1, 2, 3)
```

스코프 도중 딕셔너리에 값을 설정하고 테스트가 종료될 때 딕셔너리를 원래 상태로 복원하기 위한 *patch.dict()*도 있습니다:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

Mock은 파이썬 매직 메서드의 모킹을 지원합니다. 매직 메서드를 사용하는 가장 쉬운 방법은 *MagicMock* 클래스를 사용하는 것입니다. 다음과 같은 것을 할 수 있도록 합니다:

```
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
'foobarbaz'
>>> mock.__str__.assert_called_with()
```

Mock을 사용하면 함수(또는 다른 Mock 인스턴스)를 매직 메서드에 대입할 수 있고 적절하게 호출될 것입니다. *MagicMock* 클래스는 모든 매직 메서드(아마도, 모든 유용한 메서드)가 미리 만들어져 있는 Mock 변형일 뿐입니다.

다음은 평범한 Mock 클래스로 매직 메서드를 사용하는 예입니다:

```
>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='wheeeeee')
>>> str(mock)
'wheeeeee'
```

테스트의 모의 객체가 대체하는 객체와 같은 api를 갖도록, 자동 사양를 사용할 수 있습니다. 자동 사양은 패치 할 *autospec* 인자를 patch에 전달하거나 *create_autospec()* 함수를 통해 수행할 수 있습니다. 자동 사양은 대체하는 객체와 같은 어트리뷰트와 메서드를 갖는 모의 객체를 만들고, 모든 함수와 (생성자를 포함한) 메서드는 실제 객체와 같은 호출 서명을 갖습니다.

이것은 모의 객체가 잘못 사용될 경우 프로덕션 코드와 같은 방식으로 실패하도록 합니다:

```
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...     pass
...
>>> mock_function = create_autospec(function, return_value='fishy')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> mock_function(1, 2, 3)
'fishy'
>>> mock_function.assert_called_once_with(1, 2, 3)
>>> mock_function('wrong arguments')
Traceback (most recent call last):
 ...
TypeError: missing a required argument: 'b'
```

*create_autospec()*이 클래스에 사용되면 __init__ 메서드의 서명을 복사하고, 콜러블 객체에 사용되면 __call__ 메서드의 서명을 복사합니다.

26.6.2 Mock 클래스

*Mock*은 코드 전체에서 스텁과 테스트 이중화의 사용을 대체하기 위한 유연한 모의 객체입니다. 모의 객체는 콜러블이고 어트리뷰트에 액세스할 때 새 모의 객체로 어트리뷰트를 만듭니다[1]. 같은 어트리뷰트에 액세스하면 항상 같은 모의 객체를 반환합니다. 모의 객체는 사용 방법을 기록하여, 코드가 모의 객체에 대해 수행한 작업에 대한 어서션을 만들 수 있도록 합니다.

*MagicMock*은 *Mock*의 서브 클래스이며, 모든 매직 메서드가 미리 만들어져 사용할 준비가 되어있습니다. 콜러블이 아닌 변형도 있어서, 콜러블이 아닌 객체를 모킹할 때 유용합니다: *NonCallableMock*과 *NonCallableMagicMock*

patch() 데코레이터를 사용하면 특정 모듈의 클래스를 *Mock* 객체로 쉽게 대체 할 수 있습니다. 기본적으로 *patch()*는 *MagicMock*을 생성합니다. *patch()*에 *new_callable* 인자를 사용하여 대체 *Mock* 클래스를 지정할 수 있습니다.

class unittest.mock.**Mock**(*spec=None, side_effect=None, return_value=DEFAULT, wraps=None, name=None, spec_set=None, unsafe=False, **kwargs*)

새로운 *Mock* 객체를 만듭니다. *Mock*은 Mock 객체의 동작을 지정하는 몇 가지 선택적 인자를 취합니다:

- *spec*: 문자열 리스트이거나 모의 객체의 사양으로 작동하는 기존 객체 (클래스나 인스턴스)일 수 있습니다. 객체를 전달하면 객체에 dir을 호출하여 문자열 리스트가 형성됩니다 (지원되지 않는 매직 어트리뷰트와 메서드는 제외합니다). 이 리스트에 없는 어트리뷰트에 액세스하면 *AttributeError*가 발생합니다.

  *spec*이 (문자열 리스트 대신에) 객체이면, __class__는 spec 객체의 클래스를 반환합니다. 이것은 모의 객체가 *isinstance()* 검사를 통과할 수 있도록 합니다.

- *spec_set*: *spec*의 더 엄격한 변형. 사용되면, *spec_set*으로 전달된 객체에 없는 모의 객체의 어트리뷰트를 설정하거나 얻으려고 시도하면 *AttributeError*가 발생합니다.

- *side_effect*: Mock이 호출될 때마다 호출되는 함수. *side_effect* 어트리뷰트를 참조하십시오. 예외를 발생시키거나 반환 값을 동적으로 변경하는 데 유용합니다. 함수는 모의 객체와 같은 인자로 호출되며, *DEFAULT*를 반환하지 않는 한, 이 함수의 반환 값이 반환 값으로 사용됩니다.

  또는 *side_effect*는 예외 클래스나 인스턴스일 수 있습니다. 이 경우 모의 객체가 호출될 때 그 예외가 발생합니다.

  *side_effect*가 이터러블이면 모의 객체에 대한 각 호출은 이터러블의 다음 값을 반환합니다.

  *side_effect*는 None으로 설정하여 지울 수 있습니다.

- *return_value*: 모의 객체가 호출될 때 반환되는 값. 기본적으로 이것은 새로운 Mock입니다 (처음 액세스할 때 만들어집니다). *return_value* 어트리뷰트를 참조하십시오.

- *unsafe*: 기본적으로, 이름이 *assert*, *assret*, *asert*, *aseert* 또는 *assrt*로 시작하는 어트리뷰트를 액세스하면 *AttributeError*가 발생합니다. unsafe=True를 전달하면 이러한 어트리뷰트에 액세스할 수 있습니다.

[1] 유일한 예외는 매직 메서드와 어트리뷰트(앞뒤에 이중 밑줄을 가진 것)입니다. Mock은 이것을 만들지 않고 대신 *AttributeError*를 발생시킵니다. 이는 인터프리터가 종종 묵시적으로 이러한 메서드를 요청하고, 매직 메서드를 기대할 때 새 Mock 객체를 얻으면 매우 혼동되기 때문입니다. 매직 메서드 지원이 필요하면 매직 메서드를 참조하십시오.

Added in version 3.5.

- *wraps*: 모의 객체가 감쌀 항목. *wraps*가 `None`이 아니면 Mock을 호출할 때 래핑 된 객체로 호출이 전달됩니다 (실제 결과를 반환합니다). 모의 객체에 대한 어트리뷰트 액세스는 래핑 된 객체의 해당 어트리뷰트를 래핑하는 Mock 객체를 반환합니다 (따라서 존재하지 않는 어트리뷰트에 액세스하려고 시도하면 `AttributeError`가 발생합니다).

  모의 객체에 명시적으로 *return_value*가 설정되면 호출은 래핑 된 객체로 전달되지 않고 대신 *return_value*가 반환됩니다.

- *name*: 모의 객체에 이름이 있으면 모의 객체의 repr에 사용됩니다. 디버깅에 유용할 수 있습니다. 이름은 자식 모의 객체로 전파됩니다.

임의의 키워드 인자로 모의 객체를 호출할 수도 있습니다. 이것들은 모의 객체가 만들어진 후에 어트리뷰트를 설정하는 데 사용됩니다. 자세한 내용은 `configure_mock()` 메서드를 참조하십시오.

assert_called()

모의 객체가 적어도 한 번 호출되었다고 어서트 합니다.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called()
```

Added in version 3.6.

assert_called_once()

모의 객체가 정확히 한 번 호출되었다고 어서트 합니다.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
...
AssertionError: Expected 'method' to have been called once. Called 2 times.
Calls: [call(), call()].
```

Added in version 3.6.

assert_called_with(**args, **kwargs*)

이 메서드는 마지막 호출이 특정 방식으로 이루어졌음을 어서트하는 편리한 방법입니다:

```
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

assert_called_once_with(**args, **kwargs*)

모의 객체가 정확히 한 번 호출되었고 해당 호출이 지정된 인자로 이루어졌다고 어서트 합니다.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
Calls: [call('foo', bar='baz'), call('other', bar='values')].
```

assert_any_call(**args, **kwargs*)

지정된 인자로 모의 객체가 호출되었다고 어서트 합니다.

호출이 가장 최근 호출일 때만 통과하는 *assert_called_with()*와 *assert_called_once_with()*, 그리고 *assert_called_once_with()*의 경우 유일한 호출이어야 하는 것과 달리 모의 객체가 호출된 적이 있으면 어서트가 통과합니다.

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

assert_has_calls(*calls, any_order=False*)

지정된 호출로 모의 객체가 호출되었다고 어서트 합니다. calls에 대해 *mock_calls* 리스트를 검사합니다.

*any_order*가 거짓이면 호출은 순차적이어야 합니다. 지정된 호출 전후에 추가 호출이 있을 수 있습니다.

*any_order*가 참이면 호출 순서는 상관없지만, 모두 *mock_calls*에 나타나야 합니다.

```
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```

assert_not_called()

모의 객체가 호출되지 않았다고 어서트 합니다.

```
>>> m = Mock()
>>> m.hello.assert_not_called()
>>> obj = m.hello()
>>> m.hello.assert_not_called()
Traceback (most recent call last):
  ...
AssertionError: Expected 'hello' to not have been called. Called 1 times.
Calls: [call()].
```

Added in version 3.5.

reset_mock(**, return_value=False, side_effect=False*)

reset_mock 메서드는 모의 객체의 모든 호출 어트리뷰트를 재설정합니다:

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

This can be useful where you want to make a series of assertions that reuse the same object.

return_value parameter when set to `True` resets `return_value`:

```
>>> mock = Mock(return_value=5)
>>> mock('hello')
5
>>> mock.reset_mock(return_value=True)
>>> mock('hello')
<Mock name='mock()' id='...'>
```

side_effect parameter when set to `True` resets `side_effect`:

```
>>> mock = Mock(side_effect=ValueError)
>>> mock('hello')
Traceback (most recent call last):
  ...
ValueError
>>> mock.reset_mock(side_effect=True)
>>> mock('hello')
<Mock name='mock()' id='...'>
```

Note that `reset_mock()` *doesn't* clear the `return_value`, `side_effect` or any child attributes you have set using normal assignment by default.

Child mocks are reset as well.

버전 3.6에서 변경: reset_mock 함수에 두 개의 키워드 전용 인자를 추가했습니다.

mock_add_spec(*spec*, *spec_set=False*)

모의 객체에 사양을 추가합니다. *spec*은 객체이거나 문자열 리스트일 수 있습니다. *spec*의 어트리뷰트 만 모의 객체에서 어트리뷰트로 꺼낼 수 있습니다.

*spec_set*이 참이면 스펙에 있는 어트리뷰트 만 설정할 수 있습니다.

attach_mock(*mock*, *attribute*)

이름과 부모를 치환해서, 이것의 어트리뷰트(attribute)로 모의 객체(mock)를 연결합니다. 연결된 모의 객체에 대한 호출은 이것의 `method_calls`와 `mock_calls` 어트리뷰트에 기록됩니다.

configure_mock(***kwargs*)

키워드 인자를 통해 모의 객체의 어트리뷰트를 설정합니다.

표준 점 표기법과 메서드 호출에서 딕셔너리 언 패킹을 사용해서 자식 모의 객체에 어트리뷰트와 반환 값 및 부작용을 설정할 수 있습니다:

```
>>> mock = Mock()
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

모의 객체에 대한 생성자 호출에서 같은 것을 할 수 있습니다:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

*configure_mock()*은 모의 객체를 만든 후에 구성을 더 쉽게 할 수 있도록 하기 위해 존재합니다.

__dir__()

Mock 객체는 `dir(some_mock)`의 결과를 유용한 결과로 제한합니다. *spec*이 있는 모의 객체에서 이것은 모의 객체에 허용되는 모든 어트리뷰트를 포함합니다.

이 필터링이 하는 일과 해제 방법에 대해서는 *FILTER_DIR*을 참조하십시오.

_get_child_mock(***kw*)

어트리뷰트와 반환 값에 대한 자식 모의 객체를 만듭니다. 기본적으로 자식 모의 객체는 부모와 같은 형입니다. Mock의 서브 클래스는 자식 모의 객체가 만들어지는 방법을 사용자 정의하기 위해 이를 재정의할 수 있습니다.

콜러블이 아닌 모의 객체의 경우 (사용자 정의 서브 클래스 대신) 콜러블 변형이 사용됩니다.

called

모의 객체가 호출되었는지를 나타내는 불리언:

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

call_count

모의 객체가 몇 번이나 호출되었는지를 알려주는 정수:

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

return_value

모의 객체를 호출할 때 반환되는 값을 구성하려면 이것을 설정하십시오:

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

기본 반환 값은 모의 객체이며 일반적인 방식으로 구성할 수 있습니다:

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()()' id='...'>
>>> mock.return_value.assert_called_with()
```

생성자에서 *return_value*를 설정할 수도 있습니다:

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

side_effect

이것은 모의 객체가 호출될 때 호출되는 함수, 이터러블 또는 발생시킬 예외(클래스나 인스턴스)일 수 있습니다.

함수를 전달하면 모의 객체와 같은 인자로 호출되며 함수가 *DEFAULT* 싱글톤을 반환하지 않는 한 모의 객체에 대한 호출은 함수가 반환하는 것을 반환합니다. 함수가 *DEFAULT*를 반환하면 모의 객체는 (*return_value*에서) 정상값을 반환합니다.

이터러블을 전달하면, 모든 호출에서 값을 산출해야 하는 이터레이터를 얻는 데 사용됩니다. 이 값은 발생시킬 예외 인스턴스나 모의 객체 호출에서 반환될 값일 수 있습니다 (*DEFAULT* 처리는 함수 경우와 같습니다).

(API의 예외 처리를 테스트하기 위해) 예외를 발생시키는 모의 객체 예:

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

*side_effect*를 사용하여 일련의 값을 반환하기:

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

콜러블 사용하기:

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

생성자에서 *side_effect*를 설정할 수 있습니다. 다음은 모의 객체가 호출된 값에 1을 더해서 반환하는 예입니다:

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
>>> mock(-8)
-7
```

*side_effect*를 None으로 설정하면 지워집니다:

```
>>> m = Mock(side_effect=KeyError, return_value=3)
>>> m()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Traceback (most recent call last):
 ...
KeyError
>>> m.side_effect = None
>>> m()
3
```

call_args

This is either `None` (if the mock hasn' t been called), or the arguments that the mock was last called with. This will be in the form of a tuple: the first member, which can also be accessed through the `args` property, is any positional arguments the mock was called with (or an empty tuple) and the second member, which can also be accessed through the `kwargs` property, is any keyword arguments (or an empty dictionary).

```
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
None
>>> mock()
>>> mock.call_args
call()
>>> mock.call_args == ()
True
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock.call_args.args
(3, 4)
>>> mock.call_args.kwargs
{}
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args.args
(3, 4, 5)
>>> mock.call_args.kwargs
{'key': 'fish', 'next': 'w00t!'}
```

call_args_list, *method_calls* 및 *mock_calls* 리스트의 멤버와 함께 *call_args*는 *call* 객체입니다. 이들은 튜플이므로, 개별 인자를 얻고 더 복잡한 어서션을 하기 위해 언팩할 수 있습니다. *call* 튜플을 참조하십시오.

버전 3.8에서 변경: `args`와 `kwargs` 프로퍼티를 추가했습니다.

call_args_list

이것은 모의 객체에 대한 모든 호출을 순서대로 나열한 리스트입니다 (따라서 리스트의 길이는 호출된 횟수입니다). 호출이 있기 전에는 빈 리스트입니다. *call* 객체는 *call_args_list*와 비교할 호출 리스트를 편리하게 구성하는 데 사용할 수 있습니다.

```
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
[call(), call(3, 4), call(key='fish', next='w00t!')]
>>> expected = [(), ((3, 4),), ({'key': 'fish', 'next': 'w00t!'},)]
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> mock.call_args_list == expected
True
```

*call_args_list*의 멤버는 *call* 객체입니다. 이들은 개별 인자를 얻기 위해 튜플로서 언팩될 수 있습니다. *call* 튜플을 참조하십시오.

method_calls

자신에 대한 호출 추적뿐만 아니라, 모의 객체는 메서드와 어트리뷰트, 그리고 그들의 메서드와 어트리뷰트에 대한 호출도 추적합니다:

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>
>>> mock.method_calls
[call.method(), call.property.method.attribute()]
```

*method_calls*의 멤버는 *call* 객체입니다. 이들은 개별 인자를 얻기 위해 튜플로서 언팩될 수 있습니다. *call* 튜플을 참조하십시오.

mock_calls

*mock_calls*는 모의 객체, 그것의 메서드, 매직 메서드 및 반환 값 모의 객체에 대한 모든 호출을 기록합니다.

```
>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
>>> mock.second()
<MagicMock name='mock.second()' id='...'>
>>> int(mock)
1
>>> result(1)
<MagicMock name='mock()()' id='...'>
>>> expected = [call(1, 2, 3), call.first(a=3), call.second(),
... call.__int__(), call()(1)]
>>> mock.mock_calls == expected
True
```

*mock_calls*의 멤버는 *call* 객체입니다. 이들은 개별 인자를 얻기 위해 튜플로서 언팩될 수 있습니다. *call* 튜플을 참조하십시오.

참고

*mock_calls*가 기록되는 방식은 중첩된 호출이 수행될 때 상위 호출의 매개 변수가 기록되지 않아서 항상 같다고 비교됨을 뜻합니다.:

```
>>> mock = MagicMock()
>>> mock.top(a=3).bottom()
<MagicMock name='mock.top().bottom()' id='...'>
>>> mock.mock_calls
[call.top(a=3), call.top().bottom()]
>>> mock.mock_calls[-1] == call.top(a=-1).bottom()
True
```

__class__

일반적으로 객체의 __class__ 어트리뷰트는 해당 형을 반환합니다. spec이 있는 모의 객체의 경우, __class__는 대신 spec 클래스를 반환합니다. 이를 통해 모의 객체는 다음과 같이 대체 / 가장하는 객체에 대한 *isinstance()* 테스트를 통과할 수 있습니다:

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

__class__는 대입할 수 있습니다. 이를 통해 spec을 사용하지 않고도 모의 객체가 *isinstance()* 검사를 통과할 수 있습니다.:

```
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class unittest.mock.**NonCallableMock**(*spec=None, wraps=None, name=None, spec_set=None, **kwargs*)

콜러블이 아닌 *Mock* 버전. 생성자 매개 변수는 *Mock*과 같은 의미를 가지며, 콜러블이 아닌 모의 객체에서 의미가 없는 *return_value*와 *side_effect*는 예외입니다.

클래스 나 인스턴스를 spec이나 spec_set으로 사용하는 모의 객체는 *isinstance()* 테스트를 통과할 수 있습니다:

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

Mock 클래스는 매직 메서드 모킹을 지원합니다. 자세한 내용은 매직 메서드를 참조하십시오.

모의 객체 클래스와 *patch()* 데코레이터는 모두 구성을 위해 임의의 키워드 인자를 취합니다. *patch()* 데코레이터의 경우 키워드는 만들어지는 모의 객체의 생성자로 전달됩니다. 키워드 인자는 모의 객체의 어트리뷰트를 구성하기 위한 것입니다.:

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

자식 모의 객체의 반환 값과 부작용은 점 표기법을 사용하여 같은 방식으로 설정할 수 있습니다. 호출에서 점으로 구분된 이름을 직접 사용할 수 없어서 딕셔너리를 만들고 **를 사용하여 언팩해야 합니다:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

spec(또는 *spec_set*)으로 만들어진 콜러블 모의 객체는 호출을 모의 객체와 일치시킬 때 사양 객체의 서명을 검사합니다. 따라서, 위치나 이름으로 전달되었는지와 관계없이 실제 호출의 인자와 일치할 수 있습니다:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

이 것 은 *assert_called_with()*, *assert_called_once_with()*, *assert_has_calls()* 및 *assert_any_call()*에 적용됩니다. 자동 사양할 때, 모의 객체의 메서드 호출에도 적용됩니다.

버전 3.4에서 변경: spec 되거나 자동 사양된 모의 객체에 대한 서명 검사를 추가했습니다.

class unittest.mock.**PropertyMock**(**args, **kwargs*)

클래스에서 *property*나 다른 디스크립터로 사용하기 위한 모의 객체. *PropertyMock*은 __get__() 과 __set__() 메서드를 제공하므로 꺼낼 때 반환 값을 지정할 수 있습니다.

객체에서 *PropertyMock* 인스턴스를 가져오면 인자 없이 모의 객체를 호출합니다. 설정하면 설정되는 값으로 모의 객체를 호출합니다.

```
>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo', new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
...     print(this_foo.foo)
...     this_foo.foo = 6
...
mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]
```

모의 객체 어트리뷰트가 저장되는 방식으로 인해 *PropertyMock*을 모의 객체에 직접 연결할 수 없습니다. 대신 모의 객체의 형 객체에 연결할 수 있습니다:

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
3
>>> p.assert_called_once_with()
```

조심

If an *AttributeError* is raised by *PropertyMock*, it will be interpreted as a missing descriptor and __getattr__() will be called on the parent mock:

```
>>> m = MagicMock()
>>> no_attribute = PropertyMock(side_effect=AttributeError)
>>> type(m).my_property = no_attribute
>>> m.my_property
<MagicMock name='mock.my_property' id='140165240345424'>
```

See `__getattr__()` for details.

class unittest.mock.**AsyncMock**(*spec=None, side_effect=None, return_value=DEFAULT, wraps=None, name=None, spec_set=None, unsafe=False, **kwargs*)

*MagicMock*의 비동기 버전. *AsyncMock* 객체는 객체가 비동기 함수로 인식되도록 동작하고, 호출 결과는 어웨이터블 입니다.

```
>>> mock = AsyncMock()
>>> inspect.iscoroutinefunction(mock)
True
>>> inspect.isawaitable(mock())
True
```

`mock()`의 결과는 어웨이트 한 후 `side_effect`나 `return_value`를 제공하는 비동기 함수입니다:

- `side_effect`가 함수이면, 비동기 함수는 해당 함수의 결과를 반환합니다,
- `side_effect`가 예외이면, 비동기 함수는 예외를 발생시킵니다,
- `side_effect`가 이터러블이면, 비동기 함수는 이터러블의 다음 값을 반환하지만, 결과 시퀀스가 소진되면 `StopAsyncIteration`이 즉시 발생합니다,
- `side_effect`가 정의되지 않으면, 비동기 함수는 `return_value`에 의해 정의된 값을 반환하므로, 기본적으로 비동기 함수는 새 *AsyncMock* 객체를 반환합니다.

*Mock*이나 *MagicMock*의 *spec*을 비동기 함수로 설정하면 호출 후에 코루틴 객체가 반환됩니다.

```
>>> async def async_func(): pass
...
>>> mock = MagicMock(async_func)
>>> mock
<MagicMock spec='function' id='...'>
>>> mock()
<coroutine object AsyncMockMixin._mock_call at ...>
```

Mock, *MagicMock* 또는 *AsyncMock*의 *spec*을 비동기와 동기 함수가 있는 클래스로 설정하면 동기 함수를 자동으로 감지하고 그들을 *MagicMock*(부모 모의 객체가 *AsyncMock*이나 *MagicMock*일 때)이나 *Mock*(부모 모의 객체가 *Mock*일 때)으로 설정합니다. 모든 비동기 함수는 *AsyncMock*이 됩니다.

```
>>> class ExampleClass:
...     def sync_foo():
...         pass
...     async def async_foo():
...         pass
...
>>> a_mock = AsyncMock(ExampleClass)
>>> a_mock.sync_foo
<MagicMock name='mock.sync_foo' id='...'>
>>> a_mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
>>> mock = Mock(ExampleClass)
>>> mock.sync_foo
<Mock name='mock.sync_foo' id='...'>
>>> mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
```

Added in version 3.8.

assert_awaited()

모의 객체가 적어도 한 번 어웨이트 되었다고 어서트 합니다. 이것은 호출된 객체와 별개임에 유의하십시오, `await` 키워드를 반드시 사용해야 합니다:

```
>>> mock = AsyncMock()
>>> async def main(coroutine_mock):
...     await coroutine_mock
...
>>> coroutine_mock = mock()
>>> mock.called
True
>>> mock.assert_awaited()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited.
>>> asyncio.run(main(coroutine_mock))
>>> mock.assert_awaited()
```

assert_awaited_once()

모의 객체가 정확히 한 번 어웨이트 되었다고 어서트 합니다.

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_awaited_with(**args*, ***kwargs*)

마지막 어웨이트가 지정된 인자로 이루어졌다고 어서트 합니다.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_with('foo', bar='bar')
>>> mock.assert_awaited_with('other')
Traceback (most recent call last):
...
AssertionError: expected await not found.
Expected: mock('other')
Actual: mock('foo', bar='bar')
```

assert_awaited_once_with(**args*, ***kwargs*)

모의 객체가 정확히 한 번, 지정된 인자로 어웨이트 되었다고 어서트 합니다.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_any_await(**args, **kwargs*)

지정된 인자로 모의 객체가 어웨이트된 적이 있다고 어서트 합니다.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> asyncio.run(main('hello'))
>>> mock.assert_any_await('foo', bar='bar')
>>> mock.assert_any_await('other')
Traceback (most recent call last):
...
AssertionError: mock('other') await not found
```

assert_has_awaits(*calls, any_order=False*)

지정된 호출로 모의 객체가 어웨이트 되었다고 어서트 합니다. 어웨이트에 대해 `await_args_list` 리스트가 검사됩니다.

*any_order*가 거짓이면 어웨이트는 순차적이어야 합니다. 지정된 어웨이트 전이나 후에 추가 호출이 있을 수 있습니다.

*any_order*가 참이면 어웨이트 순서와 관계없이 모두 `await_args_list`에 나타나야 합니다.

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> calls = [call("foo"), call("bar")]
>>> mock.assert_has_awaits(calls)
Traceback (most recent call last):
...
AssertionError: Awaits not found.
Expected: [call('foo'), call('bar')]
Actual: []
>>> asyncio.run(main('foo'))
>>> asyncio.run(main('bar'))
>>> mock.assert_has_awaits(calls)
```

assert_not_awaited()

모의 객체가 어웨이트 된 적이 없다고 어서트 합니다.

```
>>> mock = AsyncMock()
>>> mock.assert_not_awaited()
```

reset_mock(**args, **kwargs*)

`Mock.reset_mock()`을 참조하십시오. 또한 `await_count`를 0으로, `await_args`를 None으로 설정하고, `await_args_list`를 지웁니다.

await_count

모의 객체가 몇 번 어웨이트 되었는지 추적하는 정수.

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.await_count
1
>>> asyncio.run(main())
>>> mock.await_count
2
```

await_args

이것은 None(모의 객체가 어웨이트 되지 않았을 때)이거나 모의 객체가 마지막으로 어웨이트 된 인자입니다. *Mock.call_args*와 같게 기능합니다.

```
>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
>>> mock.await_args
>>> asyncio.run(main('foo'))
>>> mock.await_args
call('foo')
>>> asyncio.run(main('bar'))
>>> mock.await_args
call('bar')
```

await_args_list

이것은 모의 객체에 대한 모든 어웨이트를 순서대로 나열한 리스트입니다 (따라서 리스트의 길이는 어웨이트 한 횟수입니다). 어웨이트 전에는 빈 리스트입니다.

```
>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
>>> mock.await_args_list
[]
>>> asyncio.run(main('foo'))
>>> mock.await_args_list
[call('foo')]
>>> asyncio.run(main('bar'))
>>> mock.await_args_list
[call('foo'), call('bar')]
```

class unittest.mock.**ThreadingMock**(*spec=None, side_effect=None, return_value=DEFAULT, wraps=None, name=None, spec_set=None, unsafe=False, *, timeout=UNSET, **kwargs*)

A version of *MagicMock* for multithreading tests. The *ThreadingMock* object provides extra methods to wait for a call to be invoked, rather than assert on it immediately.

The default timeout is specified by the timeout argument, or if unset by the *ThreadingMock.DEFAULT_TIMEOUT* attribute, which defaults to blocking (None).

You can configure the global default timeout by setting *ThreadingMock.DEFAULT_TIMEOUT*.

wait_until_called(*, *timeout=UNSET*)

모의 객체가 호출될 때까지 기다립니다.

If a timeout was passed at the creation of the mock or if a timeout argument is passed to this function, the function raises an *AssertionError* if the call is not performed in time.

```
>>> mock = ThreadingMock()
>>> thread = threading.Thread(target=mock)
>>> thread.start()
>>> mock.wait_until_called(timeout=1)
>>> thread.join()
```

wait_until_any_call_with(**args*, ***kwargs*)

지정된 인자로 모의 객체가 호출될 때까지 기다립니다.

If a timeout was passed at the creation of the mock the function raises an *AssertionError* if the call is not performed in time.

```
>>> mock = ThreadingMock()
>>> thread = threading.Thread(target=mock, args=("arg1", "arg2",), kwargs={
↪"arg": "thing"})
>>> thread.start()
>>> mock.wait_until_any_call_with("arg1", "arg2", arg="thing")
>>> thread.join()
```

DEFAULT_TIMEOUT

Global default timeout in seconds to create instances of *ThreadingMock*.

Added in version 3.13.

호출

모의 객체는 콜러블입니다. 호출은 *return_value* 어트리뷰트로 설정된 값을 반환합니다. 기본 반환 값은 새로운 Mock 객체입니다; (명시적으로나 Mock을 호출하여) 반환 값에 처음으로 액세스할 때 만들어지지만 - 저장되고 매번 같은 값이 반환됩니다.

객체에 대한 호출은 *call_args*와 *call_args_list* 같은 어트리뷰트로 기록됩니다.

*side_effect*가 설정되면 호출이 기록된 후에 호출되므로, side_effect에서 예외가 발생해도 호출은 여전히 기록됩니다.

호출될 때 모의 객체가 예외를 발생시키는 가장 간단한 방법은 *side_effect*를 예외 클래스나 인스턴스로 만드는 것입니다:

```
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
  ...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
  ...
KeyError: 'Bang!'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]
```

*side_effect*가 함수면 해당 함수가 반환하는 것이 모의 객체에 대한 호출이 반환하는 것입니다. side_effect 함수는 모의 객체와 같은 인자로 호출됩니다. 이를 통해 입력에 따라 호출의 반환 값을 동적으로 변경할 수 있습니다:

```
>>> def side_effect(value):
...     return value + 1
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

모의 객체가 여전히 기본 반환 값(새로운 모의 객체)을, 또는 설정된 반환 값을, 반환하도록 하려면 두 가지 방법이 있습니다. *side_effect* 내부에서 *return_value*를 반환하거나 *DEFAULT*를 반환하십시오:

```
>>> m = MagicMock()
>>> def side_effect(*args, **kwargs):
...     return m.return_value
...
>>> m.side_effect = side_effect
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> m.side_effect = side_effect
>>> m()
3
```

*side_effect*를 제거하고 기본 동작으로 돌아가려면, `side_effect`를 `None`으로 설정하십시오:

```
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

*side_effect*는 이터러블 객체일 수도 있습니다. 모의 객체에 대한 반복 호출은 이터러블에서 값을 반환합니다 (이터러블이 소진되고 *StopIteration*이 발생할 때까지):

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
  ...
StopIteration
```

이터러블의 멤버가 예외이면 반환되는 대신 예외를 발생시킵니다:

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
 ...
ValueError
>>> m()
66
```

어트리뷰트 삭제

모의 객체는 요청 시 어트리뷰트를 만듭니다. 이를 통해 모든 형의 객체인 것처럼 가장할 수 있습니다.

모의 객체가 *hasattr()* 호출에 대해 `False`를 반환하거나, 어트리뷰트를 가져올 때 *AttributeError*를 발생시키길 원할 수 있습니다. 모의 객체에 `spec`으로 객체를 제공하여 그렇게 할 수 있지만, 항상 편리하지는 않습니다.

어트리뷰트를 삭제하여 어트리뷰트를 "차단"합니다. 일단 삭제되면, 어트리뷰트에 액세스할 때 *AttributeError*가 발생합니다.

```
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
    ...
AttributeError: f
```

모의 객체 이름과 이름 어트리뷰트

"name"은 *Mock* 생성자에 대한 인자이므로, 모의 객체가 "name" 어트리뷰트를 가지려면 생성 시 전달할 수 없습니다. 두 가지 대안이 있습니다. 한가지 옵션은 *configure_mock()*을 사용하는 것입니다:

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

더 간단한 옵션은 모의 객체를 만든 후 단순히 "name" 어트리뷰트를 설정하는 것입니다:

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

모의 객체를 어트리뷰트로 연결하기

모의 객체를 다른 모의 객체의 어트리뷰트로 (또는 반환 값으로) 연결하면 그 모의 객체의 "자식"이 됩니다. 자식에 대한 호출은 부모의 *method_calls*와 *mock_calls* 어트리뷰트에 기록됩니다. 이는 자식 모의 객체를 구성한 다음 부모에 연결하는 데 유용합니다. 또는 자식들에 대한 모든 호출을 기록하는 부모에 여러 모의 객체를 연결하고 모의 객체 간의 호출 순서에 대한 어서션을 하는 데 유용합니다:

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

이에 대한 예외는 모의 객체에 이름이 있는 경우입니다. 어떤 이유로든 원하지 않을 때 "부모가 되는 것 (parenting)"을 방지 할 수 있습니다.

```
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

`patch()`가 만든 모의 객체에는 자동으로 이름이 지정됩니다. 부모에게 이름을 가진 모의 객체를 연결하려면 `attach_mock()` 메서드를 사용하십시오:

```
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...
>>> parent.mock_calls
[call.child1('one'), call.child2('two')]
```

26.6.3 패처

patch 데코레이터는 데코레이트 하는 함수의 스코프 내에서만 객체를 패치하는 데 사용됩니다. 예외가 발생하더라도 자동으로 패치 해제를 처리합니다. 이러한 함수는 모두 with 문에서 사용될 수 있고, 클래스 데코레이터로도 사용할 수 있습니다.

patch

> **참고**
>
> 중요한 것은 올바른 이름 공간에서 패치를 수행하는 것입니다. 패치할 곳 절을 참조하십시오.

unittest.mock.**patch**(*target, new=DEFAULT, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs*)

`patch()`는 함수 데코레이터, 클래스 데코레이터 또는 컨텍스트 관리자 역할을 합니다. 함수 본문이나 with 문 내부에서, *target*은 *new* 객체로 패치됩니다. 함수/with 문이 종료될 때 패치가 복구됩니다.

*new*가 생략되면, 대상(target)은 패치된 객체가 비동기 함수이면 `AsyncMock`으로, 그렇지 않으면 `MagicMock`으로 치환됩니다. `patch()`가 데코레이터로 사용되고 *new*가 생략되면, 만들어진 모의 객체는 데코레이트 되는 함수에 대한 추가 인자로 전달됩니다. `patch()`가 컨텍스트 관리자로 사용되면 만들어진 모의 객체는 컨텍스트 관리자에 의해 반환됩니다.

*target*은 `'package.module.ClassName'` 형식의 문자열이어야 합니다. *target*이 임포트되고 지정된 객체를 *new* 객체로 치환하므로, `patch()`를 호출하는 환경에서 *target*을 임포트 할 수 있어야 합니다. 데코레이트 하는 시간이 아니라 데코레이트 된 함수가 실행될 때 대상(target)을 임포트 합니다.

patch가 여러분을 위해 만든다면 *spec*과 *spec_set* 키워드 인자는 `MagicMock`으로 전달됩니다.

또한 `spec=True`나 `spec_set=True`를 전달하면, patch는 모킹되는 객체를 spec/spec_set 객체로 전달합니다.

*new_callable*을 사용하면 *new* 객체를 만들기 위해 호출될 다른 클래스나 콜러블 객체를 지정할 수 있습니다. 기본적으로 `AsyncMock`이 비동기 함수에 사용되고 `MagicMock`이 나머지에 사용됩니다.

*spec*의 더 강력한 형태는 *autospec*입니다. `autospec=True`를 설정하면 치환될 객체의 사양으로 모의 객체가 만들어집니다. 모의 객체의 모든 어트리뷰트는 또한 치환되는 객체의 해당 어트리뷰트의 사양을 갖습니다. 모킹되는 메서드와 함수는 인자를 검사하고 잘못된 서명으로 호출되면 `TypeError`를 발생시킵니다. 클래스를 치환하는 모의 객체의 경우, 반환 값('인스턴스')은 클래스와 같은 사양을 갖습니다. `create_autospec()` 함수와 자동 사양을 참조하십시오.

`autospec=True` 대신 `autospec=some_object`를 전달하여 치환되는 것 대신 임의의 객체를 사양으로 사용할 수 있습니다.

기본적으로 `patch()`는 존재하지 않는 어트리뷰트를 치환하지 못합니다. `create=True`를 전달하고, 어트리뷰트가 존재하지 않으면, 패치된 함수가 호출될 때 patch가 어트리뷰트를 만들고 패치된 함수가 종료된 후 다시 삭제합니다. 이는 프로덕션 코드가 실행 시간에 만드는 어트리뷰트에 대해 테스트를 작성하는 데 유용합니다. 위험할 수 있어서 기본적으로 꺼져있습니다. 이 기능을 켜면 실제로 존재하지 않는 API에 대해 통과하는 테스트를 작성할 수 있습니다!

> **참고**
>
> 버전 3.5에서 변경: 모듈에 있는 내장(builtins)을 패치하는 경우 `create=True`를 전달할 필요가 없습니다, 기본적으로 추가됩니다.

패치는 `TestCase` 클래스 데코레이터로 사용할 수 있습니다. 클래스의 각 테스트 메서드를 데코레이트 하여 작동합니다. 테스트 메서드가 공통 패치 집합을 공유할 때 관리용 코드가 줄어듭니다. `patch()`는 `patch.TEST_PREFIX`로 시작하는 메서드 이름을 조회하는 것으로 테스트를 찾습니다. 기본적으로 이것은 `'test'`인데, `unittest`가 테스트를 찾는 방식과 일치합니다. `patch.TEST_PREFIX`를 설정하여 대체 접두사를 지정할 수 있습니다.

with 문에서 patch를 컨텍스트 관리자로 사용할 수 있습니다. 여기서 패치는 with 문 다음에 들여쓰기 된 블록에 적용됩니다. "as"를 사용하면 패치된 객체는 "as" 뒤에 오는 이름으로 연결됩니다; `patch()`가 모의 객체를 만들 때 매우 유용합니다.

`patch()`는 임의의 키워드 인자를 취합니다. 이들은 만들어질 때 패치되는 객체가 비동기이면 `AsyncMock`으로, 그렇지 않으면 `Mock`으로, 또는 지정되면 *new_callable*로 전달됩니다.

`patch.dict(...)`, `patch.multiple(...)` 및 `patch.object(...)`는 대체 사용 사례에 사용할 수 있습니다.

`patch()`를 함수 데코레이터로 사용하여, 모의 객체를 만들고 데코레이트 된 함수에 전달합니다:

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

클래스를 패치하면 클래스를 `MagicMock` 인스턴스로 치환합니다. 테스트 대상 코드에서 클래스가 인스턴스 화 되면 사용될 모의 객체의 `return_value`가 됩니다.

클래스가 여러 번 인스턴스 화 되면 `side_effect`를 사용하여 매번 새 모의 객체를 반환할 수 있습니다. 또는 *return_value*를 원하는 것으로 설정할 수 있습니다.

패치된 클래스에서 인스턴스의 메서드에 대한 반환 값을 구성하려면 *return_value*에서 이를 수행해야 합니다. 예를 들면:

```
>>> class Class:
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
...
```

*spec*이나 *spec_set*을 사용하고 *patch()*가 클래스를 대체하고 있다면, 만들어진 모의 객체의 반환 값은 같은 사양을 갖습니다.

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

new_callable 인자는 생성된 모의 객체의 기본 *MagicMock*에 대한 대체 클래스를 사용하려고 할 때 유용합니다. 예를 들어, *NonCallableMock*을 사용하려면:

```
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock) as mock_thing:
...     assert thing is mock_thing
...     thing()
...
Traceback (most recent call last):
  ...
TypeError: 'NonCallableMock' object is not callable
```

또 다른 사용 사례는 객체를 *io.StringIO* 인스턴스로 교체하는 것입니다:

```
>>> from io import StringIO
>>> def foo():
...     print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

*patch()*가 여러분 대신 모의 객체를 만들 때, 보통 가장 먼저 해야 할 일은 모의 객체를 구성하는 것입니다. 이 구성 중 일부는 patch 호출에서 수행 할 수 있습니다. 호출에 전달하는 임의의 키워드는 만들어진 모의 객체의 어트리뷰트를 설정하는 데 사용됩니다:

```
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

만들어진 모의 객체 어트리뷰트뿐만 아니라 자식 모의 객체의 *return_value*와 *side_effect*와 같은 어트리뷰트도 구성 할 수 있습니다. 키워드 인자로 직접 전달하는 것은 문법적으로 유효하지 않지만, 이것들을 키로 갖는 딕셔너리는 여전히 **를 사용하여 *patch()* 호출로 확장될 수 있습니다:

```
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
  ...
KeyError
```

기본적으로, 존재하지 않는 모듈의 함수(또는 클래스의 메서드나 어트리뷰트)를 패치하려고 시도하면 *AttributeError*로 실패합니다:

```
>>> @patch('sys.non_existing_attribute', 42)
... def test():
...     assert sys.non_existing_attribute == 42
...
>>> test()
Traceback (most recent call last):
  ...
AttributeError: <module 'sys' (built-in)> does not have the attribute 'non_
↪existing_attribute'
```

그러나 *patch()* 호출에 `create=True`를 추가하면 이전 예제가 기대한 대로 작동합니다:

```
>>> @patch('sys.non_existing_attribute', 42, create=True)
... def test(mock_stdout):
...     assert sys.non_existing_attribute == 42
...
>>> test()
```

버전 3.8에서 변경: *patch()*는 이제 target이 비동기 함수면 *AsyncMock*을 반환합니다.

patch.object

patch.**object**(*target, attribute, new=DEFAULT, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs*)

모의 객체로 객체(*target*)의 이름 있는 멤버(*attribute*)를 패치합니다.

*patch.object()*는 데코레이터, 클래스 데코레이터 또는 컨텍스트 관리자로 사용할 수 있습니다. 인자 *new*, *spec*, *create*, *spec_set*, *autospec* 및 *new_callable*은 *patch()*와 같은 의미입니다. *patch()*와 마찬가지로, *patch.object()*는 만드는 모의 객체를 구성하기 위해 임의의 키워드 인자를 취합니다.

클래스 데코레이터로 사용될 때, *patch.object()*는 감쌀 메서드를 선택하는 데 `patch.TEST_PREFIX`를 사용합니다.

세 개의 인자나 두 개의 인자로 *patch.object()*를 호출 할 수 있습니다. 세 개의 인자 형식은 패치 할 객체, 어트리뷰트 이름 및 어트리뷰트를 대체 할 객체를 취합니다.

두 개의 인자 형식으로 호출하면 대체 객체를 생략하고, 모의 객체가 만들어져 데코레이트 된 함수에 대한 추가 인자로 전달됩니다:

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
>>> test()
```

spec, *create* 및 `patch.object()`에 대한 다른 인자는 `patch()`와 같은 의미입니다.

patch.dict

`patch.dict`(*in_dict*, *values=()*, *clear=False*, ***kwargs*)

Patch a dictionary, or dictionary like object, and restore the dictionary to its original state after the test, where the restored dictionary is a copy of the dictionary as it was before the test.

*in_dict*는 딕셔너리나 컨테이너와 같은 매핑일 수 있습니다. 매핑이면 적어도 가져오기(get), 설정(set) 및 삭제(delete)와 키 이터레이션을 지원해야 합니다.

*in_dict*는 딕셔너리 이름을 지정하는 문자열일 수도 있으며, 이때는 임포트 해서 가져옵니다.

*values*는 딕셔너리에 설정할 값의 딕셔너리일 수 있습니다. *values*는 `(key, value)` 쌍의 이터러블일 수도 있습니다.

*clear*가 참이면 새 값이 설정되기 전에 딕셔너리가 지워집니다.

딕셔너리에 값을 설정하기 위해 임의의 키워드 인자로 `patch.dict()`를 호출할 수도 있습니다.

버전 3.8에서 변경: `patch.dict()`는 이제 컨텍스트 관리자로 사용될 때 패치된 딕셔너리를 반환합니다.

`patch.dict()`는 컨텍스트 관리자, 데코레이터 또는 클래스 데코레이터로 사용할 수 있습니다:

```
>>> foo = {}
>>> @patch.dict(foo, {'newkey': 'newvalue'})
... def test():
...     assert foo == {'newkey': 'newvalue'}
...
>>> test()
>>> assert foo == {}
```

클래스 데코레이터로 사용될 때 `patch.dict()`는 감쌀 메서드를 선택하는 데 `patch.TEST_PREFIX`(기본값은 `'test'`)를 따릅니다:

```
>>> import os
>>> import unittest
>>> from unittest.mock import patch
>>> @patch.dict('os.environ', {'newkey': 'newvalue'})
... class TestSample(unittest.TestCase):
...     def test_sample(self):
...         self.assertEqual(os.environ['newkey'], 'newvalue')
```

테스트에 다른 접두사를 사용하려면, `patch.TEST_PREFIX`를 설정하여 패처에 다른 접두사를 알릴 수 있습니다. 값을 변경하는 방법에 대한 자세한 내용은 *TEST_PREFIX*를 참조하십시오.

`patch.dict()`를 사용하여 멤버를 딕셔너리에 추가하고 (또는 단순히 테스트에서 딕셔너리를 변경하고) 테스트가 끝나면 딕셔너리가 복원되도록 할 수 있습니다.

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}) as patched_foo:
...     assert foo == {'newkey': 'newvalue'}
...     assert patched_foo == {'newkey': 'newvalue'}
...     # You can add, update or delete keys of foo (or patched_foo, it's the same␣
↪dict)
...     patched_foo['spam'] = 'eggs'
...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> assert foo == {}
>>> assert patched_foo == {}
```

```
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
...     print(os.environ['newkey'])
...
newvalue
>>> assert 'newkey' not in os.environ
```

patch.dict() 호출에서 키워드를 사용하여 딕셔너리에 값을 설정할 수 있습니다:

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

*patch.dict()*는 실제로 딕셔너리가 아닌 딕셔너리류 객체와 함께 사용할 수 있습니다. 최소한 아이템 가져오기, 설정하기, 삭제하기 및 이터레이션이나 멤버십 검사를 지원해야 합니다. 이것은 매직 메서드 `__getitem__()`, `__setitem__()`, `__delitem__()` 및 *__iter__()*나 `__contains__()`에 해당합니다.

```
>>> class Container:
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
...     assert thing['two'] == 3
...
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

patch.multiple

patch.**multiple**(*target, spec=None, create=False, spec_set=None, autospec=None, new_callable=None, **kwargs*)

한 번의 호출로 여러 패치를 수행합니다. 패치할 객체(객체나 임포트로 객체를 가져올 문자열로)와 패치에 대한 키워드 인자를 취합니다:

```
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
    ...
```

*patch.multiple()*가 모의 객체를 만들기를 원하면 *DEFAULT*를 값으로 사용하십시오. 이 경우 만들어진 모의 객체는 키워드로 데코레이트 된 함수로 전달되며, *patch.multiple()*이 컨텍스트 관리자

로 사용될 때는 딕셔너리가 반환됩니다.

*patch.multiple()*은 데코레이터, 클래스 데코레이터 또는 컨텍스트 관리자로 사용할 수 있습니다. *spec*, *spec_set*, *create*, *autospec* 및 *new_callable* 인자는 *patch()*와 같은 의미입니다. 이 인자는 *patch.multiple()*이 수행한 모든 패치에 적용됩니다.

클래스 데코레이터로 사용될 때 *patch.multiple()*은 감쌀 메서드를 선택하는 데 `patch.TEST_PREFIX`를 사용합니다.

*patch.multiple()*가 모의 객체를 만들기를 원하면 *DEFAULT*를 값으로 사용할 수 있습니다. *patch.multiple()*을 데코레이터로 사용하면 만들어진 모의 객체가 키워드로 데코레이트 된 함수에 전달됩니다.

```
>>> thing = object()
>>> other = object()

>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()
```

*patch.multiple()*은 다른 `patch` 데코레이터와 중첩될 수 있지만, 키워드로 전달된 인자를 *patch()*가 만든 모든 표준 인자들 뒤에 넣습니다:

```
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

*patch.multiple()*가 컨텍스트 관리자로 사용되면, 컨텍스트 관리자가 반환한 값은 만들어진 모의 객체 이름을 키로 갖는 딕셔너리입니다:

```
>>> with patch.multiple('__main__', thing=DEFAULT, other=DEFAULT) as values:
...     assert 'other' in repr(values['other'])
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other
...
```

패처 메서드: start와 stop

모든 패처에는 `start()`와 `stop()` 메서드가 있습니다. 이를 통해 `setUp` 메서드나 데코레이터를 중첩하거나 with 문을 사용하지 않고 여러 패치를 수행하려는 곳에서 패치를 수행하는 것이 더 간단해집니다.

이것들을 사용하려면 *patch()*, *patch.object()* 또는 *patch.dict()*를 정상적으로 호출하고 반환된 `patcher` 객체에 대한 참조를 유지하십시오. 그런 다음 `start()`를 호출하여 패치를 제자리에 놓고 `stop()`을 사용하여 패치를 취소할 수 있습니다.

*patch()*를 사용하여 모의 객체를 만들면 `patcher.start` 호출로 반환됩니다.

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> assert module.ClassName is not original
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

일반적인 사용 사례는 *TestCase*의 setUp 메서드에서 여러 패치를 수행하는 것입니다:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

조심

이 기법을 사용하는 경우 stop을 호출하여 패치가 "실행 취소" 되도록 해야 합니다. setUp에서 예외가 발생하면 tearDown이 호출되지 않기 때문에, 이것은 생각보다 까다로울 수 있습니다. *unittest.TestCase.addCleanup()*은 이것을 더 쉽게 만듭니다:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
```

추가 보너스로 더는 patcher 객체에 대한 참조를 유지할 필요가 없습니다.

*patch.stopall()*을 사용하여 시작된 모든 패치를 중지할 수도 있습니다.

patch.**stopall**()

모든 활성 패치를 중지합니다. start로 시작한 패치만 중지합니다.

내장 패치

모듈 내의 어떤 내장(builtins)도 패치 할 수 있습니다. 다음 예제는 내장 *ord()*를 패치합니다:

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
>>> test()
101
```

TEST_PREFIX

모든 패처를 클래스 데코레이터로 사용할 수 있습니다. 이런 식으로 사용하면 클래스의 모든 테스트 메서드를 감쌉니다. 패처는 `'test'`로 시작하는 메서드를 테스트 메서드로 인식합니다. 이것은 *unittest.TestLoader*가 기본적으로 테스트 메서드를 찾는 것과 같은 방법입니다.

테스트에 다른 접두사를 사용하고 싶을 수도 있습니다. `patch.TEST_PREFIX`를 설정하여 다른 접두사를 패처에 알릴 수 있습니다:

```
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>>
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

패치 데코레이터 중첩하기

여러 패치를 수행하려면 단순히 데코레이터를 쌓으면 됩니다.

이 패턴을 사용하여 여러 패치 데코레이터를 쌓을 수 있습니다:

```
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.static_method('foo')
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

데코레이터는 아래에서 위쪽으로 적용됨에 유의하십시오. 이것은 파이썬이 데코레이터를 적용하는 표준 방식입니다. 생성된 모의 객체가 테스트 함수에 전달되는 순서는 이 순서와 일치합니다.

패치할 곳

*patch()*는 *name*이 가리키는 객체를 다른 객체로 (임시로) 변경하여 작동합니다. 개별 객체를 가리키는 많은 이름이 있을 수 있어서, 패치가 작동하려면 테스트 대상 시스템에서 사용하는 이름을 패치해야 합니다.

기본 원칙은 객체가 조회되는 곳을 패치하는 것입니다. 이것은 정의된 곳과 반드시 같은 곳일 필요는 없습니다. 몇 가지 예가 이를 명확히 하는 데 도움이 됩니다.

테스트하려는 프로젝트가 다음 구조로 되어 있다고 가정하십시오:

```
a.py
    -> SomeClass를 정의합니다

b.py
    -> from a import SomeClass
    -> some_function이 SomeClass의 인스턴스를 만듭니다
```

이제 `some_function`을 테스트하고 싶지만, *patch()*를 사용하여 `SomeClass`를 모킹하고 싶습니다. 문제는 모듈 b를 임포트 할 때, 모듈 a에서 `SomeClass`를 임포트해야 한다는 것입니다. *patch()*를 사용하여 `a.SomeClass`를 모킹하면 테스트에 영향을 미치지 않습니다; 모듈 b는 이미 실제 `SomeClass`에 대한 참조를 가지고 있으며 패치가 효과가 없는 것처럼 보입니다.

핵심은 `SomeClass`가 사용되는 곳(또는 그것을 조회하는 곳)을 패치하는 것입니다. 이 경우 `some_function`은 우리가 임포트 한 모듈 b에서 `SomeClass`를 조회합니다. 패치는 다음과 같아야 합니다:

```
@patch('b.SomeClass')
```

하지만, `from a import SomeClass` 대신 모듈 b가 `import a`를 수행하고 `some_function`이 `a.SomeClass`를 사용하는 대체 시나리오를 생각해보십시오. 이 두 가지 임포트 형식은 모두 일반적입니다. 이 경우 패치하려는 클래스가 모듈에서 조회되므로 대신 `a.SomeClass`를 패치해야 합니다:

```
@patch('a.SomeClass')
```

디스크립터와 프락시 객체 패치하기

patch 와 *patch.object* 모두 디스크립터를 올바르게 패치하고 복원합니다: 클래스 메서드, 정적 메서드 및 프로퍼티. 이것들을 인스턴스가 아닌 클래스에서 패치해야 합니다. 이들은 또한 장고 settings 객체와 같이 어트리뷰트 액세스를 프락시 하는 일부 객체에서도 작동합니다.

26.6.4 MagicMock과 매직 메서드 지원

매직 메서드 모킹하기

*Mock*은 "매직 메서드"라고도 하는 파이썬 프로토콜 메서드 모킹을 지원합니다. 이를 통해 모의 객체가 컨테이너나 파이썬 프로토콜을 구현하는 다른 객체를 대체할 수 있습니다.

매직 메서드는 일반 메서드와는 다르게 조회되므로[2], 이 지원은 특별히 구현되었습니다. 즉, 특정 매직 메서드만 지원됩니다. 지원되는 목록에 이들이 거의 모두 포함됩니다. 누락된 것이 있으면 알려주십시오.

관심 있는 메서드를 함수나 모의 객체 인스턴스로 설정하여 매직 메서드를 모킹합니다. 함수를 사용하면 반드시 `self`를 첫 번째 인자로 취해야 합니다[3].

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
```

(다음 페이지에 계속)

[2] 매직 메서드는 인스턴스가 아닌 클래스에서 조회되어야 합니다. 다른 버전의 파이썬은 이 규칙을 적용하는 데 일관적이지 않습니다. 지원되는 프로토콜 메서드는 지원되는 모든 버전의 파이썬에서 작동해야 합니다.

[3] 이 함수는 기본적으로 클래스에 연결되어 있지만, 각 `Mock` 인스턴스는 다른 것들과 격리되어 있습니다.

(이전 페이지에서 계속)

```
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]
```

이것의 한 가지 사용 사례는 `with` 문에서 컨텍스트 관리자로 사용되는 객체를 모킹하는 것입니다:

```
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...     assert m == 'foo'
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

매직 메서드 호출은 *method_calls*에는 나타나지 않지만, *mock_calls*에 기록됩니다.

> **참고**
>
> *spec* 키워드 인자를 사용하여 모의 객체를 만들면 spec에 없는 매직 메서드를 설정하려고 시도할 때 *AttributeError*가 발생합니다.

지원되는 매직 메서드의 전체 목록은 다음과 같습니다:

- `__hash__`, `__sizeof__`, `__repr__` 및 `__str__`
- `__dir__`, `__format__` 및 `__subclasses__`
- `__round__`, `__floor__`, `__trunc__` 및 `__ceil__`
- 비교: `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__` 및 `__ne__`
- 컨테이너 메서드: `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`, `__len__`, `__iter__`, `__reversed__` 및 `__missing__`
- 컨텍스트 관리자: `__enter__`, `__exit__`, `__aenter__` 및 `__aexit__`
- 단항 숫자 메서드: `__neg__`, `__pos__` 및 `__invert__`
- 숫자 메서드 (뒤집히거나 제자리 변형 포함): `__add__`, `__sub__`, `__mul__`, `__matmul__`, `__truediv__`, `__floordiv__`, `__mod__`, `__divmod__`, `__lshift__`, `__rshift__`, `__and__`, `__xor__`, `__or__` 및 `__pow__`
- 숫자 변환 메서드: `__complex__`, `__int__`, `__float__` 및 `__index__`
- 디스크립터 메서드: `__get__`, `__set__` 및 `__delete__`
- 피클링: `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` 및 `__setstate__`
- 파일 시스템 경로 표현: `__fspath__`

- 비동기 이터레이션 메서드: `__aiter__`와 `__anext__`

버전 3.8에서 변경: *os.PathLike.__fspath__()*에 대한 지원이 추가되었습니다.

버전 3.8에서 변경: `__aenter__`, `__aexit__`, `__aiter__` 및 `__anext__`에 대한 지원이 추가되었습니다.

다음과 같은 메서드가 존재하지만, 모의 객체가 사용 중이거나, 동적으로 설정할 수 없거나, 문제를 일으킬 수 있어서 지원되지 않습니다:

- `__getattr__`, `__setattr__`, `__init__` 및 `__new__`
- `__prepare__`, `__instancecheck__`, `__subclasscheck__`, `__del__`

매직 모의 객체

두 가지 `MagicMock` 변형이 있습니다: *MagicMock*과 *NonCallableMagicMock*.

class unittest.mock.**MagicMock**(**args*, ***kw*)

`MagicMock`은 대부분의 매직 메서드의 기본 구현이 있는 *Mock*의 서브 클래스입니다. 매직 메서드를 직접 구성하지 않고도 `MagicMock`을 사용할 수 있습니다.

생성자 매개 변수는 *Mock*과 같은 의미입니다.

*spec*이나 *spec_set* 인자를 사용하면 오직 사양에 존재하는 매직 메서드만 만들어집니다.

class unittest.mock.**NonCallableMagicMock**(**args*, ***kw*)

콜러블이 아닌 *MagicMock* 버전.

생성자 매개 변수는 콜러블이 아닌 모의 객체에 의미가 없는 *return_value*와 *side_effect*를 제외하고 *MagicMock*과 같은 의미입니다.

매직 메서드는 *MagicMock* 객체로 설정되므로, 일반적인 방법으로 구성하고 사용할 수 있습니다:

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

기본적으로 많은 프로토콜 메서드는 특정 형의 객체를 반환하도록 요구합니다. 이러한 메서드는 기본 반환 값으로 사전 구성되어 있어서, 반환 값에 관심이 없을 때 아무 조치를 하지 않아도 사용할 수 있습니다. 기본값을 변경하고 싶으면, 여전히 반환 값을 수동으로 설정할 수 있습니다.

메서드와 기본값:

- `__lt__`: *NotImplemented*
- `__gt__`: `NotImplemented`
- `__le__`: `NotImplemented`
- `__ge__`: `NotImplemented`
- `__int__`: `1`
- `__contains__`: `False`
- `__len__`: `0`
- `__iter__`: `iter([])`
- `__exit__`: `False`
- `__aexit__`: `False`
- `__complex__`: `1j`
- `__float__`: `1.0`
- `__bool__`: `True`

- `__index__`: 1
- `__hash__`: 모의 객체의 기본 hash
- `__str__`: 모의 객체의 기본 str
- `__sizeof__`: 모의 객체의 기본 sizeof

예를 들면:

```
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False
```

두 동등 비교 메서드 `__eq__()` 와 `__ne__()` 는 특별합니다. 이들의 반환 값을 변경하여 다른 것을 반환하지 않는 한, *side_effect* 어트리뷰트를 사용하여 아이덴티티에 대한 기본 동등 비교를 수행합니다:

```
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
>>> mock == 3
True
```

`MagicMock.__iter__()` 의 반환 값은 임의의 이터러블 객체일 수 있으며 이터레이터일 필요는 없습니다:

```
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

반환 값이 이터레이터면, 일단 이터레이트 하면 이를 소진하고 후속 이터레이션은 빈 목록을 줍니다:

```
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

`MagicMock` 에는 불분명하고 쓸모없는 일부를 제외하고 지원되는 모든 매직 메서드가 구성되어 있습니다. 원한다면 여전히 설정할 수 있습니다.

`MagicMock` 에서 지원되지만, 기본적으로 설정되지 않는 매직 메서드는 다음과 같습니다:

- `__subclasses__`
- `__dir__`
- `__format__`
- `__get__`, `__set__` 및 `__delete__`
- `__reversed__` 와 `__missing__`

- `__reduce__`, `__reduce_ex__`, `__getinitargs__`, `__getnewargs__`, `__getstate__` 및 `__setstate__`
- `__getformat__`

26.6.5 도우미

sentinel

`unittest.mock.`**`sentinel`**

`sentinel` 객체는 테스트에 고유한(unique) 객체를 제공하는 편리한 방법을 제공합니다.

어트리뷰트는 이름으로 어트리뷰트에 액세스할 때 요청에 따라 만들어집니다. 같은 어트리뷰트에 액세스하면 항상 같은 객체가 반환됩니다. 반환된 객체는 테스트 실패 메시지를 읽을 수 있도록 적절한 repr을 갖습니다.

버전 3.7에서 변경: `sentinel` 어트리뷰트는 이제 복사되거나 피클될 때 아이덴티티를 유지합니다.

때로 테스트할 때 특정 객체가 다른 메서드에 대한 인자로 전달되거나 반환되는지 테스트해야 할 때가 있습니다. 이것을 테스트하기 위해 이름 붙은 sentinel 객체를 만드는 것이 일반적일 수 있습니다. *sentinel*은 이와 같은 객체의 아이덴티티를 만들고 테스트하는 편리한 방법을 제공합니다.

이 예에서는 `method`를 `sentinel.some_object`를 반환하도록 몽키 패치합니다:

```
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> result
sentinel.some_object
```

DEFAULT

`unittest.mock.`**`DEFAULT`**

DEFAULT 객체는 미리 만들어진 sentinel(실제로 `sentinel.DEFAULT`)입니다. *side_effect* 함수에서 일반 반환 값을 사용해야 함을 나타내는 데 사용할 수 있습니다.

call

`unittest.mock.`**`call`**(**args, **kwargs*)

*call()*은 *call_args*, *call_args_list*, *mock_calls* 및 *method_calls*와 비교할 때, 더 간단한 어서션을 만들기 위한 도우미 객체입니다. *call()*은 *assert_has_calls()*와 함께 사용할 수도 있습니다.

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'), call()]
True
```

`call.`**`call_list`**()

여러 호출을 나타내는 호출 객체의 경우, *call_list()*는 마지막 호출뿐만 아니라 모든 중간 호출의 목록을 반환합니다.

`call_list`는 "연쇄 호출(chained calls)"에 대한 어서션을 만드는 데 특히 유용합니다. 연쇄 호출은 한 줄의 코드에 있는 여러 번의 호출입니다. 이로 인해 모의 객체의 *mock_calls*에 여러 항목이 생성됩니다. 호출 시퀀스를 수동으로 구성하는 것은 지루할 수 있습니다.

*call_list()*는 같은 연쇄 호출에서 호출 시퀀스를 구성 할 수 있습니다.:

```
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MagicMock name='mock().method().other()()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
[call(1),
 call().method(arg='foo'),
 call().method().other('bar'),
 call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

call 객체는 구성 방식에 따라 (위치 인자, 키워드 인자)나 (이름, 위치 인자, 키워드 인자)의 튜플입니다. 이것들을 직접 만들 때 특별히 흥미롭지는 않지만, *Mock.call_args*, *Mock.call_args_list* 및 *Mock.mock_calls* 어트리뷰트에 있는 call 객체는 포함된 개별 인자를 얻기 위해 검사될 수 있습니다.

*Mock.call_args*와 *Mock.call_args_list*에 있는 call 객체는 (위치 인자, 키워드 인자)의 2-튜플이지만, *Mock.mock_calls*에 있는 call 객체는, 여러분이 직접 생성하는 객체와 함께, (이름, 위치 인자, 키워드 인자)의 3-튜플입니다.

이들의 "튜플성(tupleness)"을 사용하여 더 복잡한 검사와 어서션을 위해 개별 인자를 가져올 수 있습니다. 위치 인자는 튜플(위치 인자가 없으면 비어있는 튜플)이고 키워드 인자는 딕셔너리입니다:

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
>>> kall.args
(1, 2, 3)
>>> kall.kwargs
{'arg': 'one', 'arg2': 'two'}
>>> kall.args is kall[0]
True
>>> kall.kwargs is kall[1]
True
```

```
>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg': 'two', 'arg2': 'three'}
>>> name is m.mock_calls[0][0]
True
```

create_autospec

unittest.mock.**create_autospec**(*spec, spec_set=False, instance=False, **kwargs*)

다른 객체를 사양으로 사용하여 모의 객체를 만듭니다. 모의 객체 어트리뷰트는 *spec* 객체의 해당 어트리뷰트를 사양으로 사용합니다.

모킹되는 함수나 메서드는 올바른 서명으로 호출되도록 인자를 점검합니다.

spec_set 이 True 이면 사양 객체에 존재하지 않는 어트리뷰트를 설정하려는 시도는 *AttributeError*를 발생시킵니다.

클래스가 사양으로 사용되면 모의 객체의 반환 값(클래스의 인스턴스)은 같은 사양을 갖습니다. `instance=True`를 전달하여 클래스를 인스턴스 객체의 사양으로 사용할 수 있습니다. 반환된 모의 객체는 모의 객체의 인스턴스가 콜러블일 때만 콜러블입니다.

*create_autospec()*은 또한 만들어진 모의 객체의 생성자에 전달되는 임의의 키워드 인자를 취합니다.

*create_autospec()*과 *patch()*에 대한 *autospec* 인자로 자동 사양을 사용하는 방법에 대한 예는 자동 사양을 참조하십시오.

버전 3.8에서 변경: 대상이 비동기 함수이면 *create_autospec()*은 이제 *AsyncMock*을 반환합니다.

ANY

unittest.mock.**ANY**

때로는 모의 객체 호출에서 인자의 일부에 대한 어서션을 만들 필요가 있지만, 일부 인자에는 신경 쓰지 않거나 *call_args*에서 개별적으로 꺼내어 더 복잡한 어서션을 만들고 싶을 수도 있습니다.

특정 인자를 무시하려면 모든 것과 같다고 비교되는 객체를 전달할 수 있습니다. *assert_called_with()*와 *assert_called_once_with()*에 대한 호출은 전달된 내용과 관계없이 성공합니다.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

*ANY*는 *mock_calls*와 같은 호출 리스트와 비교할 때도 사용할 수 있습니다.:

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

ANY is not limited to comparisons with call objects and so can also be used in test assertions:

```
class TestStringMethods(unittest.TestCase):

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', ANY])
```

FILTER_DIR

unittest.mock.**FILTER_DIR**

*FILTER_DIR*은 모의 객체가 *dir()*에 응답하는 방식을 제어하는 모듈 수준 변수입니다. 기본값은 `True`이며, 아래 설명된 필터링을 사용하여 유용한 멤버만 표시합니다. 이 필터링이 마음에 들지 않거나 진단 목적으로 필터를 꺼야 하면, `mock.FILTER_DIR = False`를 설정하십시오.

필터링을 켜면, `dir(some_mock)`는 유용한 어트리뷰트만 표시하고 일반적으로 표시되지 않는 동적으로 만들어진 어트리뷰트를 포함합니다. *spec*(또는 물론 *autospec*)으로 모의 객체를 만들었으면, 아직 액세스하지 않았다 할지라도 원본의 모든 어트리뷰트가 표시됩니다:

```
>>> dir(Mock())
['assert_any_call',
 'assert_called',
 'assert_called_once',
 'assert_called_once_with',
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
 'assert_called_with',
 'assert_has_calls',
 'assert_not_called',
 'attach_mock',
 ...
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...
```

*Mock*에 대해 *dir()*을 호출한 결과에서 그다지 유용하지 않은 (모킹되는 대상이 아닌 *Mock* 전용) 밑줄과 이중 밑줄 접두어 어트리뷰트가 필터링 되었습니다. 이 동작이 마음에 들지 않으면 모듈 수준 스위치 *FILTER_DIR*를 설정하여 끌 수 있습니다.:

```
>>> from unittest import mock
>>> mock.FILTER_DIR = False
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
 '__class__',
 ...
```

또는 `vars(my_mock)`(인스턴스 멤버)과 `dir(type(my_mock))`(형 멤버)을 사용하여 *FILTER_DIR*에 관계없이 필터링을 무시할 수 있습니다.

mock_open

`unittest.mock.`**`mock_open`**(*mock=None*, *read_data=None*)

*open()*의 사용을 대체하기 위한 모의 객체를 만드는 도우미 함수. 직접 호출되거나 컨텍스트 관리자로 사용되는 *open()*에 대해 작동합니다.

mock 인자는 구성할 모의 객체입니다. `None`(기본값)이면 표준 파일 핸들에서 사용 가능한 메서드나 어트리뷰트로 API가 제한된 *MagicMock*이 만들어집니다.

*read_data*는 파일 핸들의 *read()*, *readline()* 및 *readlines()* 메서드가 반환할 문자열입니다. 이러한 메서드를 호출하면 *read_data*에서 데이터가 고갈될 때까지 데이터를 가져옵니다. 이러한 메서드의 모의 객체는 매우 단순합니다: *mock*이 호출될 때마다 *read_data*는 처음으로 되감깁니다. 테스트되는 코드에 공급하는 데이터를 더 세밀하게 제어하려면 이 모의 객체를 스스로 사용자 정의해야 합니다. 이것으로 불충분하면, PyPI의 메모리 내 파일 시스템 패키지 중 하나가 테스트를 위한 진짜 같은 파일 시스템을 제공할 수 있습니다.

버전 3.4에서 변경: *readline()*과 *readlines()* 지원이 추가되었습니다. *read()*의 모의 객체는 호출마다 *read_data*를 반환하는 대신 소비하도록 변경되었습니다.

버전 3.5에서 변경: *read_data*는 이제 *mock*을 호출할 때마다 재설정됩니다.

버전 3.8에서 변경: 이터레이션(가령 for 루프)에서 *read_data*를 올바르게 소비하도록 구현에 *__iter__()*를 추가했습니다.

*open()*을 컨텍스트 관리자로 사용하는 것은 파일 핸들이 올바르게 닫히도록 보장하는 좋은 방법이고 점차 일반화되고 있습니다:

```
with open('/some/path', 'w') as f:
    f.write('something')
```

문제는 *open()*에 대한 호출을 모킹하더라도 컨텍스트 관리자로 사용되는 것은 반환된 객체라는 것입니다 (그리고 `__enter__()`와 `__exit__()`가 호출됩니다).

*MagicMock*으로 컨텍스트 관리자를 모킹하는 것은 도우미 함수가 유용할 만큼 충분히 일반적입니다.

```
>>> m = mock_open()
>>> with patch('__main__.open', m):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...
>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]
>>> m.assert_called_once_with('foo', 'w')
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

그리고 파일을 읽기 위해서는:

```
>>> with patch('__main__.open', mock_open(read_data='bibble')) as m:
...     with open('foo') as h:
...         result = h.read()
...
>>> m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

자동 사양

자동 사양은 기존의 `spec` 기능을 기반으로 합니다. 모의 객체 api를 원본 객체(사양)의 api로 제한하지만, 재귀적(게으르게(lazily) 구현되었습니다)이라서 모의 객체의 어트리뷰트는 사양의 어트리뷰트와 같은 api만 갖습니다. 또한 모킹된 함수 / 메서드는 원본과 같은 호출 서명을 가지므로 잘못 호출되면 *TypeError*를 발생시킵니다.

자동 사양이 작동하는 방식을 설명하기 전에, 이것이 필요한 이유는 이렇습니다.

*Mock*은 매우 강력하고 유연한 객체이지만, 모킹에 일반적으로 존재하는 결함을 지니고 있습니다. 멤버 이름 등을 변경하는 등 일부 코드를 리팩토링하면, 여전히 예전 *api*를 사용하는 코드에 대한 테스트는 실제 객체 대신 모의 객체를 사용하면 모두 통과합니다. 이것은 코드가 손상되어도 테스트를 모두 통과 할 수 있다는 뜻입니다.

버전 3.5에서 변경: Before 3.5, tests with a typo in the word assert would silently pass when they should raise an error. You can still achieve this behavior by passing `unsafe=True` to Mock.

이것이 단위 테스트뿐만 아니라 통합 테스트가 필요한 또 다른 이유입니다. 모든 것을 분리해서 테스트하는 것은 괜찮고 멋지지만, 단위들이 어떻게 "서로 연결되어" 있는지 테스트하지 않으면 테스트가 발견할 수도 있을 버그에 대한 여지가 여전히 많습니다.

*unittest.mock*은 이미 이에 도움이 되는 기능인 사양(speccing)을 제공합니다. 모의 객체에 클래스나 인스턴스를 `spec`으로 사용하면, 실제 클래스에 존재하는 모의 객체 어트리뷰트에만 액세스 할 수 있습니다:

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with  # Intentional typo!
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute 'assret_called_with'
```

사양은 모의 객체 자체에만 적용되므로, 모의 객체의 메서드와 관계된 문제는 여전히 있습니다:

```
>>> mock.header_items()
<mock.Mock object at 0x...>
>>> mock.header_items.assret_called_with()  # Intentional typo!
```

자동 사양은 이 문제를 해결합니다. `autospec=True`를 *patch()* / *patch.object()*로 전달하거나 *create_autospec()* 함수를 사용하여 사양이 있는 모의 객체를 만들 수 있습니다. `autospec=True` 인자를 *patch()*에 사용하면 대체 중인 객체가 사양 객체로 사용됩니다. 사양화(speccing)는 "게으르게(lazily)" 수행되므로 (모의 객체의 어트리뷰트에 액세스할 때 사양이 만들어집니다) 성능이 크게 저하되지 않고도 매우 복잡하거나 깊이 중첩된 객체(가령 모듈을 임포트 하는 모듈을 임포트 하는 모듈)와 함께 사용할 수 있습니다.

사용 예는 다음과 같습니다:

```
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='...'>
```

`request.Request`에 spec이 있음을 알 수 있습니다. `request.Request`는 생성자에서 두 개의 인자를 취합니다 (하나는 *self*입니다). 잘못 호출하면 이렇게 됩니다:

```
>>> req = request.Request()
Traceback (most recent call last):
 ...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

사양은 인스턴스 화 된 클래스(즉, 사양화된 모의 객체의 반환 값)에도 적용됩니다:

```
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request' id='...'>
```

`Request` 객체는 콜러블이 아니므로, 모킹된 `request.Request`의 인스턴스 화의 반환 값은 콜러블이 아닌 모의 객체입니다. 사양이 적용되면 어서션에 오타가 있을 때 올바른 에러를 발생시킵니다:

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with  # Intentional typo!
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute 'assret_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

많은 경우 단지 기존 *patch()* 호출에 `autospec=True`를 추가하면 오타와 api 변경으로 인한 버그로부터 보호할 수 있습니다.

*patch()*를 통해 *autospec*을 사용하는 것뿐만 아니라 자동 사양 모의 객체를 직접 만들기 위한 *create_autospec()*도 있습니다:

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='...'>
```

그러나 경고와 제한이 없는 것은 아니기 때문에 기본 동작이 아닙니다. 사양 객체에서 사용 가능한 어트리뷰트를 알기 위해 autospec은 사양을 검사(어트리뷰트를 액세스합니다)해야 합니다. 모의 객체 어트리뷰

트를 탐색할 때 원본 객체의 대응하는 탐색이 수면 아래에서 발생합니다. 사양 객체 중 어느 하나가 코드 실행을 트리거 할 수 있는 프로퍼티나 디스크립터를 갖는 경우 자동 사양을 사용하지 못할 수 있습니다. 반면에, 내부 검사가 안전하도록 객체를 설계하는 것이 훨씬 좋습니다[4].

더 심각한 문제는 인스턴스 어트리뷰트가 `__init__()` 메서드에서 만들어지고 클래스에 전혀 존재하지 않는 것이 일반적이라는 것입니다. *autospec*은 동적으로 만들어진 어트리뷰트를 알 수 없으며 api를 가시적 어트리뷰트로 제한합니다.

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

이 문제를 해결하는 방법에는 몇 가지가 있습니다. 가장 쉬운, 하지만 가장 덜 성가시다고 할 수는 없는, 방법은 단순히 생성 후에 모의 객체에 필요한 어트리뷰트를 설정하는 것입니다. *autospec*은 사양에 존재하지 않는 어트리뷰트를 꺼내는 것을 허락하지 않기 때문에 그것을 설정하는 것을 막지는 않습니다:

```
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a = 33
...
```

존재하지 않는 어트리뷰트를 설정하지 못하게 하는 *spec*과 *autospec*의 더 적극적인 버전이 있습니다. 이것은 코드가 오직 유효한 어트리뷰트만 설정하도록 보장하려는 경우 유용하지만, 명백히 이 특정 시나리오를 막습니다:

```
>>> with patch('__main__.Something', autospec=True, spec_set=True):
...   thing = Something()
...   thing.a = 33
...
Traceback (most recent call last):
 ...
AttributeError: Mock object has no attribute 'a'
```

아마도 문제를 해결하는 가장 좋은 방법은 `__init__()`에서 초기화되는 인스턴스 멤버의 기본값으로 클래스 어트리뷰트를 추가하는 것입니다. `__init__()`에서 기본 어트리뷰트만 설정하고 있다면 클래스 어트리뷰트(물론 인스턴스 간에 공유됩니다)를 통해 제공하는 것이 더 빠르기도 합니다. 예를 들어

```
class Something:
    a = 33
```

이것은 또 다른 문제를 일으킵니다. 나중에 다른 형의 객체가 될 멤버에 대해 기본값 `None`을 제공하는 것이 일반적입니다. 아무런 어트리뷰트나 메서드에도 액세스할 수 없도록 하므로 `None`은 사양으로 쓸모가 없습니다. `None`은 결코 스펙으로 유용하지 않고, 일반적으로 다른 형의 멤버를 나타낼 것이기 때문에, autospec은 `None`으로 설정된 멤버의 사양을 사용하지 않습니다. 이것들은 평범한 모의 객체일 것입니다 (아마도 - MagicMock):

```
>>> class Something:
...     member = None
...
```

(다음 페이지에 계속)

[4] 이것은 클래스나 이미 인스턴스 화 된 객체에만 적용됩니다. 모킹된 클래스를 호출하여 모의 객체 인스턴스를 만드는 것은 실제 인스턴스를 만들지 않습니다. (*dir()* 호출과 함께) 어트리뷰트 조회만 수행됩니다.

(이전 페이지에서 계속)

```
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

프로덕션 클래스를 수정하여 기본값을 추가하는 것이 마음에 들지 않으면 더 많은 선택지가 있습니다. 이 중 하나는 단순히 클래스가 아닌 인스턴스를 사양으로 사용하는 것입니다. 다른 하나는 프로덕션 클래스의 서브 클래스를 만들고 프로덕션 클래스에 영향을 주지 않으면서 기본값을 서브 클래스에 추가하는 것입니다. 이 두 가지 모두 사양으로 대체 객체를 사용해야 합니다. 고맙게도 `patch()`는 이것을 지원합니다 - 단순히 대체 객체를 *autospec* 인자로 전달할 수 있습니다:

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> class SomethingForTest(Something):
...   a = 33
...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>
```

실링 모의 객체 봉인하기

`unittest.mock.`**`seal`**(*mock*)

Seal은 봉인되는 모의 객체나 이것의 재귀적으로 이미 모의 객체인 어트리뷰트의 어트리뷰트를 액세스할 때 자동 모의 객체 생성을 비활성화합니다.

이름이나 사양을 가진 모의 객체 인스턴스가 어트리뷰트에 대입되면 봉인 체인에서 고려되지 않습니다. 이것은 seal이 모의 객체의 일부를 고정하는 것을 방지할 수 있게 합니다.

```
>>> mock = Mock()
>>> mock.submock.attribute1 = 2
>>> mock.not_submock = mock.Mock(name="sample_name")
>>> seal(mock)
>>> mock.new_attribute  # AttributeError를 발생시킵니다.
>>> mock.submock.attribute2  # AttributeError를 발생시킵니다.
>>> mock.not_submock.attribute2  # 예외를 발생시키지 않습니다.
```

Added in version 3.7.

26.6.6 Order of precedence of `side_effect`, `return_value` and *wraps*

The order of their precedence is:

1. `side_effect`
2. `return_value`
3. *wraps*

If all three are set, mock will return the value from `side_effect`, ignoring `return_value` and the wrapped object altogether. If any two are set, the one with the higher precedence will return the value. Regardless of the order of which was set first, the order of precedence remains unchanged.

```
>>> from unittest.mock import Mock
>>> class Order:
...     @staticmethod
...     def get_value():
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...         return "third"
...
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.get_value.side_effect = ["first"]
>>> order_mock.get_value.return_value = "second"
>>> order_mock.get_value()
'first'
```

As `None` is the default value of *side_effect*, if you reassign its value back to `None`, the order of precedence will be checked between *return_value* and the wrapped object, ignoring *side_effect*.

```
>>> order_mock.get_value.side_effect = None
>>> order_mock.get_value()
'second'
```

If the value being returned by *side_effect* is *DEFAULT*, it is ignored and the order of precedence moves to the successor to obtain the value to return.

```
>>> from unittest.mock import DEFAULT
>>> order_mock.get_value.side_effect = [DEFAULT]
>>> order_mock.get_value()
'second'
```

When *Mock* wraps an object, the default value of *return_value* will be *DEFAULT*.

```
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.return_value
sentinel.DEFAULT
>>> order_mock.get_value.return_value
sentinel.DEFAULT
```

The order of precedence will ignore this value and it will move to the last successor which is the wrapped object.

As the real call is being made to the wrapped object, creating an instance of this mock will return the real instance of the class. The positional arguments, if any, required by the wrapped object must be passed.

```
>>> order_mock_instance = order_mock()
>>> isinstance(order_mock_instance, Order)
True
>>> order_mock_instance.get_value()
'third'
```

```
>>> order_mock.get_value.return_value = DEFAULT
>>> order_mock.get_value()
'third'
```

```
>>> order_mock.get_value.return_value = "second"
>>> order_mock.get_value()
'second'
```

But if you assign `None` to it, this will not be ignored as it is an explicit assignment. So, the order of precedence will not move to the wrapped object.

```
>>> order_mock.get_value.return_value = None
>>> order_mock.get_value() is None
True
```

Even if you set all three at once when initializing the mock, the order of precedence remains the same:

```
>>> order_mock = Mock(spec=Order, wraps=Order,
...                   **{"get_value.side_effect": ["first"],
...                      "get_value.return_value": "second"}
...                   )
...
>>> order_mock.get_value()
'first'
>>> order_mock.get_value.side_effect = None
>>> order_mock.get_value()
'second'
>>> order_mock.get_value.return_value = DEFAULT
>>> order_mock.get_value()
'third'
```

If *side_effect* is exhausted, the order of precedence will not cause a value to be obtained from the successors. Instead, `StopIteration` exception is raised.

```
>>> order_mock = Mock(spec=Order, wraps=Order)
>>> order_mock.get_value.side_effect = ["first side effect value",
...                                     "another side effect value"]
>>> order_mock.get_value.return_value = "second"
```

```
>>> order_mock.get_value()
'first side effect value'
>>> order_mock.get_value()
'another side effect value'
```

```
>>> order_mock.get_value()
Traceback (most recent call last):
 ...
StopIteration
```

26.7 unittest.mock — 시작하기

Added in version 3.3.

26.7.1 모의 객체 사용하기

메서드를 패치하는 모의 객체

Mock 객체의 일반적인 용도는 다음과 같습니다:

- 메서드 패치하기
- 객체에 대한 메서드 호출 기록하기

객체의 메서드를 대체하여 시스템의 다른 부분에서 올바른 인자로 호출되었는지 확인할 수 있습니다:

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

일단 모의 객체가 사용되면 (이 예제에서는 `real.method`) 사용 방법에 대한 어서션을 만들 수 있도록 하는 메서드와 어트리뷰트를 제공합니다.

> **참고**
>
> 이 예의 대부분에서 *Mock*과 *MagicMock* 클래스는 교환할 수 있습니다. MagicMock이 더 유능한 클래스이기 때문에 기본 사용하기에 적합합니다.

일단 모의 객체가 호출되면 그것의 *called* 어트리뷰트가 True로 설정됩니다. 더 중요하게 *assert_called_with()*나 *assert_called_once_with()* 메서드를 사용하여 올바른 인자로 호출되었는지 확인할 수 있습니다.

이 예제는 ProductionClass().method를 호출하면 something 메서드가 호출되는지 테스트합니다:

```
>>> class ProductionClass:
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

객체의 메서드 호출을 위한 모의 객체

마지막 예제에서 우리는 객체에 메서드를 직접 패치하여 올바르게 호출되었는지 확인했습니다. 또 다른 일반적인 사용 사례는 객체를 메서드(또는 테스트 중인 시스템의 일부)에 전달한 다음 올바른 방식으로 사용되는지 확인하는 것입니다.

아래의 간단한 ProductionClass에는 closer 메서드가 있습니다. 객체로 호출되면 그것의 close를 호출합니다.

```
>>> class ProductionClass:
...     def closer(self, something):
...         something.close()
...
```

따라서 테스트하려면 close 메서드를 가진 객체를 전달하고 올바르게 호출되었는지 확인해야 합니다.

```
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

모의 객체에 'close' 메서드를 제공하기 위해 어떤 작업도 수행할 필요가 없습니다. close에 액세스하면 만들어집니다. 따라서, 'close' 가 아직 호출되지 않았다면 테스트에서 액세스할 때 만들어지지만, *assert_called_with()*는 실패 예외를 발생시킵니다.

클래스 모킹하기

일반적인 사용 사례는 테스트 중인 코드가 인스턴스 화하는 클래스를 모킹하는 것입니다. 클래스를 패치하면, 해당 클래스가 모의 객체로 바뀝니다. 인스턴스는 클래스를 호출해서 만들어집니다. 이는 모킹된 클래스의 반환 값을 확인하여 "모의 인스턴스"에 액세스한다는 것을 뜻합니다.

아래 예제에는 Foo를 인스턴스 화하고 그것의 메서드를 호출하는 some_function 함수가 있습니다. *patch()*에 대한 호출은 클래스 Foo를 모의 객체로 대체합니다. Foo 인스턴스는 모의 객체를 호출한 결과라서, 모의 객체 *return_value*를 수정하여 구성됩니다.

```
>>> def some_function():
...     instance = module.Foo()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     return instance.method()
...
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
...     result = some_function()
...     assert result == 'the result'
```

모의 객체 이름 붙이기

모의 객체에 이름을 지정하는 것이 유용할 수 있습니다. 이름은 모의 객체의 repr에 표시되며 모의 객체가 테스트 실패 메시지에 나타날 때 유용할 수 있습니다. 이 이름은 모의 객체의 어트리뷰트나 메서드에도 전파됩니다:

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

모든 호출 추적하기

메서드에 대한 단일 호출 이상을 추적하려는 경우가 종종 있습니다. *mock_calls* 어트리뷰트는 모의 객체의 자식 어트리뷰트에 대한 모든 호출을 기록합니다 - 그리고 그들의 자식에 대해서도 마찬가지입니다.

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

`mock_calls`에 대한 어서션을 만들고 예기치 않은 메서드가 호출되면, 어서션이 실패합니다. 이 기능은 예상한 호출이 이루어졌음을 어서트 할 뿐만 아니라, 추가 호출 없이 올바른 순서로 호출되었는지 확인하기 때문에 유용합니다:

call 객체를 사용하여 `mock_calls`와 비교할 리스트를 구성합니다:

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

그러나, 모의 객체를 반환하는 호출에 대한 매개 변수는 기록되지 않아서, 조상을 만드는 데 사용되는 매개 변수가 중요한 중첩된 호출을 추적할 수 없습니다:

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False).deliver()
True
```

반환 값과 어트리뷰트 설정하기

모의 객체에서 반환 값을 설정하는 것은 아주 간단합니다:

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

물론 모의 객체의 메서드에 대해서도 마찬가지입니다:

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

생성자에서 반환 값을 설정할 수도 있습니다:

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

모의 객체에 어트리뷰트 설정이 필요하면, 그냥 하면 됩니다:

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

때로는 예를 들어 `mock.connection.cursor().execute("SELECT 1")` 와 같은 더 복잡한 상황을 모킹하고 싶을 수도 있습니다. 이 호출이 리스트를 반환하도록 하려면, 중첩 호출의 결과를 구성해야 합니다.

*call*을 사용하여 다음과 같이 "연쇄 호출(chained call)" 로 일련의 호출 집합을 구성할 수 있습니다:

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT 1").call_list()
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True
```

`.call_list()` 호출이 호출 객체를 연쇄 호출을 나타내는 호출 리스트로 변환합니다.

모의 객체로 예외 발생시키기

유용한 어트리뷰트는 *side_effect*입니다. 이것을 예외 클래스나 인스턴스로 설정하면 모의 객체가 호출될 때 예외가 발생합니다.

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

부작용 함수와 이터러블

side_effect는 함수나 이터러블로 설정할 수도 있습니다. side_effect의 이터러블로서의 사용 사례는 모의 객체가 여러 번 호출되고, 각 호출이 다른 값을 반환하기를 원하는 곳입니다. side_effect를 이터러블로 설정하면 모의 객체에 대한 모든 호출은 이터러블에서 다음 값을 반환합니다:

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

모의 객체 호출에 전달되는 것에 따라 반환 값을 동적으로 변경하는 것과 같은 고급 사용 사례의 경우, side_effect는 함수가 될 수 있습니다. 함수는 모의 객 체와 같은 인자로 호출됩니다. 함수가 반환하는 것을 호출이 반환합니다:

```
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
...
>>> mock = MagicMock(side_effect=side_effect)
>>> mock(1, 2)
1
>>> mock(2, 3)
2
```

비동기 이터레이터 모킹하기

파이썬 3.8부터, AsyncMock과 MagicMock은 __aiter__를 통해 async-iterators를 모킹하는 것을 지원합니다. __aiter__의 *return_value* 어트리뷰트를 사용하여 이터레이션에 사용될 반환 값을 설정할 수 있습니다.

```
>>> mock = MagicMock()  # AsyncMock also works here
>>> mock.__aiter__.return_value = [1, 2, 3]
>>> async def main():
...     return [i async for i in mock]
...
>>> asyncio.run(main())
[1, 2, 3]
```

비동기 컨텍스트 관리자 모킹하기

파이썬 3.8부터, AsyncMock과 MagicMock은 __aenter__와 __aexit__를 통해 async-context-managers를 모킹하는 것을 지원합니다. 기본적으로, __aenter__와 __aexit__는 비동기 함수를 반환하는 AsyncMock 인스턴스입니다.

```
>>> class AsyncContextManager:
...     async def __aenter__(self):
...         return self
...     async def __aexit__(self, exc_type, exc, tb):
...         pass
...
>>> mock_instance = MagicMock(AsyncContextManager())  # AsyncMock also works here
>>> async def main():
...     async with mock_instance as result:
...         pass
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
>>> asyncio.run(main())
>>> mock_instance.__aenter__.assert_awaited_once()
>>> mock_instance.__aexit__.assert_awaited_once()
```

기존 객체에서 모의 객체 만들기

모킹을 과도하게 사용하는 한 가지 문제는 테스트를 실제 코드가 아닌 모킹의 구현에 연결한다는 것입니다. `some_method`를 구현하는 클래스가 있다고 가정해봅시다. 다른 클래스에 대한 테스트에서, `some_method`*도* 제공하는 이 객체의 모의 객체를 제공합니다. 나중에 첫 번째 클래스를 리팩토링하여, 더는 `some_method`를 갖지 않습니다 - 그러면 이제 코드가 망가졌음에도 테스트는 계속 통과합니다!

`Mock`은 *spec* 키워드 인자를 사용하여, 모의 객체를 위한 사양으로 객체를 제공할 수 있도록 합니다. 사양 객체에 존재하지 않는 모의 객체의 메서드/어트리뷰트에 액세스하면 어트리뷰트 에러가 즉시 발생합니다. 사양의 구현을 변경하면, 해당 클래스를 사용하는 테스트는 해당 테스트에서 클래스를 인스턴스 화하지 않고도 즉시 실패하기 시작합니다.

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
   ...
AttributeError: Mock object has no attribute 'old_method'. Did you mean: 'class_
↪method'?
```

또한 사양을 사용하면 일부 매개 변수가 위치 인자나 이름 붙인 인자 중 어느 것으로 전달되는지와 관계없이 모의 객체에 대한 호출을 더 스마트하게 일치시킬 수 있도록 합니다:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, 3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(a=1, b=2, c=3)
```

이 더 스마트한 인치가 모의 객체에 대한 메서드 호출에서도 작동하게 하려면, 자동 사양을 사용할 수 있습니다.

임의의 어트리뷰트를 읽는 것뿐만 아니라 설정하지 못하게 하는 더 강력한 사양 형식을 원하면 *spec* 대신 *spec_set*을 사용할 수 있습니다.

Using side_effect to return per file content

`mock_open()` is used to patch `open()` method. `side_effect` can be used to return a new Mock object per call. This can be used to return different contents per file stored in a dictionary:

```
DEFAULT = "default"
data_dict = {"file1": "data1",
             "file2": "data2"}

def open_side_effect(name):
    return mock_open(read_data=data_dict.get(name, DEFAULT))()

with patch("builtins.open", side_effect=open_side_effect):
    with open("file1") as file1:
        assert file1.read() == "data1"

    with open("file2") as file2:
        assert file2.read() == "data2"
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    with open("file3") as file2:
        assert file2.read() == "default"
```

26.7.2 패치 데코레이터

> **참고**
>
> *patch()*를 사용할 때는 그것이 조회되는 이름 공간에서 객체를 패치하는 것이 중요합니다. 이것은 일반적으로 간단하지만, 빠른 안내를 위해 패치할 곳을 읽으십시오.

테스트에서 흔히 필요한 것은 클래스 어트리뷰트나 모듈 어트리뷰트를 패치하는 것입니다, 예를 들어 내장(builtin)을 패치하거나 모듈에 있는 인스턴스 화 되는 클래스를 패치하는 것. 모듈과 클래스는 사실상 전역이라서, 테스트 후에 패치를 실행 취소해야 합니다, 그렇지 않으면 패치가 다른 테스트로 지속하여, 진단하기 어려운 문제를 일으킵니다.

mock은 세 가지 편리한 데코레이터를 제공합니다: *patch()*, *patch.object()* 및 *patch.dict()*. patch는 패치 할 어트리뷰트를 지정하기 위해 package.module.Class.attribute 형식의 단일 문자열을 취합니다. 또한 선택적으로 어트리뷰트(또는 클래스나 무엇이건)를 바꾸려는 값을 취합니다. 'patch.object' 는 객체와 패치하려는 어트리뷰트의 이름 및 선택적으로 패치 할 값을 취합니다.

patch.object:

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original

>>> @patch('package.module.attribute', sentinel.attribute)
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
...
>>> test()
```

모듈(*builtins*를 포함하는)을 패치하려면 *patch.object()* 대신 *patch()*를 사용하십시오:

```
>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
...
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"
```

필요하면 package.module 형식으로 모듈 이름을 '점으로 표시' 할 수 있습니다:

```
>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

좋은 패턴은 실제로 테스트 메서드 자체를 데코레이트 하는 것입니다:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel.attribute)
...
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original
```

Mock으로 패치하려면, 하나의 인자만으로 *patch()*를 사용할 수 있습니다 (또는 두 개의 인자로 *patch.object()*). 모의 객체가 여러분을 위해 만들어지고 테스트 함수 / 메서드로 전달됩니다:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
...         SomeClass.static_method()
...         mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()
```

이 패턴을 사용하여 여러 패치 데코레이터를 쌓을 수 있습니다:

```
>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1, MockClass1)
...         self.assertIs(package.module.ClassName2, MockClass2)
...
>>> MyTest('test_something').test_something()
```

패치 데코레이터를 중첩할 때 모의 객체는 적용한 순서와 같은 순서(데코레이터가 적용되는 일반적인 파이썬 순서)로 데코레이트 된 함수로 전달됩니다. 이것은 밑에서 위로 올라가는 순서를 뜻해서, 위의 예에서 `test_module.ClassName2`의 모의 객체가 먼저 전달됩니다.

스코프 도중 딕셔너리에 값을 설정하고 테스트가 끝날 때 딕셔너리를 원래 상태로 복원하기 위한 *patch.dict()*도 있습니다:

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

`patch`, `patch.object` 및 `patch.dict`는 모두 컨텍스트 관리자로 사용할 수 있습니다.

*patch()*를 사용하여 모의 객체를 만드는 곳에서, with 문의 "as" 형식을 사용하여 모의 객체에 대한 참조를 얻을 수 있습니다:

```
>>> class ProductionClass:
...     def method(self):
...         pass
...
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
>>> mock_method.assert_called_with(1, 2, 3)
```

대안으로 patch, patch.object 및 patch.dict는 클래스 데코레이터로 사용될 수 있습니다. 이런 식으로 사용될 때 이름이 "test" 로 시작하는 모든 메서드에 데코레이터를 개별적으로 적용하는 것과 같습니다.

26.7.3 추가 예

다음은 약간 더 고급 시나리오에 대한 몇 가지 예입니다.

연쇄 호출 모킹하기

일단 *return_value* 어트리뷰트를 이해하면 연쇄 호출 모킹은 모의 객체를 사용하면 실제로 간단합니다. 모의 객체가 처음 호출되거나 호출되기 전에 return_value를 가져오면, 새 *Mock*이 만들어집니다.

이것은 return_value 모의 객체를 조사하여 모킹 된 객체에 대한 호출에서 반환된 객체가 어떻게 사용되었는지 확인할 수 있음을 뜻합니다:

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

여기서부터는 구성하고 연쇄 호출에 대한 어서션을 만드는 간단한 단계입니다. 물론 또 다른 대안은 처음부터 더 테스트하기 쉬운 방식으로 코드를 작성하는 것입니다…

그래서, 다음과 같은 코드가 있다고 가정합시다:

```
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
...         response = self.backend.get_endpoint('foobar').create_call('spam',
↪'eggs').start_call()
...         # more code
```

BackendProvider 가 이미 잘 테스트 되었다고 가정하면, method() 를 어떻게 테스트해야 할까요? 특히, 코드 섹션 # more code가 response 객체를 올바른 방식으로 사용하는지 테스트하고 싶습니다.

이 호출의 연쇄는 인스턴스 어트리뷰트에서 이루어지기 때문에 Something 인스턴스에서 backend 어트리뷰트를 몽키 패치 할 수 있습니다. 이 특별한 경우에는 start_call에 대한 최종 호출의 반환 값에만 관심이 있어서 해야 할 구성이 많지 않습니다. 반환하는 객체가 '파일류(file-like)' 라고 가정하고, response 객체가 spec으로 내장 *open()*을 사용하도록 할 것입니다.

이를 위해 모의 백 엔드로 모의 인스턴스를 만들고 모의 response 객체를 만듭니다. 응답을 최종 start_call의 반환 값으로 설정하기 위해 다음을 수행할 수 있습니다:

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_
↪value = mock_response
```

configure_mock() 메서드를 사용하여 약간 더 좋은 방법으로 반환 값을 직접 설정할 수 있습니다:

```
>>> something = Something()
>>> mock_response = Mock(spec=open)
>>> mock_backend = Mock()
>>> config = {'get_endpoint.return_value.create_call.return_value.start_call.
↪return_value': mock_response}
>>> mock_backend.configure_mock(**config)
```

이것들로 우리는 "모의 백 엔드" 를 몽키 패치하고 실제 호출을 할 수 있습니다:

```
>>> something.backend = mock_backend
>>> something.method()
```

*mock_calls*를 사용하면 단일 어서션으로 연쇄 호출을 확인할 수 있습니다. 연쇄 호출은 한 줄의 코드에 있는 여러 번의 호출이라서, `mock_calls`에는 여러 항목이 있게 됩니다. *call.call_list()*를 사용하여 이 호출의 리스트를 만들 수 있습니다:

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

부분 모킹

어떤 테스트에서 *datetime.date.today()*에 대한 호출이 알려진 날짜를 반환하도록 모킹하려고 했지만, 테스트 중인 코드가 새로운 date 객체를 만들지 못하도록 막고 싶지 않았습니다. 불행히도 *datetime.date*는 C로 작성되었고, 그래서 정적 *datetime.date.today()* 메서드를 그저 몽키 패치 할 수 없었습니다.

date 클래스를 모의 객체로 효과적으로 래핑하지만, 생성자 호출은 실제 클래스로 전달하는 (그리고 진짜 인스턴스를 반환하는) 간단한 방법을 찾았습니다.

patch 데코레이터는 여기서 테스트 중인 모듈에서 `date` 클래스를 모킹하는 데 사용됩니다. 그런 다음 모의 date 클래스의 *side_effect* 어트리뷰트는 실제 날짜를 반환하는 람다 함수로 설정됩니다. 모의 date 클래스가 호출되면 진짜 date가 생성되어 `side_effect`에 의해 반환됩니다.

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args, **kw)
...
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)
```

*datetime.date*를 전역적으로 패치하지 않았음에 유의하십시오. `date`를 사용하는 모듈에서 패치했습니다. 패치할 곳을 참조하십시오.

`date.today()`가 호출되면 알려진 날짜가 반환되지만, `date(...)` 생성자에 대한 호출은 여전히 일반 date를 반환합니다. 이렇게 하지 않으면 테스트 중인 코드와 정확히 같은 알고리즘을 사용하여 예상 결과를 계산해야 할 수 있습니다, 이는 고전적인 테스트 안티 패턴입니다.

date 생성자에 대한 호출은 `mock_date` 어트리뷰트(`call_count`와 그 친구들)에 기록되며 테스트에 유용할 수 있습니다.

date나 기타 내장 클래스 모킹을 처리하는 다른 방법은 이 블로그 페이지에서 다루고 있습니다.

제너레이터 메서드 모킹하기

파이썬 제너레이터는 `yield` 문을 사용하는 함수나 메서드로, 이터레이트 할 때 일련의 값을 반환합니다[1].

제너레이터 메서드 / 함수가 호출되면 제너레이터 객체를 반환합니다. 이터레이트 하는 대상은 제너레이터 객체입니다. 이터레이션을 위한 프로토콜 메서드는 *__iter__()*이고, *MagicMock*을 사용하여 이를 모킹할 수 있습니다.

다음은 제너레이터로 구현된 "iter" 메서드를 갖는 예제 클래스입니다:

```
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
```

(다음 페이지에 계속)

[1] 제너레이터 표현식과 제너레이터의 더 고급 사용도 있지만, 여기서는 다루지 않습니다. 제너레이터와 이것이 얼마나 강력한지에 대한 아주 좋은 소개: Generator Tricks for Systems Programmers.

(이전 페이지에서 계속)

```
...             yield i
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

이 클래스, 특히 "iter" 메서드를 어떻게 모킹할까요?

이터레이션(*list* 호출로 인해 묵시적으로 이루어집니다)에서 반환된 값을 구성하려면, `foo.iter()` 호출에 의해 반환된 객체를 구성해야 합니다.

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

모든 테스트 메서드에 같은 패치 적용하기

여러 테스트 메서드에 대해 여러 패치를 적용하려면 모든 메서드에 패치 데코레이터를 적용하는 것이 가장 확실한 방법입니다. 이것은 불필요한 반복처럼 느낄 수 있습니다. 대신, *patch()*(이것의 모든 다양한 형태)를 클래스 데코레이터로 사용할 수 있습니다. 이것은 클래스의 모든 테스트 메서드에 패치를 적용합니다. 테스트 메서드는 이름이 `test`로 시작하는 메서드로 식별됩니다:

```
>>> @patch('mymodule.SomeClass')
... class MyTest(unittest.TestCase):
...
...     def test_one(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def test_two(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def not_a_test(self):
...         return 'something'
...
>>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'
```

패치를 관리하는 다른 방법은 패처 메서드: *start*와 *stop*을 사용하는 것입니다. 이를 통해 패치를 `setUp`과 `tearDown` 메서드로 옮길 수 있습니다.

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
...     def tearDown(self):
...         self.patcher.stop()
...
>>> MyTest('test_foo').run()
```

이 기법을 사용하면 `stop`을 호출하여 패치가 "실행 취소"되도록 해야 합니다. setUp에서 예외가 발생하면 tearDown이 호출되지 않기 때문에, 생각보다 복잡할 수 있습니다. *unittest.TestCase.addCleanup()*은

이것을 더 쉽게 만듭니다:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('mymodule.foo')
...         self.addCleanup(patcher.stop)
...         self.mock_foo = patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
>>> MyTest('test_foo').run()
```

연결되지 않은 메서드 모킹하기

오늘 테스트를 작성하는 동안 연결되지 않은 메서드(*unbound method*)를 패치해야 했습니다 (인스턴스가 아닌 클래스의 메서드를 패치하는 것입니다). 어떤 객체가 이 특정 메서드를 호출했는지에 대한 어서션을 하고 싶기 때문에 첫 번째 인자로 self가 전달되는 것이 필요했습니다. 문제는 이것을 위해 모의 객체로 패치 할 수 없다는 것인데, 연결되지 않은 메서드를 모의 객체로 바꾸면 인스턴스에서 가져올 때 연결된 메서드가 되지 않아서, self가 전달되지 않기 때문입니다. 해결 방법은 연결되지 않은 메서드를 실제 함수로 대신 패치하는 것입니다. *patch()* 데코레이터는 메서드를 모의 객체로 패치하기 너무 쉽게 만들어서 실제 함수를 만들어야 하는 것이 성가십니다.

`autospec=True`를 패치로 전달하면 실제 함수 객체로 패치가 수행됩니다. 이 함수 객체는 그것이 교체하는 것과 같은 서명을 갖지만, 수면 아래에서 모의 객체로 위임합니다. 전과 똑같은 방식으로 여전히 자동 생성된 모의 객체를 얻습니다. 그것이 의미하는 것은, 클래스에서 연결되지 않은 메서드를 패치하는데 사용하면 모킹 된 함수가 인스턴스에서 꺼낼 때 연결된 메서드로 바뀐다는 것입니다. `self`가 첫 번째 인자로 전달되고, 정확히 제가 원하는 것입니다:

```
>>> class Foo:
...   def foo(self):
...     pass
...
>>> with patch.object(Foo, 'foo', autospec=True) as mock_foo:
...   mock_foo.return_value = 'foo'
...   foo = Foo()
...   foo.foo()
...
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

`autospec=True`를 사용하지 않으면 연결되지 않은 메서드가 대신 Mock 인스턴스로 패치되고, `self`로 호출되지 않습니다.

모의 객체로 여러 호출 확인하기

mock에는 모의 객체가 어떻게 사용되는지에 대한 어서션을 만드는 데 유용한 API가 있습니다.

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

모의 객체가 한 번만 호출되면 *call_count*가 1이라는 것도 어서트하는 *assert_called_once_with()* 메서드를 사용할 수 있습니다.

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Traceback (most recent call last):
    ...
AssertionError: Expected 'foo_bar' to be called once. Called 2 times.
Calls: [call('baz', spam='eggs'), call()].
```

assert_called_with와 assert_called_once_with는 모두 가장 최근 호출에 대한 어서션을 합니다. 모의 객체가 여러 번 호출될 것이고, 이 호출 모두에 대해 어서션을 하고 싶다면 *call_args_list*를 사용할 수 있습니다:

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```

call 도우미를 사용하면 이러한 호출에 대한 어서션을 쉽게 할 수 있습니다. 예상 호출 리스트를 만들어서 call_args_list와 비교할 수 있습니다. 이것은 call_args_list의 repr과 매우 유사합니다:

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

가변 인자에 대처하기

드물지만 여러분을 괴롭힐 수 있는 또 다른 상황은 모의 객체가 가변 인자로 호출되는 경우입니다. call_args와 call_args_list는 인자에 대한 참조를 저장합니다. 테스트 중인 코드에 의해 인자가 변경되면 모의 객체가 호출되었을 때의 값에 대해 더는 어서션 할 수 없습니다.

다음은 문제를 보여주는 예제 코드입니다. 'mymodule' 에 정의된 다음 함수를 상상해보십시오:

```
def frob(val):
    pass

def grob(val):
    "먼저 frob 한 다음 val을 clear 합니다"
    frob(val)
    val.clear()
```

grob이 올바른 인자로 frob을 호출하는지 테스트하려고 할 때 어떤 일이 발생하는지 보십시오:

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
...
>>> val
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
    ...
AssertionError: Expected: (({6},), {})
Called with: ((set(),), {})
```

한가지 가능성은 모의 객체가 전달한 인자를 복사하는 것일 수 있습니다. 그러면 동일성(equality)을 위해 객체 아이덴티티에 의존하는 어서션을 수행하면 문제가 발생할 수 있습니다.

여기에 *side_effect* 함수를 사용하는 한 가지 해결책이 있습니다. 모의 객체에 side_effect 함수를 제공하면 모의 객체와 같은 인자로 side_effect가 호출됩니다. 이는 인자를 복사하여 나중에 어서션을

위해 저장할 기회를 제공합니다. 이 예제에서는 다른 모의 객체를 사용하여 인자를 저장해서 어서션을 수행하기 위해 모의 객체 메서드를 사용할 수 있습니다. 다시 한번 도우미 함수가 이것을 설정합니다.

```
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = {6}
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with({6})
>>> new_mock.call_args
call({6})
```

`copy_call_args`는 호출될 모의로 호출됩니다. 어서션을 수행할 새로운 모의 객체를 반환합니다. `side_effect` 함수는 인자의 사본을 만들고 사본으로 `new_mock`을 호출합니다.

참고

모의 객체를 한 번만 사용하려는 경우 호출 시점에서 인자를 확인하는 쉬운 방법이 있습니다. 단순히 `side_effect` 함수 내에서 확인할 수 있습니다.

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
    ...
AssertionError
```

다른 접근법은 인자를 복사(*copy.deepcopy()*를 사용해서)하는 *Mock*이나 *MagicMock*의 서브 클래스를 만드는 것입니다. 구현 예는 다음과 같습니다:

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, /, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super().__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> c.assert_called_with(arg)
Traceback (most recent call last):
    ...
AssertionError: expected call not found.
Expected: mock({1})
Actual: mock(set())
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

Mock 이나 MagicMock 을 서브 클래싱할 때 모든 동적으로 만들어진 어트리뷰트와 return_value 는 자동으로 서브 클래스를 사용합니다. 즉, CopyingMock 의 모든 자식도 CopyingMock 형이 됩니다.

중첩 패치

patch 를 컨텍스트 관리자로 것이 멋지기는 하지만, 여러 패치를 수행하면 오른쪽으로 점점 더 들여쓰기 되는 문장으로 중첩될 수 있습니다:

```
>>> class MyTest(unittest.TestCase):
...
...     def test_foo(self):
...         with patch('mymodule.Foo') as mock_foo:
...             with patch('mymodule.Bar') as mock_bar:
...                 with patch('mymodule.Spam') as mock_spam:
...                     assert mymodule.Foo is mock_foo
...                     assert mymodule.Bar is mock_bar
...                     assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

unittest cleanup 함수와 패처 메서드: *start* 와 *stop* 을 사용하면 중첩된 들여쓰기 없이 같은 효과를 얻을 수 있습니다. 간단한 도우미 메서드인 create_patch 는 제 자리에서 패치를 설치하고 만들어진 모의 객체를 반환합니다:

```
>>> class MyTest(unittest.TestCase):
...
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
...
...     def test_foo(self):
...         mock_foo = self.create_patch('mymodule.Foo')
...         mock_bar = self.create_patch('mymodule.Bar')
...         mock_spam = self.create_patch('mymodule.Spam')
...
...         assert mymodule.Foo is mock_foo
...         assert mymodule.Bar is mock_bar
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

MagicMock으로 딕셔너리 모킹하기

딕셔너리나 다른 컨테이너 객체를 모킹하여, 여전히 딕셔너리처럼 동작하면서 이에 대한 모든 액세스를 기록하고 싶을 수 있습니다.

딕셔너리처럼 동작하는 *MagicMock*을 사용하고 *side_effect*가 딕셔너리 액세스를 우리의 제어하에 있는 실제 하부 딕셔너리로 위임하게 해서 목적을 달성할 수 있습니다.

MagicMock의 __getitem__() 과 __setitem__() 메서드가 호출될 때 (일반 딕셔너리 액세스), side_effect는 키로 호출됩니다 (그리고 __setitem__의 경우는 값도). 반환되는 것을 제어 할 수도 있습니다.

MagicMock이 사용된 후에 *call_args_list*와 같은 어트리뷰트를 사용하여 딕셔너리가 어떻게 사용되었는지를 어서트할 수 있습니다:

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
...     return my_dict[name]
...
>>> def setitem(name, val):
...     my_dict[name] = val
...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

참고

MagicMock을 사용하는 대신 Mock을 사용하고 오직 원하는 매직 메서드만 제공할 수 있습니다:

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

세 번째 옵션은 MagicMock을 사용하지만, dict를 *spec* (또는 *spec_set*) 인자로 전달하여 만들어진 MagicMock이 딕셔너리 매직 메서드 만 갖도록 하는 것입니다:

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

이러한 부작용 함수가 제자리에 들어가면, mock은 일반 딕셔너리처럼 작동하지만, 액세스를 기록합니다. 존재하지 않는 키에 액세스하려고 하면 *KeyError*를 발생시키기조차 합니다.

```
>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
Traceback (most recent call last):
    ...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'
```

사용된 후에 일반적인 모의 객체 메서드와 어트리뷰트를 사용하여 액세스에 대한 어서션을 할 수 있습니다:

```
>>> mock.__getitem__.call_args_list
[call('a'), call('c'), call('d'), call('b'), call('d')]
>>> mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
>>> my_dict
{'a': 1, 'b': 'fish', 'c': 3, 'd': 'eggs'}
```

Mock 서브 클래스와 그 어트리뷰트

*Mock*을 서브 클래싱하려는 여러 가지 이유가 있습니다. 한 가지 이유는 도우미 메서드를 추가하는 것입니다. 다음은 시시한 예입니다:

```
>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)
>>> mymock
<MyMock id='...'>
>>> mymock.has_been_called()
False
>>> mymock()
>>> mymock.has_been_called()
True
```

`Mock` 인스턴스의 표준 동작은 어트리뷰트와 반환 값 모의 객체가 액세스 되는 모의 객체의 형과 같은 형이라는 것입니다. 이를 통해 `Mock` 어트리뷰트는 `Mock` 이고 `MagicMock` 어트리뷰트는 `MagicMock` 이 됩니다[2]. 따라서 도우미 메서드를 추가하기 위해 서브 클래싱하면 이 메서드는 서브 클래스 인스턴스의 어트리뷰트와 반환 값 모의 객체에서도 사용할 수 있습니다.

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
>>> mymock.foo.has_been_called()
True
```

때때로 이것은 불편합니다. 예를 들어, 한 사용자가 Twisted adaptor 를 만들기 위해 Mock을 서브 클래싱했습니다. 이것을 어트리뷰트에도 적용하면 실제로 에러가 발생합니다.

`Mock`(이것의 모든 종류에서)은 `_get_child_mock` 이라는 메서드를 사용하여 어트리뷰트와 반환 값을 위한 이러한 "서브 모의 객체"를 만듭니다. 이 메서드를 재정의하여 서브 클래스가 어트리뷰트에 사용되지 않도록 할 수 있습니다. 서명은 모의 객체 생성자에 전달되는 임의의 키워드 인자(`**kwargs`)를 취하는 것입니다:

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, /, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
```

(다음 페이지에 계속)

[2] 이 규칙의 예외는 콜러블이 아닌 모의 객체입니다. 어트리뷰트는 콜러블 변형을 사용하는데, 그렇지 않으면 콜러블이 아닌 모의 객체가 콜러블 메서드를 가질 수 없기 때문입니다.

(이전 페이지에서 계속)

```
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

patch.dict로 임포트를 모킹하기

모킹이 어려울 수 있는 한 가지 상황은 함수 내부에 지역 임포트가 있는 경우입니다. 이것들은 모킹하기가 더 어려운데, 우리가 패치 할 수 있는 모듈 이름 공간의 객체를 사용하지 않기 때문입니다.

일반적으로 지역 임포트는 피해야 합니다. 그것들은 때때로 순환 의존성을 막기 위해 수행되는데, 보통 문제를 해결하는 더 좋은 방법(코드를 리팩터 하십시오)이 있습니다. 또는 임포트를 지연시켜서 "선불 비용"을 방지하기 위해 수행합니다. 이 또한 무조건적인 지역 임포트보다 더 나은 방법으로 해결할 수 있습니다 (모듈을 클래스나 모듈 어트리뷰트로 저장하고 처음 사용할 때만 임포트 합니다).

그 외에도 `mock`을 사용하여 임포트 결과에 영향을 주는 방법이 있습니다. 임포트는 *sys.modules* 딕셔너리에서 객체를 가져옵니다. 객체를 가져온다는 것에 유의하십시오, 이것이 모듈일 필요는 없습니다. 처음으로 모듈을 임포트 하면 `sys.modules`에 모듈 객체가 배치되어서, 일반적으로 무언가를 임포트 할 때 모듈을 다시 받습니다. 그러나 반드시 그런 것은 아닙니다.

즉, *patch.dict()*를 사용하여 임시로 *sys.modules*에 모의 객체를 넣을 수 있습니다. 이 패치가 활성화되어있는 동안 모든 임포트는 모의 객체를 가져옵니다. 패치가 완료되면 (데코레이트 된 함수가 종료되거나, with 문 본문이 완료되거나 `patcher.stop()` 이 호출되면) 이전에 있던 모든 것이 안전하게 복원됩니다.

다음은 'foobe' 모듈을 모킹하는 예입니다.

```
>>> import sys
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     import fooble
...     fooble.blob()
...
<Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

보시다시피 `import fooble`은 성공하지만, 끝났을 때 *sys.modules*에는 'foobe' 이 남아 있지 않습니다.

이것은 `from module import name` 형식에서도 작동합니다:

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     from fooble import blob
...     blob.blip()
...
<Mock name='mock.blob.blip()' id='...'>
>>> mock.blob.blip.assert_called_once_with()
```

약간의 추가 작업으로 패키지 임포트를 모킹 할 수도 있습니다:

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...     from package.module import fooble
...     fooble()
...
<Mock name='mock.module.fooble()' id='...'>
>>> mock.module.fooble.assert_called_once_with()
```

호출 순서 추적과 덜 상세한 호출 어서션

Mock 클래스를 사용하면 *method_calls* 어트리뷰트를 통해 모의 객체에서 메서드 호출의 순서를 추적할 수 있습니다. 개별 모의 객체 간의 호출 순서를 추적할 수는 없지만, *mock_calls*를 사용하여 같은 효과를 얻을 수 있습니다.

모의 객체들은 mock_calls에서 자식 모의 객체에 대한 호출을 추적하고, 모의 객체의 임의 어트리뷰트에 액세스하면 자식 모의 객체를 만들기 때문에, 부모 모의 객체로부터 개별 모의 객체를 만들 수 있습니다. 그러면 해당 자식 모의 객체에 대한 호출은 부모의 mock_calls에 순서대로 기록됩니다:

```
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar
```

```
>>> mock_foo.something()
<Mock name='mock.foo.something()' id='...'>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='...'>
```

```
>>> manager.mock_calls
[call.foo.something(), call.bar.other.thing()]
```

그런 다음 관리자 모의 객체의 mock_calls 어트리뷰트와 비교하여 순서를 포함하여 호출에 대해 어서션할 수 있습니다:

```
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

patch가 모의 객체를 만들고 제자리에 배치하는 경우, *attach_mock()* 메서드를 사용하여 모의 객체를 관리자 모의 객체에 연결할 수 있습니다. 연결 후 호출은 관리자의 mock_calls에 기록됩니다.

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
...         manager.attach_mock(MockClass2, 'MockClass2')
...         MockClass1().foo()
...         MockClass2().bar()
<MagicMock name='mock.MockClass1().foo()' id='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>
>>> manager.mock_calls
[call.MockClass1(),
call.MockClass1().foo(),
call.MockClass2(),
call.MockClass2().bar()]
```

많은 호출이 이루어졌지만, 특정 시퀀스에만 관심이 있다면 대안은 *assert_has_calls()* 메서드를 사용하는 것입니다. 이것은 호출 리스트를 취합니다 (*call* 객체로 구성됩니다). 해당 호출 시퀀스가 *mock_calls*에 있으면 어서션이 성공합니다.

```
>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='...'>
>>> m.one().two().three()
<MagicMock name='mock.one().two().three()' id='...'>
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)
```

연쇄 호출 `m.one().two().three()` 가 모의 객체에 대한 호출의 전부는 아니지만, 어서션이 여전히 성공합니다.

때로는 모의 객체에 여러 번의 호출이 있을 수 있고, 그 호출들의 일부에 대만 어서션에만 관심이 있을 수 있습니다. 순서에 신경 쓰지 않을 수도 있습니다. 이 경우 `any_order=True` 를 `assert_has_calls` 로 전달할 수 있습니다:

```
>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]
>>> m.assert_has_calls(calls, any_order=True)
```

더 복잡한 인자 일치

ANY 와 같은 기본 개념을 사용하여 모의 객체에 인자로 사용되는 객체에 대해 더 복잡한 어서션을 수행하도록 매처(matchers)를 구현할 수 있습니다.

기본적으로 객체 아이덴티티에 기반하여 를 기준으로 같다고 비교되는 (이것이 사용자 정의 클래스의 파이썬 기본 동작입니다) 어떤 객체가 모의 객체로 전달되기를 기대한다고 가정합시다. *assert_called_with()* 를 사용하려면 정확히 같은 객체를 전달해야 합니다. 이 객체의 일부 어트리뷰트에만 관심이 있다면 이러한 어트리뷰트를 확인하는 매처를 만들 수 있습니다.

이 예에서 `assert_called_with` 에 대한 '표준' 호출이 충분하지 않다는 것을 볼 수 있습니다:

```
>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
Traceback (most recent call last):
    ...
AssertionError: expected call not found.
Expected: mock(<__main__.Foo object at 0x...>)
Actual: mock(<__main__.Foo object at 0x...>)
```

`Foo` 클래스를 위한 비교 함수는 다음과 같습니다:

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

그리고 동등 비교 연산에 이와 같은 비교 함수를 사용할 수 있는 매처 객체는 다음과 같습니다:

```
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
```

이 모든 것을 종합하면:

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

Matcher는 compare 함수와 비교하려는 Foo 객체로 인스턴스 화 됩니다. assert_called_with에서는 Matcher 동등 비교 메서드가 호출되는데, 이 메서드는 모의 객체가 호출된 객체와 우리가 매처를 만들 때 제공한 객체를 비교합니다. 일치하면 assert_called_with가 통과하고, 그렇지 않으면 *AssertionError*가 발생합니다:

```
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
Traceback (most recent call last):
    ...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

약간의 조정만으로 비교 함수가 *AssertionError*를 직접 발생시키고 더 유용한 실패 메시지를 제공할 수 있습니다.

버전 1.5부터, 파이썬 테스트 라이브러리 PyHamcrest는 여기에서 유용할 수 있는 유사한 기능을 동등 비교 매처의 형태로 제공합니다 (hamcrest.library.integration.match_equality).

26.8 test — 파이썬 용 회귀 테스트 패키지

참고

test 패키지는 파이썬 내부 용으로만 사용됩니다. 파이썬의 핵심 개발자를 위해 설명됩니다. 여기에 언급된 코드는 파이썬 릴리스 사이에 예고 없이 변경되거나 제거될 수 있어서, 파이썬의 표준 라이브러리 외부에서 이 패키지를 사용하는 것은 권장되지 않습니다.

---

test 패키지에는 *test.support*와 *test.regrtest*뿐만 아니라 파이썬에 대한 모든 회귀 테스트가 포함되어 있습니다. *test.support*는 테스트를 향상하는 데 사용되며 *test.regrtest*는 테스트 스위트를 구동합니다.

이름이 test_로 시작하는 *test* 패키지의 각 모듈은 특정 모듈이나 기능에 대한 테스트 스위트입니다. 모든 새로운 테스트는 *unittest*나 *doctest* 모듈을 사용하여 작성해야 합니다. 일부 오래된 테스트는 sys.stdout으로 인쇄된 출력을 비교하는 "전통적인" 테스트 스타일을 사용하여 작성되었습니다; 이 테스트 스타일은 폐지된 것으로 간주합니다.

더 보기

모듈 *unittest*
PyUnit 회귀 테스트 작성.

모듈 *doctest*
독스트링에 포함된 테스트.

26.8.1 test 패키지를 위한 단위 테스트 작성하기

unittest 모듈을 사용하는 테스트는 몇 가지 지침을 따르는 것이 좋습니다. 하나는 테스트 모듈의 이름을 test_로 시작하고 테스트 중인 모듈의 이름으로 끝나도록 짓는 것입니다. 테스트 모듈의 테스트 메서드는 test_로 시작하고 메서드가 테스트하는 내용에 대한 설명으로 끝나야 합니다. 이는 테스트 드라이버가 메서드를 테스트 메서드로 인식하기 위해 필요합니다. 또한, 메서드에 대한 독스트링이 포함되어서는 안 됩니다. 주석(가령 # Tests function returns only True or False)을 사용하여 테스트 메서드에

대한 설명을 제공해야 합니다. 이는 독스트링이 존재하면 이것이 인쇄되어 실행되는 테스트가 인쇄되지 않기 때문입니다.

기본 상용구가 자주 사용됩니다:

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # setUp()과 tearDown()은 필요할 때만 사용합니다

    def setUp(self):
        ... 테스트 준비를 위해 실행할 코드 ...

    def tearDown(self):
        ... 테스트 후 정리를 위해 실행할 코드 ...

    def test_feature_one(self):
        # feature one을 테스트합니다.
        ... 테스트 코드 ...

    def test_feature_two(self):
        # feature two를 테스트합니다.
        ... 테스트 코드 ...

    ... 더 많은 테스트 메서드 ...

class MyTestCase2(unittest.TestCase):
    ... MyTestCase1 과 같은 구조 ...

... 더 많은 테스트 클래스 ...

if __name__ == '__main__':
    unittest.main()
```

이 코드 패턴을 사용하면 *test.regrtest*에서 자체적으로 *unittest* CLI를 지원하는 스크립트로나 `python -m unittest` CLI를 통해 테스트 스위트를 실행할 수 있습니다.

회귀 테스트의 목표는 코드를 깨려고 시도하는 것입니다. 이는 따라야 할 몇 가지 지침으로 이어집니다:

- 테스트 스위트는 모든 클래스, 함수 및 상수를 괴롭혀야 합니다. 여기에는 외부 세계에 제공되는 외부 API뿐만 아니라 "내부(private)" 코드도 포함됩니다.
- 화이트 박스 테스트 (테스트 작성 시 테스트 중인 코드 검사)가 선호됩니다. 블랙박스 테스트(게시된 사용자 인터페이스 만 테스트)는 모든 경계와 에지 케이스가 테스트 되었는지 확인하기에 충분하지 않습니다.
- 유효하지 않은 값을 포함하여 가능한 모든 값이 테스트 되었는지 확인하십시오. 이렇게 하면 모든 유효한 값이 받아들여질 뿐만 아니라 부적절한 값이 올바르게 처리되었는지 확인하게 됩니다.
- 가능한 한 많은 코드 경로를 소진하십시오. 분기가 발생하는 위치를 테스트하고 입력을 조정하여 코드에서 여러 경로가 사용되는지 확인합니다.
- 테스트 된 코드에 대해 발견된 버그에 대한 명시적 테스트를 추가합니다. 이렇게 하면 나중에 코드가 변경되어도 에러가 다시 발생하는지 확인합니다.
- 테스트 후에 정리해야 합니다 (가령 모든 임시 파일 닫고 제거하기).
- 테스트가 운영 체제의 특정 조건에 의존하면 테스트를 시도하기 전에 조건이 이미 존재하는지 확인하십시오.

- 가능한 한 적은 수의 모듈을 임포트 하고 가능한 한 빨리 수행하십시오. 이렇게 하면 테스트의 외부 종속성이 최소화되고 모듈 임포트의 부작용에 따른 비정상적인 동작이 최소화됩니다.
- 코드 재사용을 극대화하십시오. 때에 따라, 테스트는 사용되는 입력 유형에 따라 조금씩 달라집니다. 입력을 지정하는 클래스로 기본 테스트 클래스를 서브 클래싱하여 코드 중복을 최소화합니다:

```
class TestFuncAcceptsSequencesMixin:

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = (1, 2, 3)
```

이 패턴을 사용할 때, *unittest.TestCase*를 상속한 모든 클래스가 테스트로 실행된다는 점을 기억하십시오. 위 예제의 `TestFuncAcceptsSequencesMixin` 클래스는 데이터가 없어서 자체적으로 실행할 수 없기 때문에, *unittest.TestCase*를 상속하지 않습니다.

더 보기

테스트 주도 개발(Test Driven Development)
코드 전에 테스트를 작성하는 것에 관한 Kent Beck의 책.

26.8.2 명령 줄 인터페이스를 사용하여 테스트 실행하기

`-m` 옵션 덕분에 *test* 패키지를 스크립트로 실행하여 파이썬의 회귀 테스트 스위트를 구동 할 수 있습니다: **python -m test**. 내부적으로는 *test.regrtest*를 사용합니다; 이전 파이썬 버전에서 사용된 호출 **python -m test.regrtest**는 여전히 작동합니다. 스크립트를 단독으로 실행하면 *test* 패키지의 모든 회귀 테스트 실행이 자동으로 시작됩니다. 패키지에서 이름이 `test_`로 시작하는 모든 모듈을 찾아서, 임포트하고, `test_main()` 함수가 있으면 실행하고, `test_main`이 없으면 unittest.TestLoader.loadTestsFromModule을 통해 테스트를 로드하여 이를 수행합니다. 실행할 테스트 이름도 스크립트에 전달할 수 있습니다. 단일 회귀 테스트를 지정하면 (**python -m test test_spam**) 출력이 최소화되고 테스트의 통과나 실패 여부만 인쇄됩니다.

*test*를 직접 실행하면 테스트에 사용할 수 있는 리소스를 설정할 수 있습니다. `-u` 명령 줄 옵션을 사용하여 이 작업을 수행합니다. `-u` 옵션의 값으로 `all`을 지정하면 가능한 모든 자원을 사용할 수 있습니다: **python -m test -uall**. 하나를 제외한 모든 리소스가 필요한 경우 (더 흔한 경우입니다), 원하지 않는 리소스의 쉼표로 구분된 목록이 `all` 뒤에 나열될 수 있습니다. **python -m test -uall,-audio,-largefile** 명령은 `audio`와 `largefile` 리소스를 제외한 모든 리소스로 *test*를 실행합니다. 모든 리소스와 추가 명령 줄 옵션 목록을 보려면, **python -m test -h**를 실행하십시오.

회귀 테스트를 실행하는 다른 방법은 테스트가 실행되는 플랫폼에 따라 다릅니다. 유닉스에서는, 파이썬이 빌드된 최상위 디렉터리에서 **make test**를 실행할 수 있습니다. 윈도우에서는, `PCbuild` 디렉터리에서 **rt.bat**을 실행하면 모든 회귀 테스트가 실행됩니다.

Added in version 3.14: Output is colorized by default and can be controlled using environment variables.

26.9 test.support — 파이썬 테스트 스위트용 유틸리티

test.support 모듈은 파이썬의 회귀 테스트 스위트를 지원합니다.

> **참고**
>
> *test.support*는 공용 모듈이 아닙니다. 파이썬 개발자가 테스트를 작성하는 데 도움이 되도록 여기에 설명하고 있습니다. 이 모듈의 API는 릴리스 간에 하위 호환성에 대한 고려 없이 변경될 수 있습니다.

이 모듈은 다음 예외를 정의합니다:

exception test.support.**TestFailed**

테스트가 실패할 때 발생하는 예외. 이것은 폐지되었고 *unittest* 기반 테스트와 *unittest.TestCase*의 어서션 메서드로 대체합니다.

exception test.support.**ResourceDenied**

*unittest.SkipTest*의 서브 클래스. 리소스(가령 네트워크 연결)를 사용할 수 없을 때 발생합니다. *requires()* 함수에 의해 발생합니다.

test.support 모듈은 다음 상수를 정의합니다:

test.support.**verbose**

상세 출력이 활성화될 때 `True`. 실행 중인 테스트에 대한 자세한 정보가 필요할 때 확인해야 합니다. *verbose*는 *test.regrtest*에 의해 설정됩니다.

test.support.**is_jython**

실행 중인 인터프리터가 Jython이면 `True`.

test.support.**is_android**

`True` if `sys.platform` is `android`.

test.support.**is_emscripten**

`True` if `sys.platform` is `emscripten`.

test.support.**is_wasi**

`True` if `sys.platform` is `wasi`.

test.support.**is_apple_mobile**

`True` if `sys.platform` is `ios`, `tvos`, or `watchos`.

test.support.**is_apple**

`True` if `sys.platform` is `darwin` or `is_apple_mobile` is `True`.

test.support.**unix_shell**

윈도우가 아니면 셸 경로; 그렇지 않으면 `None`.

test.support.**LOOPBACK_TIMEOUT**

`127.0.0.1`과 같은 네트워크 로컬 루프 백 인터페이스에서 리스닝하는 네트워크 서버를 사용하는 테스트의 초 단위 제한 시간.

제한 시간은 테스트 실패를 방지 할 수 있을 만큼 깁니다: 클라이언트와 서버가 다른 스레드나 다른 프로세스에서 실행될 수 있다는 점을 고려합니다.

제한 시간은 *socket.socket*의 *connect()*, *recv()* 및 *send()* 메서드를 위해 충분히 길어야 합니다.

기본값은 5초입니다.

*INTERNET_TIMEOUT*도 참조하십시오.

test.support.**INTERNET_TIMEOUT**

인터넷으로 가는 네트워크 요청에 대한 초 단위 제한 시간.

어떤 이유로 든 인터넷 요청이 블록 되면 테스트가 너무 오래 기다리지 않을 만큼 제한 시간이 적당히 짧습니다.

일반적으로, *INTERNET_TIMEOUT*을 사용하는 시간 초과는 테스트를 실패로 표시해서는 안 되며, 대신 테스트를 건너뜁니다: *transient_internet()*을 참조하십시오.

기본값은 1분입니다.

*LOOPBACK_TIMEOUT*도 참조하십시오.

test.support.**SHORT_TIMEOUT**

테스트가 "너무 오래" 걸리면 테스트를 실패로 표시하는 초 단위 제한 시간.

제한 시간 값은 regrtest `--timeout` 명령 줄 옵션에 따라 다릅니다.

*SHORT_TIMEOUT*을 사용하는 테스트가 느린 빌드 봇에서 무작위로 실패하기 시작하면, 대신 *LONG_TIMEOUT*을 사용합니다.

기본값은 30초입니다.

test.support.**LONG_TIMEOUT**

테스트 멈춤을 감지하기 위한 초 단위 제한 시간.

가장 느린 파이썬 빌드 봇에서 테스트 실패의 위험을 줄이기에 충분히 깁니다. 테스트가 "너무 오래" 걸리면 테스트를 실패로 표시하는 데 사용해서는 안 됩니다. 제한 시간 값은 regrtest `--timeout` 명령 줄 옵션에 따라 다릅니다.

기본값은 5분입니다.

LOOPBACK_TIMEOUT, *INTERNET_TIMEOUT* 및 *SHORT_TIMEOUT*도 참조하십시오.

test.support.**PGO**

PGO에 유용하지 않은 테스트를 건너뛸 수 있을 때 설정합니다.

test.support.**PIPE_MAX_SIZE**

쓰기 블로킹을 일으키기 위해, 하부 OS 파이프 버퍼 크기보다 클 가능성이 높은 상수.

test.support.**Py_DEBUG**

`True` if Python was built with the `Py_DEBUG` macro defined, that is, if Python was built in debug mode.

Added in version 3.12.

test.support.**SOCK_MAX_SIZE**

쓰기 블로킹을 일으키기 위해, 하부 OS 소켓 버퍼 크기보다 클 가능성이 높은 상수.

test.support.**TEST_SUPPORT_DIR**

*test.support*를 포함하는 최상위 디렉터리로 설정합니다.

test.support.**TEST_HOME_DIR**

테스트 패키지의 최상위 디렉터리로 설정합니다.

test.support.**TEST_DATA_DIR**

테스트 패키지 내의 `data` 디렉터리로 설정합니다.

test.support.**MAX_Py_ssize_t**

대용량 메모리 테스트를 위해 *sys.maxsize*로 설정합니다.

test.support.**max_memuse**

대용량 메모리 테스트를 위한 메모리 제한으로 *set_memlimit()*에 의해 설정됩니다. *MAX_Py_ssize_t*에 의해 제한됩니다.

test.support.**real_max_memuse**

대용량 메모리 테스트를 위한 메모리 제한으로 *set_memlimit()*에 의해 설정됩니다. *MAX_Py_ssize_t*에 의해 제한되지 않습니다.

test.support.**MISSING_C_DOCSTRINGS**

Set to `True` if Python is built without docstrings (the `WITH_DOC_STRINGS` macro is not defined). See the `configure --without-doc-strings` option.

See also the *HAVE_DOCSTRINGS* variable.

test.support.**HAVE_DOCSTRINGS**

Set to `True` if function docstrings are available. See the `python -OO` option, which strips docstrings of functions implemented in Python.

See also the *MISSING_C_DOCSTRINGS* variable.

test.support.**TEST_HTTP_URL**

네트워크 테스트를 위한 전용 HTTP 서버의 URL을 정의합니다.

test.support.**ALWAYS_EQ**

모든 것과 같은 객체. 혼합형 비교를 테스트하는 데 사용됩니다.

test.support.**NEVER_EQ**

어떤 것과도 같지 않은 객체 (*ALWAYS_EQ*에도 해당합니다). 혼합형 비교를 테스트하는 데 사용됩니다.

test.support.**LARGEST**

모든 것보다 큰 객체 (자신은 제외하고). 혼합형 비교를 테스트하는 데 사용됩니다.

test.support.**SMALLEST**

모든 것보다 작은 객체 (자신은 제외하고). 혼합형 비교를 테스트하는 데 사용됩니다.

test.support 모듈은 다음 함수를 정의합니다:

test.support.**busy_retry**(*timeout, err_msg=None, /, *, error=True*)

Run the loop body until `break` stops the loop.

After *timeout* seconds, raise an *AssertionError* if *error* is true, or just stop the loop if *error* is false.

예제:

```
for _ in support.busy_retry(support.SHORT_TIMEOUT):
    if check():
        break
```

error=False 사용 예:

```
for _ in support.busy_retry(support.SHORT_TIMEOUT, error=False):
    if check():
        break
else:
    raise RuntimeError('my custom error')
```

test.support.**sleeping_retry**(*timeout, err_msg=None, /, *, init_delay=0.010, max_delay=1.0, error=True*)

Wait strategy that applies exponential backoff.

Run the loop body until `break` stops the loop. Sleep at each loop iteration, but not at the first iteration. The sleep delay is doubled at each iteration (up to *max_delay* seconds).

See *busy_retry()* documentation for the parameters usage.

Example raising an exception after SHORT_TIMEOUT seconds:

```
for _ in support.sleeping_retry(support.SHORT_TIMEOUT):
    if check():
        break
```

error=False 사용 예:

```
for _ in support.sleeping_retry(support.SHORT_TIMEOUT, error=False):
    if check():
        break
else:
    raise RuntimeError('my custom error')
```

test.support.**is_resource_enabled**(*resource*)

*resource*가 활성화되고 사용할 수 있으면 True를 반환합니다. 사용 가능한 리소스 목록은 *test.regrtest*가 테스트를 실행할 때만 설정됩니다.

test.support.**python_is_optimized**()

파이썬이 -O0이나 -Og로 빌드되지 않았으면 True를 반환합니다.

test.support.**with_pymalloc**()

_testcapi.WITH_PYMALLOC을 반환합니다.

test.support.**requires**(*resource*, *msg=None*)

*resource*를 사용할 수 없으면 *ResourceDenied*를 발생시킵니다. *msg*는 *ResourceDenied*가 발생한다면 이에 대한 인자입니다. __name__이 '__main__'인 함수에 의해 호출되면 항상 True를 반환합니다. *test.regrtest*에서 테스트를 실행할 때 사용됩니다.

test.support.**sortdict**(*dict*)

정렬된 키로 *dict*의 repr을 반환합니다.

test.support.**findfile**(*filename*, *subdir=None*)

*filename*이라는 파일의 경로를 반환합니다. 일치하는 것이 없으면 *filename*이 반환됩니다. 이것은 파일의 경로일 수 있어서 실패와 같지 않습니다.

subdir 설정은 경로 디렉터리를 직접 찾는 대신 파일을 찾는 데 사용할 상대 경로를 나타냅니다.

test.support.**get_pagesize**()

Get size of a page in bytes.

Added in version 3.12.

test.support.**setswitchinterval**(*interval*)

*sys.setswitchinterval()*을 주어진 *interval*로 설정합니다. 시스템이 멈추는 것을 방지하기 위해 안드로이드 시스템을 위한 최소 간격을 정의합니다.

test.support.**check_impl_detail**(**guards*)

이 검사를 사용하여 CPython의 구현 별 테스트를 보호하거나 인자로 보호되는 구현에서만 실행합니다. 이 함수는 호스트 플랫폼에 따라 True나 False를 반환합니다. 사용 예:

```
check_impl_detail()               # CPython에서만 (기본값).
check_impl_detail(jython=True)    # Jython에서만.
check_impl_detail(cpython=False)  # CPython을 제외한 모든 곳.
```

test.support.**set_memlimit**(*limit*)

대용량 메모리 테스트를 위해 *max_memuse*와 *real_max_memuse* 값을 설정합니다.

test.support.**record_original_stdout**(*stdout*)

*stdout*의 값을 저장합니다. regrtest가 시작될 때 stdout을 잡기 위한 것입니다.

test.support.**get_original_stdout**()

*record_original_stdout()*에 의해 설정된 원래 stdout이나 설정되지 않았으면 sys.stdout을 반환합니다.

test.support.**args_from_interpreter_flags**()

sys.flags와 sys.warnoptions의 현재 설정을 재현하는 명령 줄 인자 리스트를 반환합니다.

test.support.**optim_args_from_interpreter_flags**()

sys.flags의 현재 최적화 설정을 재현하는 명령 줄 인자 리스트를 반환합니다.

test.support.**captured_stdin**()

test.support.**captured_stdout**()

test.support.**captured_stderr**()

명명된 스트림을 *io.StringIO* 객체로 일시적으로 대체하는 컨텍스트 관리자.

출력 스트림 사용 예:

```
with captured_stdout() as stdout, captured_stderr() as stderr:
    print("hello")
    print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"
```

입력 스트림 사용 예:

```
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # sys.stdin에서 소비하는 테스트 코드를 호출합니다
    captured = input()
self.assertEqual(captured, "hello")
```

test.support.**disable_faulthandler**()

*faulthandler*를 일시적으로 비활성화하는 컨텍스트 관리자.

test.support.**gc_collect**()

가능한 한 많은 객체를 수거하도록 강제합니다. 이는 가비지 수거기가 적시에 할당 해제를 보장하지 않기 때문에 필요합니다. 이는 __del__ 메서드가 예상보다 늦게 호출될 수 있고 약한 참조(weakrefs)가 예상보다 오래 살아있을 수 있음을 의미합니다.

test.support.**disable_gc**()

진입할 때 가비지 수거기를 비활성화하는 컨텍스트 관리자. 탈출할 때, 가비지 수거기는 이전 상태로 복원됩니다.

test.support.**swap_attr**(*obj*, *attr*, *new_val*)

어트리뷰트를 새 객체로 스와프하는 컨텍스트 관리자.

용법:

```
with swap_attr(obj, "attr", 5):
    ...
```

이렇게 하면 with 블록의 기간 중 obj.attr이 5로 설정되고, 블록 끝에서 이전 값이 복원됩니다. obj에 attr이 없으면, 만들어지고 블록의 끝에서 삭제됩니다.

이전 값(또는 존재하지 않으면 None)이 "as" 절의 대상(있다면)에 대입됩니다.

test.support.**swap_item**(*obj*, *attr*, *new_val*)

항목을 새 객체로 스와프하는 컨텍스트 관리자.

용법:

```
with swap_item(obj, "item", 5):
    ...
```

이렇게 하면 `with` 블록의 기간 중 `obj["item"]` 이 5로 설정되고, 블록 끝에서 이전 값이 복원됩니다. `obj` 에 `item` 이 없으면, 만들어지고 블록의 끝에서 삭제됩니다.

이전 값(또는 존재하지 않으면 `None`)이 "as" 절의 대상(있다면)에 대입됩니다.

`test.support.`**`flush_std_streams`**`()`

Call the `flush()` method on *sys.stdout* and then on *sys.stderr*. It can be used to make sure that the logs order is consistent before writing into stderr.

Added in version 3.11.

`test.support.`**`print_warning`**`(`*msg*`)`

sys.__stderr__ 에 경고를 인쇄합니다. 메시지를 다음처럼 포맷합니다: `f"Warning -- {msg}"`. *msg* 가 여러 줄로 구성되면, 각 줄에 `"Warning -- "` 접두사를 추가합니다.

Added in version 3.9.

`test.support.`**`wait_process`**`(`*pid*, *, *exitcode*, *timeout=None*`)`

프로세스 *pid* 가 완료될 때까지 기다렸다가 프로세스 종료 코드가 *exitcode* 인지 확인합니다.

프로세스 종료 코드가 *exitcode* 와 같지 않으면 *AssertionError* 를 발생시킵니다.

프로세스가 *timeout*(기본적으로 *SHORT_TIMEOUT*) 초보다 오래 실행되면, 프로세스를 죽이고 *AssertionError* 를 발생시킵니다. 제한 시간 기능은 윈도우에서 사용할 수 없습니다.

Added in version 3.9.

`test.support.`**`calcobjsize`**`(`*fmt*`)`

Return the size of the `PyObject` whose structure members are defined by *fmt*. The returned value includes the size of the Python object header and alignment.

`test.support.`**`calcvobjsize`**`(`*fmt*`)`

Return the size of the `PyVarObject` whose structure members are defined by *fmt*. The returned value includes the size of the Python object header and alignment.

`test.support.`**`checksizeof`**`(`*test*, *o*, *size*`)`

테스트 케이스 *test* 에 대해, *o* 의 `sys.getsizeof` 에 GC 헤더 크기를 더한 값이 *size* 와 같다고 어서션합니다.

`@test.support.`**`anticipate_failure`**`(`*condition*`)`

unittest.expectedFailure() 로 테스트를 조건부로 표시하는 데코레이터. 이 데코레이터를 사용하려면 관련 추적기(tracker) 이슈를 식별하는 관련 주석이 있어야 합니다.

`test.support.`**`system_must_validate_cert`**`(`*f*`)`

TLS 인증서 유효성 검사 실패 시 데코레이트된 테스트를 건너뛰는 데코레이터.

`@test.support.`**`run_with_locale`**`(`*catstr*, **locales*`)`

다른 로케일에서 함수를 실행하기 위한 데코레이터로, 완료된 후 올바르게 재설정합니다. *catstr* 은 문자열로 된 로케일 범주입니다 (예를 들어 `"LC_ALL"`). 전달된 *locales* 는 순차적으로 시도되며, 첫 번째 유효한 로케일이 사용됩니다.

`@test.support.`**`run_with_tz`**`(`*tz*`)`

특정 시간대에서 함수를 실행하기 위한 데코레이터로, 완료된 후 올바르게 재설정합니다.

`@test.support.`**`requires_freebsd_version`**`(`**min_version*`)`

FreeBSD 에서 테스트를 실행할 때 최소 버전을 위한 데코레이터. FreeBSD 버전이 최소 버전보다 낮으면, 테스트를 건너뜁니다.

`@test.support.`**`requires_linux_version`**`(`**min_version*`)`

리눅스에서 테스트를 실행할 때 최소 버전을 위한 데코레이터. 리눅스 버전이 최소 버전보다 낮으면, 테스트를 건너뜁니다.

@test.support.**requires_mac_version**(**min_version*)

macOS에서 테스트를 실행할 때 최소 버전을 위한 데코레이터. macOS 버전이 최소 버전보다 낮으면, 테스트를 건너뜁니다.

@test.support.**requires_gil_enabled**

Decorator for skipping tests on the free-threaded build. If the *GIL* is disabled, the test is skipped.

@test.support.**requires_IEEE_754**

비 IEEE 754 플랫폼에서 테스트를 건너뛰는 데코레이터.

@test.support.**requires_zlib**

*zlib*가 없으면 테스트를 건너뛰는 데코레이터.

@test.support.**requires_gzip**

*gzip*이 없으면 테스트를 건너뛰는 데코레이터.

@test.support.**requires_bz2**

*bz2*가 없으면 테스트를 건너뛰는 데코레이터.

@test.support.**requires_lzma**

*lzma*가 없으면 테스트를 건너뛰는 데코레이터.

@test.support.**requires_resource**(*resource*)

*resource*를 사용할 수 없으면 테스트를 건너뛰는 데코레이터.

@test.support.**requires_docstrings**

*HAVE_DOCSTRINGS*일 때만 테스트를 실행하는 데코레이터.

@test.support.**requires_limited_api**

제한된 C API를 사용할 수 있을 때만 테스트를 실행하는 데코레이터.

@test.support.**cpython_only**

CPython에만 적용되는 테스트용 데코레이터.

@test.support.**impl_detail**(*msg=None*, ***guards*)

*guards*에 *check_impl_detail()*을 호출하는 데코레이터. `False`가 반환되면, 테스트를 건너뛰는 이유로 *msg*를 사용합니다.

@test.support.**thread_unsafe**(*reason=None*)

Decorator for marking tests as thread-unsafe. This test always runs in one thread even when invoked with `--parallel-threads`.

@test.support.**no_tracing**

테스트 기간 중 일시적으로 추적을 해제하는 데코레이터.

@test.support.**refcount_test**

참조 횟수를 수반하는 테스트를 위한 데코레이터. 데코레이터는 CPython에 의해 실행되지 않으면 테스트를 실행하지 않습니다. 추적 함수로 인한 예기치 않은 참조 횟수를 방지하기 위해 테스트 기간 중 모든 추적 함수가 설정 해제됩니다.

@test.support.**bigmemtest**(*size*, *memuse*, *dry_run=True*)

bigmem 테스트를 위한 데코레이터.

*size*는 테스트를 위해 요청된 크기입니다(임의의 테스트가 해석하는 단위.) *memuse*는 테스트 단위당 바이트 수, 또는 이의 적절한 추정치입니다. 예를 들어, 각각 4GiB인 두 개의 바이트 버퍼가 필요한 테스트는 `@bigmemtest(size=_4G, memuse=2)`로 데코레이트 될 수 있습니다.

size 인자는 일반적으로 데코레이트 된 테스트 메서드에 추가 인자로 전달됩니다. *dry_run*이 `True`이면, 테스트 메서드에 전달된 값이 요청된 값보다 작을 수 있습니다. *dry_run*이 `False`이면, `-M`가 지정되지 않은 경우 테스트가 더미 실행을 지원하지 않음을 의미합니다.

@test.support.**bigaddrspacetest**

주소 공간을 채우는 테스트용 데코레이터.

test.support.**linked_to_musl**()

Return `False` if there is no evidence the interpreter was compiled with `musl`, otherwise return a version triple, either `(0, 0, 0)` if the version is unknown, or the actual version if it is known. Intended for use in `skip` decorators. `emscripten` and `wasi` are assumed to be compiled with `musl`; otherwise `platform.libc_ver` is checked.

test.support.**check_syntax_error**(*testcase*, *statement*, *errtext=''*, *, *lineno=None*, *offset=None*)

statement 컴파일을 시도하여 *statement*의 구문 에러를 테스트합니다. *testcase*는 테스트를 위한 `unittest` 인스턴스입니다. *errtext*는 발생한 `SyntaxError`의 문자열 표현과 일치해야 하는 정규식입니다. *lineno*가 `None`이 아니면, 예외 줄과 비교합니다. *offset*이 `None`이 아니면, 예외의 오프셋과 비교합니다.

test.support.**open_urlresource**(*url*, **args*, ***kw*)

*url*을 엽니다. 열기에 실패하면, `TestFailed`를 발생시킵니다.

test.support.**reap_children**()

서브 프로세스가 시작될 때마다 `test_main` 끝에 이것을 사용하십시오. 이렇게 하면 여분의 자식(좀비)이 남아서 리소스를 탐하지 않도록 하고 참조 누수를 찾을 때 문제가 발생하지 않도록 할 수 있습니다.

test.support.**get_attribute**(*obj*, *name*)

어트리뷰트를 가져옵니다, `AttributeError`가 발생하면 `unittest.SkipTest`를 발생시킵니다.

test.support.**catch_unraisable_exception**()

`sys.unraisablehook()`를 사용하여 발생시킬 수 없는(unraisable) 예외를 포착하는 컨텍스트 관리자.

예외 값(`cm.unraisable.exc_value`)을 저장하면 참조 순환을 만듭니다. 컨텍스트 관리자가 탈출할 때 참조 순환이 명시적으로 끊어집니다.

객체(`cm.unraisable.object`)를 저장하면 파이널라이즈 중인 객체로 설정되어 있으면 되살릴 수 있습니다. 컨텍스트 관리자를 탈출하면 저장된 객체가 지워집니다.

용법:

```
with support.catch_unraisable_exception() as cm:
    # "발생시킬 수 없는 예외"를 만드는 코드
    ...

    # 발생시킬 수 없는 예외를 확인합니다: cm.unraisable 을 사용하십시오
    ...

# 이 지점에서 cm.unraisable 어트리뷰트는 더는 존재하지 않습니다
# (참조 순환을 끊고자)
```

Added in version 3.8.

test.support.**load_package_tests**(*pkg_dir*, *loader*, *standard_tests*, *pattern*)

테스트 패키지에서 사용하기 위한 `unittest` `load_tests` 프로토콜의 일반 구현. *pkg_dir*는 패키지의 루트 디렉터리입니다; *loader*, *standard_tests* 및 *pattern*은 `load_tests`가 기대하는 인자입니다. 간단한 경우, 테스트 패키지의 `__init__.py`는 다음과 같을 수 있습니다:

```
import os
from test.support import load_package_tests

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__), *args)
```

test.support.**detect_api_mismatch**(*ref_api*, *other_api*, *, *ignore=()*)

*ignore*에 지정된 이 검사에서 무시할 정의된 항목 리스트를 제외하고, *other_api*에서 찾을 수 없는 *ref_api*의 어트리뷰트, 함수 또는 메서드 집합을 반환합니다.

기본적으로 이것은 '_' 로 시작하는 내부(private) 어트리뷰트를 건너뛰지만, 모든 매직 메서드, 즉 '__' 로 시작하고 끝나는 것들을 포함합니다.

Added in version 3.5.

test.support.**patch**(*test_instance*, *object_to_patch*, *attr_name*, *new_value*)

*object_to_patch.attr_name*을 *new_value*로 대체합니다. 또한 *test_instance*에 대한 정리 절차를 추가하여 *object_to_patch*의 *attr_name*을 복원합니다. *attr_name*은 *object_to_patch*의 유효한 어트리뷰트여야 합니다.

test.support.**run_in_subinterp**(*code*)

서브 인터프리터에서 *code*를 실행합니다. `tracemalloc`이 활성화되면 `unittest.SkipTest`를 발생시킵니다.

test.support.**check_free_after_iterating**(*test*, *iter*, *cls*, *args*=())

이터레이션 후 *cls*의 인스턴스가 할당 해제되었음을 어서션 합니다.

test.support.**missing_compiler_executable**(*cmd_names*=[])

*cmd_names*에 이름이 나열된 컴파일러 실행 파일이나 *cmd_names*가 비어있을 때 모든 컴파일러 실행 파일이 있는지 확인하고 누락된 첫 번째 실행 파일을 반환하거나 누락된 것이 없으면 `None`을 반환합니다.

test.support.**check__all__**(*test_case*, *module*, *name_of_module*=*None*, *extra*=(), *not_exported*=())

*module*의 `__all__` 변수에 모든 공용 이름이 포함되어 있는지 어서션 합니다.

모듈의 공용 이름(그것의 API)은 공용 이름 규칙과 일치하고 *module*에 정의되었는지에 따라 자동으로 감지됩니다.

name_of_module 인자는 공용 API로 감지하기 위해 API를 정의 할 수 있는 모듈을 (문자열이나 튜플로) 지정할 수 있습니다. 이에 대한 한 가지 사례는 *module*이 다른 모듈에서 공용 API의 일부를 임포트 할 때입니다, C 백 엔드도 가능합니다 (`csv`와 그것의 `_csv` 처럼).

extra 인자는 적절한 `__module__` 어트리뷰트가 없는 객체처럼, "공용" 으로 자동 감지되지 않는 이름 집합일 수 있습니다. 제공되면, 자동 감지된 항목에 추가됩니다.

not_exported 인자는 이름이 공용처럼 보이더라도 공용 API의 일부로 취급해서는 안 되는 이름 집합일 수 있습니다.

사용 예:

```
import bar
import foo
import unittest
from test import support

class MiscTestCase(unittest.TestCase):
    def test__all__(self):
        support.check__all__(self, foo)

class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        not_exported = {'baz'}  # 문서화되지 않은 이름.
        # bar는 _bar에서 API의 일부를 임포트 합니다.
        support.check__all__(self, bar, ('bar', '_bar'),
                             extra=extra, not_exported=not_exported)
```

Added in version 3.6.

test.support.**skip_if_broken_multiprocessing_synchronize**()

Skip tests if the `multiprocessing.synchronize` module is missing, if there is no available semaphore implementation, or if creating a lock raises an `OSError`.

Added in version 3.10.

test.support.**check_disallow_instantiation**(*test_case*, *tp*, **args*, ***kwds*)

Assert that type *tp* cannot be instantiated using *args* and *kwds*.

Added in version 3.10.

test.support.**adjust_int_max_str_digits**(*max_digits*)

This function returns a context manager that will change the global `sys.set_int_max_str_digits()` setting for the duration of the context to allow execution of test code that needs a different limit on the number of digits when converting between an integer and string.

Added in version 3.11.

`test.support` 모듈은 다음 클래스를 정의합니다:

class test.support.**SuppressCrashReport**

서브 프로세스를 충돌시킬 것으로 예상되는 테스트에서 충돌 대화 상자 팝업을 방지하는 데 사용되는 컨텍스트 관리자.

윈도우에서는, SetErrorMode를 사용하여 윈도우 에러 보고 대화 상자를 비활성화합니다.

유닉스에서는, `resource.setrlimit()`를 사용하여 `resource.RLIMIT_CORE`의 소프트 제한을 0으로 설정하여 코어 덤프 파일 생성을 방지하는 데 사용됩니다.

두 플랫폼 모두에서, 이전 값은 `__exit__()`에 의해 복원됩니다.

class test.support.**SaveSignals**

파이썬 시그널 처리기가 등록한 시그널 처리기를 저장하고 복원하는 클래스.

save(*self*)

Save the signal handlers to a dictionary mapping signal numbers to the current signal handler.

restore(*self*)

Set the signal numbers from the `save()` dictionary to the saved handler.

class test.support.**Matcher**

matches(*self*, *d*, ***kwargs*)

단일 딕셔너리를 제공된 인자와 일치시키려고 합니다.

match_value(*self*, *k*, *dv*, *v*)

단일 저장된 값(*dv*)을 제공된 값(*v*)과 일치시키려고 합니다.

26.10 test.support.socket_helper — 소켓 테스트용 유틸리티

`test.support.socket_helper` 모듈은 소켓 테스트를 지원합니다.

Added in version 3.9.

test.support.socket_helper.**IPV6_ENABLED**

이 호스트에서 IPv6가 활성화되어 있으면 `True`로 설정하고, 그렇지 않으면 `False`로 설정합니다.

test.support.socket_helper.**find_unused_port**(*family=socket.AF_INET*, *socktype=socket.SOCK_STREAM*)

바인딩에 적합해야 하는 미사용 포트를 반환합니다. 이는 `sock` 매개 변수와 같은 패밀리와 유형으로 (기본값은 `AF_INET`, `SOCK_STREAM`) 임시 소켓을 만들고, 포트를 0으로 설정하여 지정된 호스트 주소 (기본값은 `0.0.0.0`)에 바인딩하여, OS에서 사용되지 않은 임시 포트를 도출하여 수행됩니다. 그런 다음 임시 소켓이 닫히고 삭제되고, 임시 포트가 반환됩니다.

이 메서드나 `bind_port()`는 테스트 기간 중 서버 소켓을 특정 포트에 바인딩해야 하는 모든 테스트에 사용해야 합니다. 어떤 것을 사용할 것인지는 호출하는 코드가 파이썬 소켓을 만드는지, 또는 사용되지 않은 포트가 생성자에서 제공되어야 하는지 또는 외부 프로그램에 전달되어야 하는지 (즉 openssl의 s_server 모드에 대한 `-accept` 인자)에 따라 다릅니다. 가능하면 항상 `find_unused_port()`보다 `bind_port()`를 선호하십시오. 하드 코딩된 포트를 사용하면 여러 테스트 인스턴스를 동시에 실행할 수 없게 되어 빌드 봇에 문제가 될 수 있어서 피하는 것이 좋습니다.

test.support.socket_helper.**bind_port**(*sock*, *host=HOST*)

소켓을 사용 가능한 포트에 바인딩하고 포트 번호를 반환합니다. 바인딩 되지 않은 포트를 사용하기 위해 임시 포트에 의존합니다. 이는 특히 빌드 봇 환경에서, 많은 테스트가 동시에 실행될 수 있어서 중요합니다. 이 메서드는 sock.family가 AF_INET이고 sock.type이 SOCK_STREAM이고, 소켓에 SO_REUSEADDR이나 SO_REUSEPORT가 설정되면 예외가 발생합니다. 테스트는 TCP/IP 소켓에 대해 이러한 소켓 옵션을 설정해서는 안 됩니다. 이러한 옵션을 설정하는 유일한 경우는 여러 UDP 소켓을 통해 멀티캐스팅을 테스트하는 것입니다.

또한, SO_EXCLUSIVEADDRUSE 소켓 옵션을 사용할 수 있으면 (즉 윈도우에서), 소켓에 설정됩니다. 이렇게 하면 다른 사람이 테스트 기간 중 우리의 호스트/포트에 바인딩하는 것을 방지할 수 있습니다.

test.support.socket_helper.**bind_unix_socket**(*sock*, *addr*)

유닉스 소켓을 바인드 합니다, PermissionError가 발생하면 unittest.SkipTest를 발생시킵니다.

@test.support.socket_helper.**skip_unless_bind_unix_socket**

유닉스 소켓용 bind() 기능이 필요한 테스트를 실행하기 위한 데코레이터.

test.support.socket_helper.**transient_internet**(*resource_name*, *, *timeout=30.0*, *errnos=()*)

인터넷 연결과 관련된 다양한 문제가 예외로 나타날 때 ResourceDenied를 발생시키는 컨텍스트 관리자.

26.11 test.support.script_helper — 파이썬 실행 테스트용 유틸리티

test.support.script_helper 모듈은 파이썬의 스크립트 실행 테스트를 지원합니다.

test.support.script_helper.**interpreter_requires_environment**()

sys.executable 인터프리터를 실행하기 위해 환경 변수가 필요하면 True를 반환합니다.

assert_python*() 함수를 사용하여 격리 모드(-I)를 시작하거나 환경 없음 모드(-E) 서브 인터프리터 프로세스를 시작해야 하는 테스트를 어노테이트 하는 @unittest.skipIf() 와 함께 사용하도록 설계되었습니다.

정상적인 빌드와 테스트는 이러한 상황을 만나지 않지만, 파이썬의 현재 홈 찾기 논리로 명확한 홈이 없는 인터프리터에서 표준 라이브러리 테스트 스위트를 실행하려고 할 때 발생할 수 있습니다.

PYTHONHOME 설정은 이러한 상황에서 대부분의 테스트 스위트를 실행하는 한 가지 방법입니다. PYTHONPATH나 PYTHONUSERSITE는 인터프리터가 시작될 수 있는지에 영향을 줄 수 있는 다른 공통 환경 변수입니다.

test.support.script_helper.**run_python_until_end**(**args*, ***env_vars*)

서브 프로세스에서 인터프리터를 실행하기 위해 *env_vars* 기반 환경을 설정합니다. 값에는 __isolated, __cleanenv, __cwd 및 TERM이 포함될 수 있습니다.

버전 3.9에서 변경: 이 함수는 더는 *stderr*에서 공백을 제거하지 않습니다.

test.support.script_helper.**assert_python_ok**(**args*, ***env_vars*)

*args*와 선택적 환경 변수 *env_vars*를 사용하여 인터프리터를 실행하면 성공(rc == 0)하고 (return code, stdout, stderr) 튜플을 반환함을 어서션 합니다.

__cleanenv 키워드 전용 매개변수가 설정되면, *env_vars*가 새로운 환경으로 사용됩니다.

파이썬은 *__isolated* 키워드 전용 매개변수가 False로 설정된 경우를 제외하고, 격리 모드(명령 줄 옵션 -I)에서 시작됩니다.

버전 3.9에서 변경: 이 함수는 더는 *stderr*에서 공백을 제거하지 않습니다.

test.support.script_helper.**assert_python_failure**(**args*, ***env_vars*)

*args*와 선택적 환경 변수 *env_vars*를 사용하여 인터프리터 실행하면 실패(rc != 0)하고 (return code, stdout, stderr) 튜플을 반환함을 어서션 합니다.

추가 옵션은 assert_python_ok()를 참조하십시오.

버전 3.9에서 변경: 이 함수는 더는 *stderr*에서 공백을 제거하지 않습니다.

test.support.script_helper.**spawn_python**(**args, stdout=subprocess.PIPE, stderr=subprocess.STDOUT, **kw*)

주어진 인자로 파이썬 서브 프로세스를 실행합니다.

*kw*는 `subprocess.Popen()`에 전달할 추가 키워드 인자입니다. `subprocess.Popen` 객체를 반환합니다.

test.support.script_helper.**kill_python**(*p*)

완료될 때까지 주어진 `subprocess.Popen` 프로세스를 실행하고 stdout을 반환합니다.

test.support.script_helper.**make_script**(*script_dir, script_basename, source, omit_suffix=False*)

*script_dir*과 *script_basename* 경로에 *source*를 포함하는 스크립트를 만듭니다. *omit_suffix*가 `False`이면, 이름에 `.py`를 추가합니다. 전체 스크립트 경로를 반환합니다.

test.support.script_helper.**make_zip_script**(*zip_dir, zip_basename, script_name, name_in_zip=None*)

*script_name*의 파일을 포함하는 확장자가 `zip`인 zip 파일을 *zip_dir*과 *zip_basename*에 만듭니다. *name_in_zip*은 아카이브 이름입니다. `(full path, full path of archive name)`을 포함하는 튜플을 반환합니다.

test.support.script_helper.**make_pkg**(*pkg_dir, init_source=''*)

*init_source*를 내용으로 하는 `__init__` 파일을 포함하는 *pkg_dir*이라는 디렉터리를 만듭니다.

test.support.script_helper.**make_zip_pkg**(*zip_dir, zip_basename, pkg_name, script_basename, source, depth=1, compiled=False*)

빈 `__init__` 파일과 *source*를 포함하는 파일 *script_basename*을 포함하는 zip 패키지 디렉터리를 *zip_dir*과 *zip_basename* 경로로 만듭니다. *compiled*가 `True`이면, 두 소스 파일이 모두 컴파일되어 zip 패키지에 추가됩니다. 전체 zip 경로와 zip 파일의 아카이브 이름의 튜플을 반환합니다.

26.12 test.support.bytecode_helper — 올바른 바이트 코드 생성 테스트를 위한 지원 도구

`test.support.bytecode_helper` 모듈은 바이트 코드 생성 테스트와 검사를 지원합니다.

Added in version 3.9.

모듈은 다음 클래스를 정의합니다:

class test.support.bytecode_helper.**BytecodeTestCase**(*unittest.TestCase*)

이 클래스에는 바이트 코드를 검사하기 위한 사용자 정의 어서션 메서드가 있습니다.

BytecodeTestCase.**get_disassembly_as_string**(*co*)

*co*의 역 어셈블리를 문자열로 반환합니다.

BytecodeTestCase.**assertInBytecode**(*x, opname, argval=_UNSPECIFIED*)

*opname*이 발견되면 명령어를 반환하고, 그렇지 않으면 `AssertionError`를 던집니다.

BytecodeTestCase.**assertNotInBytecode**(*x, opname, argval=_UNSPECIFIED*)

*opname*이 발견되면 `AssertionError`를 던집니다.

26.13 test.support.threading_helper — threading 테스트용 유틸리티

`test.support.threading_helper` 모듈은 threading 테스트를 지원합니다.

Added in version 3.10.

test.support.threading_helper.**join_thread**(*thread, timeout=None*)

timeout 내에 *thread*에 조인(join)합니다. 스레드가 *timeout* 초 후에도 여전히 살아 있으면 `AssertionError`를 발생시킵니다.

@test.support.threading_helper.**reap_threads**

테스트가 실패하더라도 스레드를 정리하는 데코레이터.

test.support.threading_helper.**start_threads**(*threads*, *unlock=None*)

Context manager to start *threads*, which is a sequence of threads. *unlock* is a function called after the threads are started, even if an exception was raised; an example would be `threading.Event.set()`. `start_threads` will attempt to join the started threads upon exit.

test.support.threading_helper.**threading_cleanup**(**original_values*)

*original_values*에 지정되지 않은 스레드를 정리합니다. 테스트가 백그라운드에서 실행 중인 스레드를 남겨두면 경고를 내도록 설계되었습니다.

test.support.threading_helper.**threading_setup**()

현재 스레드 수와 매달린(dangling) 스레드의 복사본을 반환합니다.

test.support.threading_helper.**wait_threads_exit**(*timeout=None*)

`with` 문에서 만들어진 모든 스레드가 종료할 때까지 대기하는 컨텍스트 관리자.

test.support.threading_helper.**catch_threading_exception**()

`threading.excepthook()`을 사용하여 `threading.Thread` 예외를 포착하는 컨텍스트 관리자.

예외가 포착될 때 설정되는 어트리뷰트:

- `exc_type`
- `exc_value`
- `exc_traceback`
- `thread`

`threading.excepthook()` 설명서를 참조하십시오.

이러한 어트리뷰트들은 컨텍스트 관리자 탈출 시에 삭제됩니다.

용법:

```
with threading_helper.catch_threading_exception() as cm:
    # 예외를 발생시키는 스레드를 생성하는 코드
    ...

    # 스레드 예외를 확인합니다, cm 어트리뷰트를 사용하십시오:
    # exc_type, exc_value, exc_traceback, thread
    ...

# 이 지점에서 cm 의 exc_type, exc_value, exc_traceback, thread 어트리뷰트는
# 더는 존재하지 않습니다
# (참조 순환을 피하고자)
```

Added in version 3.8.

test.support.threading_helper.**run_concurrently**(*worker_func*, *nthreads*, *args=()*, *kwargs={}*)

Run the worker function concurrently in multiple threads. Re-raises an exception if any thread raises one, after all threads have finished.

26.14 test.support.os_helper — os 테스트용 유틸리티

`test.support.os_helper` 모듈은 os 테스트를 지원합니다.

Added in version 3.10.

test.support.os_helper.**FS_NONASCII**

`os.fsencode()`로 인코딩 할 수 있는 비 ASCII 문자.

test.support.os_helper.**SAVEDCWD**

os.getcwd()로 설정합니다.

test.support.os_helper.**TESTFN**

임시 파일의 이름으로 사용하기에 안전한 이름으로 설정합니다. 만들어진 모든 임시 파일은 닫히고 언 링크(제거) 되어야 합니다.

test.support.os_helper.**TESTFN_NONASCII**

Set to a filename containing the FS_NONASCII character, if it exists. This guarantees that if the filename exists, it can be encoded and decoded with the default filesystem encoding. This allows tests that require a non-ASCII filename to be easily skipped on platforms where they can' t work.

test.support.os_helper.**TESTFN_UNENCODABLE**

엄격(strict) 모드에서 파일 시스템 인코딩으로 인코딩할 수 없는 파일명(str 형)으로 설정합니다. 이러한 파일명을 생성할 수 없으면 None일 수 있습니다.

test.support.os_helper.**TESTFN_UNDECODABLE**

엄격(strict) 모드에서 파일 시스템 인코딩으로 디코딩할 수 없는 파일명(bytes 형)으로 설정합니다. 이러한 파일명을 생성할 수 없으면 None일 수 있습니다.

test.support.os_helper.**TESTFN_UNICODE**

임시 파일을 위한 비 ASCII 이름으로 설정합니다.

class test.support.os_helper.**EnvironmentVarGuard**

환경 변수를 임시로 설정하거나 설정 해제하는 데 사용되는 클래스. 인스턴스는 컨텍스트 관리자로 사용할 수 있으며 하부 os.environ을 조회/수정하기 위한 완전한 딕셔너리 인터페이스를 가질 수 있습니다. 컨텍스트 관리자를 탈출하면 이 인스턴스를 통해 수행된 환경 변수에 대한 모든 변경 사항이 되돌려집니다.

버전 3.1에서 변경: 딕셔너리 인터페이스가 추가되었습니다.

class test.support.os_helper.**FakePath**(*path*)

단순 경로류 객체. 단지 *path* 인자를 반환하는 __fspath__() 메서드를 구현합니다. *path*가 예외이면, __fspath__() 에서 발생시킵니다.

EnvironmentVarGuard.**set**(*envvar*, *value*)

일시적으로 환경 변수 envvar를 value 값으로 설정합니다.

EnvironmentVarGuard.**unset**(*envvar*, **others*)

Temporarily unset one or more environment variables.

버전 3.14에서 변경: More than one environment variable can be unset.

test.support.os_helper.**can_symlink**()

OS가 심볼릭 링크를 지원하면 True를, 그렇지 않으면 False를 반환합니다.

test.support.os_helper.**can_xattr**()

OS가 xattr을 지원하면 True를, 그렇지 않으면 False를 반환합니다.

test.support.os_helper.**change_cwd**(*path*, *quiet=False*)

현재 작업 디렉터리를 *path*로 일시적으로 변경하고 그 디렉터리를 산출하는 컨텍스트 관리자.

*quiet*가 False이면, 컨텍스트 관리자는 에러 시 예외를 발생시킵니다. 그렇지 않으면, 경고만 발행하고 현재 작업 디렉터리를 같게 유지합니다.

test.support.os_helper.**create_empty_file**(*filename*)

*filename*으로 빈 파일을 만듭니다. 이미 있으면, 자릅니다.

test.support.os_helper.**fd_count**()

열린 파일 기술자의 수를 셉니다.

test.support.os_helper.**fs_is_case_insensitive**(*directory*)

*directory*의 파일 시스템이 대소 문자를 구분하지 않으면 True를 반환합니다.

test.support.os_helper.**make_bad_fd**()

임시 파일을 여닫아서 잘못된 파일 기술자를 만든 다음, 그 기술자를 반환합니다.

test.support.os_helper.**rmdir**(*filename*)

*filename*에 대해 os.rmdir()을 호출합니다. 윈도우 플랫폼에서는, 파일의 존재를 확인하는 대기 루프로 래핑 됩니다. 이는 파일을 열려있는 상태로 유지하고 삭제를 막을 수 있는 안티바이러스 프로그램 때문에 필요합니다.

test.support.os_helper.**rmtree**(*path*)

*path*에 대해 shutil.rmtree()를 호출하거나 os.lstat()과 os.rmdir()을 호출하여 경로와 해당 내용을 제거합니다. rmdir()과 마찬가지로, 윈도우 플랫폼에서, 이것은 파일의 존재를 확인하는 대기 루프로 래핑 됩니다.

@test.support.os_helper.**skip_unless_symlink**

심볼릭 링크 지원이 필요한 테스트를 실행하기 위한 데코레이터.

@test.support.os_helper.**skip_unless_xattr**

xattr 지원이 필요한 테스트를 실행하기 위한 데코레이터.

test.support.os_helper.**temp_cwd**(*name='tempcwd', quiet=False*)

임시로 새 디렉터리를 만들고 현재 작업 디렉터리(CWD)를 변경하는 컨텍스트 관리자.

컨텍스트 관리자는 현재 작업 디렉터리를 임시로 변경하기 전에 이름이 *name*인 임시 디렉터리를 현재 디렉터리에 만듭니다. *name*이 None이면, 임시 디렉터리는 tempfile.mkdtemp()를 사용하여 만들어집니다.

*quiet*가 False이고 만들 수 없거나 CWD를 변경할 수 없으면, 에러가 발생합니다. 그렇지 않으면, 경고만 발생하고 원래 CWD가 사용됩니다.

test.support.os_helper.**temp_dir**(*path=None, quiet=False*)

*path*에 임시 디렉터리를 만들고 그 디렉터리를 산출하는 컨텍스트 관리자.

*path*가 None이면, 임시 디렉터리는 tempfile.mkdtemp()를 사용하여 만들어집니다. *quiet*가 False이면, 컨텍스트 관리자는 에러 시 예외를 발생시킵니다. 그렇지 않으면, *path*가 지정되고 만들 수 없으면, 경고만 발행됩니다.

test.support.os_helper.**temp_umask**(*umask*)

프로세스 umask를 임시로 설정하는 컨텍스트 관리자.

test.support.os_helper.**unlink**(*filename*)

*filename*에 대해 os.unlink()를 호출합니다. rmdir()과 마찬가지로, 윈도우 플랫폼에서는, 파일의 존재를 확인하는 대기 루프로 래핑 됩니다.

26.15 test.support.import_helper — 임포트 테스트용 유틸리티

test.support.import_helper 모듈은 임포트 테스트를 지원합니다.

Added in version 3.10.

test.support.import_helper.**forget**(*module_name*)

sys.modules에서 *module_name*이라는 모듈을 제거하고 모듈의 바이트 컴파일된 파일을 삭제합니다.

test.support.import_helper.**import_fresh_module**(*name, fresh=(), blocked=(), deprecated=False*)

이 함수는 임포트 전에 sys.modules에서 명명된 모듈을 제거하여 명명된 파이썬 모듈의 새 복사본을 임포트하고 반환합니다. reload()와 달리, 원래 모듈은 이 연산의 영향을 받지 않습니다.

*fresh*는 임포트를 수행하기 전에 sys.modules 캐시에서 함께 제거되는 추가 모듈 이름의 이터러블입니다.

*blocked*는 임포트 하는 동안 모듈 캐시에서 None으로 대체되는 모듈 이름의 이터러블로, 임포트 하려고 시도하면 ImportError가 발생하도록 합니다.

명명된 모듈과 *fresh*와 *blocked* 매개 변수에 명명된 모든 모듈은 임포트를 시작하기 전에 보관되고 새 임포트가 완료되면 `sys.modules`에 다시 삽입됩니다.

*deprecated*가 `True`이면 이 임포트 중에 모듈과 패키지 폐지 메시지가 억제됩니다.

이 함수는 명명된 모듈을 임포트 할 수 없으면 `ImportError`를 발생시킵니다.

사용 예:

```
# 나머지 테스트 스위트에서 사용 중인 버전에 영향을 주지 않고 테스트용 경고 모듈의 사본을
# 얻습니다. 한 사본은 C 구현을 사용하고, 다른 사본은 순수한 파이썬 대체 구현을 사용하도록
# 강요됩니다.
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

Added in version 3.1.

`test.support.import_helper.`**`import_module`**(*name*, *deprecated=False*, *, *required_on=()*)

이 함수는 명명된 모듈을 임포트하고 반환합니다. 일반 임포트와 달리, 이 함수는 모듈을 임포트할 수 없으면 `unittest.SkipTest`를 발생시킵니다.

*deprecated*가 `True`이면 이 임포트 중에 모듈과 패키지 폐지 메시지가 억제됩니다. 모듈이 한 플랫폼에서는 필수지만, 다른 곳에서는 선택적이면, *required_on*을 `sys.platform`과 비교할 플랫폼 접두사의 이터러블로 설정합니다.

Added in version 3.1.

`test.support.import_helper.`**`modules_setup`**()

`sys.modules`의 복사본을 반환합니다.

`test.support.import_helper.`**`modules_cleanup`**(*oldmodules*)

내부 캐시를 보존하기 위해 *oldmodules*와 `encodings`를 제외한 모듈들을 제거합니다.

`test.support.import_helper.`**`unload`**(*name*)

`sys.modules`에서 *name*을 삭제합니다.

`test.support.import_helper.`**`make_legacy_pyc`**(*source*)

PEP 3147/PEP 488 pyc 파일을 레거시 pyc 위치로 옮기고 레거시 pyc 파일에 대한 파일 시스템 경로를 반환합니다. *source* 값은 소스 파일에 대한 파일 시스템 경로입니다. 반드시 존재할 필요는 없지만, PEP 3147/488 pyc 파일이 있어야 합니다.

class `test.support.import_helper.`**`CleanImport`**(**module_names*)

새 모듈 참조를 반환하도록 임포트를 강제하는 컨텍스트 관리자. 이는 임포트 시 `DeprecationWarning`의 방출과 같은 모듈 수준 동작을 테스트하는 데 유용합니다. 사용 예:

```
with CleanImport('foo'):
    importlib.import_module('foo')  # 새 참조.
```

class `test.support.import_helper.`**`DirsOnSysPath`**(**paths*)

`sys.path`에 디렉터리를 임시로 추가하는 컨텍스트 관리자.

이렇게 하면 `sys.path`의 복사본이 만들어지고, 위치 인자로 지정된 모든 디렉터리를 추가한 다음, 컨텍스트가 끝나면 `sys.path`를 복사된 설정으로 되돌립니다.

객체 교체를 포함하여, 컨텍스트 관리자 본문의 모든 `sys.path` 수정 사항은 블록 끝에서 되돌려짐에 유의하십시오.

26.16 test.support.warnings_helper — warnings 테스트용 유틸리티

test.support.warnings_helper 모듈은 warnings 테스트를 지원합니다.

Added in version 3.10.

test.support.warnings_helper.**ignore_warnings**(*, *category*)

Suppress warnings that are instances of *category*, which must be *Warning* or a subclass. Roughly equivalent to *warnings.catch_warnings()* with *warnings.simplefilter('ignore', category=category)*. For example:

```
@warning_helper.ignore_warnings(category=DeprecationWarning)
def test_suppress_warning():
    # 뭔가 합니다
```

Added in version 3.8.

test.support.warnings_helper.**check_no_resource_warning**(*testcase*)

*ResourceWarning*이 발생하지 않았는지 확인하는 컨텍스트 관리자. 컨텍스트 관리자가 끝나기 전에 *ResourceWarning*을 방출할 수 있는 객체를 제거해야 합니다.

test.support.warnings_helper.**check_syntax_warning**(*testcase*, *statement*, *errtext=''*, *, *lineno=1*, *offset=None*)

statement 컴파일을 시도하여 *statement*에서 구문 경고를 테스트합니다. *SyntaxWarning*이 한 번만 방출되고, 에러로 바꿀 때 *SyntaxError*로 변환되는지도 테스트합니다. *testcase*는 테스트를 위한 *unittest* 인스턴스입니다. *errtext*는 방출된 *SyntaxWarning*과 발생한 *SyntaxError*의 문자열 표현과 일치해야 하는 정규식입니다. *lineno*가 `None`이 아니면 경고와 예외 줄과 비교합니다. *offset*이 `None`이 아니면, 예외 오프셋과 비교합니다.

Added in version 3.8.

test.support.warnings_helper.**check_warnings**(**filters*, *quiet=True*)

경고가 올바르게 발생했는지 테스트하기 쉽게 하는 *warnings.catch_warnings()* 용 편의 래퍼. *warnings.simplefilter()*를 `always`로 설정하고 기록된 결과를 자동으로 검증하는 옵션을 사용하여 `warnings.catch_warnings(record=True)`를 호출하는 것과 거의 동등합니다.

`check_warnings`는 위치 인자로 `("message regexp", WarningCategory)` 형식의 2-튜플을 받습니다. 하나 이상의 *filters*가 제공되거나, 선택적 키워드 인자 *quiet*가 `False`이면, 경고가 예상대로인지 확인합니다: 지정된 각 필터는 둘러싸인 코드에서 발생한 경고 중 적어도 하나와 일치해야 합니다. 그렇지 않으면 테스트가 실패합니다. 지정된 필터와 일치하지 않는 경고가 발생하면 테스트가 실패합니다. 첫 번째 검사를 비활성화하려면, *quiet*를 `True`로 설정합니다.

인자가 지정되지 않으면, 기본값은 다음과 같습니다:

```
check_warnings(("", Warning), quiet=True)
```

이 경우 모든 경고가 포착되고 에러가 발생하지 않습니다.

컨텍스트 관리자에 진입하면, `WarningRecorder` 인스턴스가 반환됩니다. *catch_warnings()*의 하부 경고 리스트는 레코더 객체의 *warnings* 어트리뷰트를 통해 사용할 수 있습니다. 편의상, 가장 최근의 경고를 나타내는 객체의 어트리뷰트는 레코더 객체를 통해 직접 액세스 할 수도 있습니다 (아래 예를 참조하십시오). 경고가 발생하지 않으면, 객체에서 예상되는 경고를 나타내는 어트리뷰트는 `None`을 반환합니다.

레코더 객체에는 경고 리스트를 지우는 `reset()` 메서드도 있습니다.

컨텍스트 관리자는 다음과 같이 사용되도록 설계되었습니다:

```
with check_warnings(("assertion is always true", SyntaxWarning),
                    ("", UserWarning)):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

이 경우 경고가 발생하지 않았거나, 다른 경고가 발생하면, *check_warnings()*는 에러를 발생시킵니다.

테스트에서 경고가 발생했는지를 확인하는 것만이 아니라, 경고를 더 깊이 조사해야 할 때, 다음과 같은 코드를 사용할 수 있습니다:

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

여기에서 모든 경고가 포착되고, 테스트 코드는 포착된 경고를 직접 테스트합니다.

버전 3.2에서 변경: 새로운 선택적 인자 *filters*와 *quiet*.

class test.support.warnings_helper.**WarningsRecorder**

단위 테스트에 대한 경고를 기록하는 데 사용되는 클래스. 자세한 내용은 위의 *check_warnings()* 설명서를 참조하십시오.

CHAPTER 27

디버깅과 프로파일링

이 라이브러리들은 파이썬 개발을 돕습니다: 디버거를 사용하면 코드를 단계별로 실행하고, 스택 프레임을 분석하고, 중단 점을 설정할 수 있으며, 프로파일러는 코드를 실행하고, 프로그램의 병목 지점을 식별할 수 있도록 실행 시간을 자세하게 분석합니다. 감사 이벤트는 이것이 없다면 침입적인 디버깅이나 패치가 필요한 실행 시간 동작에 대한 가시성을 제공합니다.

27.1 감사 이벤트 표

이 표는 CPython 런타임과 표준 라이브러리에서 *sys.audit()* 나 `PySys_Audit()` 호출에 의해 발생하는 모든 이벤트를 포함합니다. 이 호출은 3.8 이상에서 추가되었습니다 (**PEP 578**을 참조하십시오).

이 이벤트 처리에 대한 정보는 *sys.addaudithook()* 과 `PySys_AddAuditHook()` 을 참조하십시오.

이 표는 CPython 설명서로부터 생성되며, 다른 구현에서 발생하는 이벤트를 나타내지 않을 수 있습니다. 실제 발생한 이벤트에 대해서는 여러분의 런타임 관련 설명서를 참조하십시오.

Audit event	Arguments
_thread.start_new_thread	`function`, `args`, `kwargs`
array.__new__	`typecode`, `initializer`
builtins.breakpoint	`breakpointhook`
builtins.id	`id`
builtins.input	`prompt`
builtins.input/result	`result`
code.__new__	`code`, `filename`, `name`, `argcount`, `posonlyargcount`, `kwonlyargcount`, `nlocals`, `st`
compile	`source`, `filename`
cpython.PyConfig_Set	`name`, `value`
cpython.PyInterpreterState_Clear	
cpython.PyInterpreterState_New	
cpython._PySys_ClearAuditHooks	
cpython.remote_debugger_script	`script_path`
cpython.run_command	`command`
cpython.run_file	`filename`
cpython.run_interactivehook	`hook`
cpython.run_module	`module-name`
cpython.run_startup	`filename`
cpython.run_stdin	

표 1 – 이전 페이지에서 계속

Audit event	Arguments
ctypes.addressof	obj
ctypes.call_function	func_pointer, arguments
ctypes.cdata	address
ctypes.cdata/buffer	pointer, size, offset
ctypes.create_string_buffer	init, size
ctypes.create_unicode_buffer	init, size
ctypes.dlopen	name
ctypes.dlsym	library, name
ctypes.dlsym/handle	handle, name
ctypes.get_errno	
ctypes.get_last_error	
ctypes.memoryview_at	address, size, readonly
ctypes.set_errno	errno
ctypes.set_exception	code
ctypes.set_last_error	error
ctypes.string_at	ptr, size
ctypes.wstring_at	ptr, size
ensurepip.bootstrap	root
exec	code_object
fcntl.fcntl	fd, cmd, arg
fcntl.flock	fd, operation
fcntl.ioctl	fd, request, arg
fcntl.lockf	fd, cmd, len, start, whence
ftplib.connect	self, host, port
ftplib.sendcmd	self, cmd
function.__new__	code
gc.get_objects	generation
gc.get_referents	objs
gc.get_referrers	objs
glob.glob	pathname, recursive
glob.glob/2	pathname, recursive, root_dir, dir_fd
http.client.connect	self, host, port
http.client.send	self, data
imaplib.open	self, host, port
imaplib.send	self, data
import	module, filename, sys.path, sys.meta_path, sys.path_hooks
marshal.dumps	value, version
marshal.load	
marshal.loads	bytes
mmap.__new__	fileno, length, access, offset
msvcrt.get_osfhandle	fd
msvcrt.locking	fd, mode, nbytes
msvcrt.open_osfhandle	handle, flags
object.__delattr__	obj, name
object.__getattr__	obj, name
object.__setattr__	obj, name, value
open	path, mode, flags
os.add_dll_directory	path
os.chdir	path
os.chflags	path, flags
os.chmod	path, mode, dir_fd
os.chown	path, uid, gid, dir_fd
os.exec	path, args, env
os.fork	
os.forkpty	

표 1 – 이전 페이지에서 계속

Audit event	Arguments
os.fwalk	top, topdown, onerror, follow_symlinks, dir_fd
os.getxattr	path, attribute
os.kill	pid, sig
os.killpg	pgid, sig
os.link	src, dst, src_dir_fd, dst_dir_fd
os.listdir	path
os.listdrives	
os.listmounts	volume
os.listvolumes	
os.listxattr	path
os.lockf	fd, cmd, len
os.mkdir	path, mode, dir_fd
os.posix_spawn	path, argv, env
os.putenv	key, value
os.remove	path, dir_fd
os.removexattr	path, attribute
os.rename	src, dst, src_dir_fd, dst_dir_fd
os.rmdir	path, dir_fd
os.scandir	path
os.setxattr	path, attribute, value, flags
os.spawn	mode, path, args, env
os.startfile	path, operation
os.startfile/2	path, operation, arguments, cwd, show_cmd
os.symlink	src, dst, dir_fd
os.system	command
os.truncate	fd, length
os.unsetenv	key
os.utime	path, times, ns, dir_fd
os.walk	top, topdown, onerror, followlinks
pathlib.Path.glob	self, pattern
pathlib.Path.rglob	self, pattern
pdb.Pdb	
pickle.find_class	module, name
poplib.connect	self, host, port
poplib.putline	self, line
pty.spawn	argv
resource.prlimit	pid, resource, limits
resource.setrlimit	resource, limits
setopencodehook	
shutil.chown	path, user, group
shutil.copyfile	src, dst
shutil.copymode	src, dst
shutil.copystat	src, dst
shutil.copytree	src, dst
shutil.make_archive	base_name, format, root_dir, base_dir
shutil.move	src, dst
shutil.rmtree	path, dir_fd
shutil.unpack_archive	filename, extract_dir, format
signal.pthread_kill	thread_id, signalnum
smtplib.connect	self, host, port
smtplib.send	self, data
socket.__new__	self, family, type, protocol
socket.bind	self, address
socket.connect	self, address
socket.getaddrinfo	host, port, family, type, protocol

표 1 – 이전 페이지에서 계속

Audit event	Arguments
socket.gethostbyaddr	ip_address
socket.gethostbyname	hostname
socket.gethostname	
socket.getnameinfo	sockaddr
socket.getservbyname	servicename, protocolname
socket.getservbyport	port, protocolname
socket.sendmsg	self, address
socket.sendto	self, address
socket.sethostname	name
sqlite3.connect	database
sqlite3.connect/handle	connection_handle
sqlite3.enable_load_extension	connection, enabled
sqlite3.load_extension	connection, path
subprocess.Popen	executable, args, cwd, env
sys._current_exceptions	
sys._current_frames	
sys._getframe	frame
sys._getframemodulename	depth
sys.addaudithook	
sys.excepthook	hook, type, value, traceback
sys.monitoring.register_callback	func
sys.remote_exec	pid
sys.set_asyncgen_hooks_finalizer	
sys.set_asyncgen_hooks_firstiter	
sys.setprofile	
sys.settrace	
sys.unraisablehook	hook, unraisable
syslog.closelog	
syslog.openlog	ident, logoption, facility
syslog.setlogmask	maskpri
syslog.syslog	priority, message
tempfile.mkdtemp	fullpath
tempfile.mkstemp	fullpath
time.sleep	secs
urllib.Request	fullurl, data, headers, method
webbrowser.open	url
winreg.ConnectRegistry	computer_name, key
winreg.CreateKey	key, sub_key, access
winreg.DeleteKey	key, sub_key, access
winreg.DeleteValue	key, value
winreg.DisableReflectionKey	key
winreg.EnableReflectionKey	key
winreg.EnumKey	key, index
winreg.EnumValue	key, index
winreg.ExpandEnvironmentStrings	str
winreg.LoadKey	key, sub_key, file_name
winreg.OpenKey	key, sub_key, access
winreg.OpenKey/result	key
winreg.PyHKEY.Detach	key
winreg.QueryInfoKey	key
winreg.QueryReflectionKey	key
winreg.QueryValue	key, sub_key, value_name
winreg.SaveKey	key, file_name
winreg.SetValue	key, sub_key, type, value

다음 이벤트는 내부적으로 발생하며 어떤 CPython의 공개 API에도 해당하지 않습니다:

감사 이벤트	인자
_winapi.CreateFile	file_name, desired_access, share_mode, creation_disposition, flags_and_attributes
_winapi.CreateJunction	src_path, dst_path
_winapi.CreateNamedPi	name, open_mode, pipe_mode
_winapi.CreatePipe	
_winapi.CreateProcess	application_name, command_line, current_directory
_winapi.OpenProcess	process_id, desired_access
_winapi.TerminateProce	handle, exit_code
_posixsubpro-cess.fork_exec	exec_list, args, env
ctypes.PyObj_FromPtr	obj

Added in version 3.14: The `_posixsubprocess.fork_exec` internal audit event.

27.2 bdb — 디버거 프레임워크

소스 코드: Lib/bdb.py

---

bdb 모듈은 중단점 설정이나 디버거를 통한 실행 관리와 같은 기본 디버거 기능을 처리합니다.

다음 예외가 정의됩니다:

exception bdb.**BdbQuit**

디버거를 종료하기 위해 *Bdb* 클래스가 발생시키는 예외.

bdb 모듈은 또한 두 가지 클래스를 정의합니다:

class bdb.**Breakpoint**(*self*, *file*, *line*, *temporary=False*, *cond=None*, *funcname=None*)

이 클래스는 임시 중단점, 무시 카운트, 비활성화와 (재) 활성화 및 조건을 구현합니다.

중단점은 *bpbynumber*라는 리스트를 통해 번호로, *bplist*를 통해 `(file, line)` 쌍으로 인덱싱됩니다. 전자는 *Breakpoint* 클래스의 단일 인스턴스를 가리킵니다. 후자는 줄당 하나 이상의 중단점이 있을 수 있어서 이러한 인스턴스의 리스트를 가리킵니다.

중단점을 만들 때, 연관된 *파일명*은 규범적 형식(canonical form)이어야 합니다. *funcname*이 정의되면, 해당 함수의 첫 번째 줄이 실행될 때 중단점 *적중*이 카운트됩니다. *조건부* 중단점은 항상 *적중*을 카운트합니다.

Breakpoint 인스턴스에는 다음과 같은 메서드가 있습니다:

deleteMe()

file/line과 관련된 리스트에서 중단점을 삭제합니다. 해당 위치의 마지막 중단점이면, file/line의 항목도 삭제합니다.

enable()

중단점을 활성화된 것으로 표시합니다.

disable()

중단점을 비활성화된 것으로 표시합니다.

bpformat()

멋지게 포맷된, 중단점에 대한 모든 정보가 포함된 문자열을 반환합니다:

- 중단점 번호.
- Temporary status (del or keep).

- File/line 위치.
- Break condition.
- Number of times to ignore.
- Number of times hit.

Added in version 3.2.

bpprint(*out=None*)

`bpformat()`의 출력을 파일 *out*, 또는 `None`이면 표준 출력으로 인쇄합니다.

`Breakpoint` 인스턴스에는 다음과 같은 어트리뷰트가 있습니다:

file

File name of the `Breakpoint`.

line

Line number of the `Breakpoint` within `file`.

temporary

`True` if a `Breakpoint` at (file, line) is temporary.

cond

Condition for evaluating a `Breakpoint` at (file, line).

funcname

Function name that defines whether a `Breakpoint` is hit upon entering the function.

enabled

`True` if `Breakpoint` is enabled.

bpbynumber

Numeric index for a single instance of a `Breakpoint`.

bplist

Dictionary of `Breakpoint` instances indexed by (`file`, `line`) tuples.

ignore

Number of times to ignore a `Breakpoint`.

hits

Count of the number of times a `Breakpoint` has been hit.

class bdb.**Bdb**(*skip=None, backend='settrace'*)

`Bdb` 클래스는 범용 파이썬 디버거 베이스 클래스 역할을 합니다.

이 클래스는 추적 기능의 세부 사항을 처리합니다; 파생 클래스는 사용자 상호 작용을 구현해야 합니다. 표준 디버거 클래스 (`pdb.Pdb`)가 예입니다.

skip 인자는, 주어지면, glob 스타일 모듈 이름 패턴의 이터러블 이어야 합니다. 디버거는 이러한 패턴 중 하나와 일치하는 모듈에서 시작되는 프레임으로 들어가지 않습니다. 프레임을 특정 모듈에서 시작된 것으로 간주하는지는 프레임 전역의 `__name__`에 의해 결정됩니다.

The *backend* argument specifies the backend to use for `Bdb`. It can be either `'settrace'` or `'monitoring'`. `'settrace'` uses `sys.settrace()` which has the best backward compatibility. The `'monitoring'` backend uses the new `sys.monitoring` that was introduced in Python 3.12, which can be much more efficient because it can disable unused events. We are trying to keep the exact interfaces for both backends, but there are some differences. The debugger developers are encouraged to use the `'monitoring'` backend to achieve better performance.

버전 3.1에서 변경: *skip* 매개 변수를 추가했습니다.

버전 3.14에서 변경: Added the *backend* parameter.

`Bdb`의 다음 메서드는 일반적으로 재정의할 필요가 없습니다.

canonic(*filename*)

*filename*의 규범적 형식을 반환합니다.

For real file names, the canonical form is an operating-system-dependent, *case-normalized absolute path*. A *filename* with angle brackets, such as "<stdin>" generated in interactive mode, is returned unchanged.

start_trace(*self*)

Start tracing. For 'settrace' backend, this method is equivalent to sys.settrace(self.trace_dispatch)

Added in version 3.14.

stop_trace(*self*)

Stop tracing. For 'settrace' backend, this method is equivalent to sys.settrace(None)

Added in version 3.14.

reset()

디버깅을 시작할 준비가 된 값으로 botframe, stopframe, returnframe 및 *quitting* 어트리뷰트를 설정합니다.

trace_dispatch(*frame*, *event*, *arg*)

이 함수는 디버그되는 프레임의 추적 함수(trace function)로 설치됩니다. 반환 값은 새로운 추적 함수(대부분 자체)입니다.

기본 구현은 실행되려고 하는 (문자열로 전달된) 이벤트의 유형에 따라 프레임을 디스패치 하는 방법을 결정합니다. *event*는 다음 중 하나일 수 있습니다:

- "line": 새로운 코드 줄이 실행되려고 합니다.
- "call": 함수가 호출되거나, 다른 코드 블록에 진입하려고 합니다.
- "return": 함수나 다른 코드 블록이 반환하려고 합니다.
- "exception": 예외가 발생했습니다.
- "c_call": C 함수가 호출되려고 합니다.
- "c_return": C 함수가 반환했습니다.
- "c_exception": C 함수가 예외를 발생시켰습니다.

파이썬 이벤트의 경우, 특수 함수(아래를 참조하세요)가 호출됩니다. C 이벤트의 경우, 아무런 액션도 취하지 않습니다.

arg 매개 변수는 앞의 이벤트에 따라 다릅니다.

추적 함수에 대한 자세한 내용은 *sys.settrace()* 설명서를 참조하십시오. 코드와 프레임 객체에 대한 자세한 내용은 types을 참조하십시오.

dispatch_line(*frame*)

디버거가 현재 행에서 중지해야 하면, *user_line()* 메서드를 호출하십시오 (서브 클래스에서 재정의되어야 합니다). *quitting* 플래그가 설정되면 (*user_line()*에서 설정할 수 있습니다) *BdbQuit* 예외를 발생시킵니다. 해당 스코프에서 추가 추적을 위해 *trace_dispatch()* 메서드에 대한 참조를 반환합니다.

dispatch_call(*frame*, *arg*)

디버거가 이 함수 호출에서 중지해야 하면, *user_call()* 메서드를 호출하십시오 (서브 클래스에서 재정의되어야 합니다). *quitting* 플래그가 설정되면 (*user_call()*에서 설정할 수 있습니다) *BdbQuit* 예외를 발생시킵니다. 해당 스코프에서 추가 추적을 위해 *trace_dispatch()* 메서드에 대한 참조를 반환합니다.

dispatch_return(*frame*, *arg*)

디버거가 이 함수 반환에서 중지해야 하면, *user_return()* 메서드를 호출하십시오 (서브 클래스에서 재정의되어야 합니다). *quitting* 플래그가 설정되면 (*user_return()*에서 설정할 수 있

습니다) `BdbQuit` 예외를 발생시킵니다. 해당 스코프에서 추가 추적을 위해 `trace_dispatch()` 메서드에 대한 참조를 반환합니다.

dispatch_exception(*frame*, *arg*)

디버거가 이 예외에서 중지해야 하면, `user_exception()` 메서드를 호출하십시오 (서브 클래스에서 재정의되어야 합니다). `quitting` 플래그가 설정되면 (`user_exception()`에서 설정할 수 있습니다) `BdbQuit` 예외를 발생시킵니다. 해당 스코프에서 추가 추적을 위해 `trace_dispatch()` 메서드에 대한 참조를 반환합니다.

일반적으로 파생된 클래스는 다음 메서드를 재정의하지 않지만, 중지와 중단점의 정의를 재정의하려고 하면 그럴 수 있습니다.

is_skipped_line(*module_name*)

Return `True` if *module_name* matches any skip pattern.

stop_here(*frame*)

Return `True` if *frame* is below the starting frame in the stack.

break_here(*frame*)

Return `True` if there is an effective breakpoint for this line.

Check whether a line or function breakpoint exists and is in effect. Delete temporary breakpoints based on information from `effective()`.

break_anywhere(*frame*)

Return `True` if any breakpoint exists for *frame*' s filename.

파생 클래스는 디버거 연산을 제어하기 위해 이 메서드를 재정의해야 합니다.

user_call(*frame*, *argument_list*)

호출된 함수 내부에서 중단이 발생할 수 있으면 `dispatch_call()`에서 호출됩니다.

argument_list is not used anymore and will always be `None`. The argument is kept for backwards compatibility.

user_line(*frame*)

`stop_here()`나 `break_here()`가 `True`를 반환할 때 `dispatch_line()`에서 호출됩니다.

user_return(*frame*, *return_value*)

`stop_here()`가 `True`를 반환할 때 `dispatch_return()`에서 호출됩니다.

user_exception(*frame*, *exc_info*)

`stop_here()`가 `True`를 반환할 때 `dispatch_exception()`에서 호출됩니다.

do_clear(*arg*)

중단점이 일시적일 때 중단점을 제거하는 방법을 처리합니다.

이 메서드는 파생 클래스에서 구현해야 합니다.

파생 클래스와 클라이언트는 다음 메서드를 호출하여 스테핑(stepping) 상태에 영향을 줄 수 있습니다.

set_step()

한 줄의 코드 후에 멈춥니다.

set_next(*frame*)

주어진 프레임 내 또는 아래의 다음 줄에서 멈춥니다.

set_return(*frame*)

주어진 프레임에서 반환할 때 멈춥니다.

set_until(*frame*, *lineno=None*)

현재 줄보다 큰 *lineno*를 갖는 줄에 도달하거나 현재 프레임에서 반환할 때 멈춥니다.

set_trace([*frame*])

*frame*에서 디버깅을 시작합니다. *frame*을 지정하지 않으면 호출자의 프레임에서 디버깅을 시작합니다.

버전 3.13에서 변경: `set_trace()` will enter the debugger immediately, rather than on the next line of code to be executed.

set_continue()

중단점에서나 완료 시에만 멈춥니다. 중단점이 없으면, 시스템 추적 함수를 `None`으로 설정합니다.

set_quit()

`quitting` 어트리뷰트를 `True`로 설정합니다. 다음에 `dispatch_*()` 메서드 중 하나를 호출할 때 `BdbQuit`가 발생합니다.

파생 클래스와 클라이언트는 다음 메서드를 호출하여 중단점을 조작할 수 있습니다. 이 메서드는 문제가 발생하면 에러 메시지가 포함된 문자열을 반환하고, 모두 정상이면 `None`을 반환합니다.

set_break(*filename*, *lineno*, *temporary=False*, *cond=None*, *funcname=None*)

새로운 중단점을 설정합니다. 인자로 전달된 *filename*에 *lineno* 줄이 없으면 에러 메시지를 반환합니다. `canonic()` 메서드에 설명된 대로 *filename*은 규범적 형식이어야 합니다.

clear_break(*filename*, *lineno*)

*filename*과 *lineno*에서 중단점을 삭제합니다. 설정되지 않았으면, 에러 메시지를 반환합니다.

clear_bpbynumber(*arg*)

`Breakpoint.bpbynumber`에서 인덱스 *arg*인 중단점을 삭제합니다. *arg*가 숫자가 아니거나 범위를 벗어나면, 에러 메시지를 반환합니다.

clear_all_file_breaks(*filename*)

*filename*에서 모든 중단점을 삭제합니다. 설정되지 않았으면, 에러 메시지를 반환합니다.

clear_all_breaks()

모든 기존 중단점을 삭제합니다. 설정되지 않았으면, 에러 메시지를 반환합니다.

get_bpbynumber(*arg*)

주어진 숫자로 지정된 중단점을 반환합니다. *arg*가 문자열이면, 숫자로 변환됩니다. *arg*가 숫자가 아닌 문자열이면, 지정된 중단점이 존재하지 않거나 삭제되었으면, `ValueError`가 발생합니다.

Added in version 3.2.

get_break(*filename*, *lineno*)

*filename*의 *lineno*에 중단점이 있으면 `True`를 반환합니다.

get_breaks(*filename*, *lineno*)

*filename*의 *lineno*에 있는 모든 중단점을 반환하거나, 없으면 빈 리스트를 반환합니다.

get_file_breaks(*filename*)

*filename*의 모든 중단점을 반환하거나, 없으면 빈 리스트를 반환합니다.

get_all_breaks()

설정된 모든 중단점을 반환합니다.

Derived classes and clients can call the following methods to disable and restart events to achieve better performance. These methods only work when using the `'monitoring'` backend.

disable_current_event()

Disable the current event until the next time `restart_events()` is called. This is helpful when the debugger is not interested in the current line.

Added in version 3.14.

restart_events()

Restart all the disabled events. This function is automatically called in `dispatch_*` methods after `user_*` methods are called. If the `dispatch_*` methods are not overridden, the disabled events will be restarted after each user interaction.

Added in version 3.14.

파생 클래스와 클라이언트는 다음 메서드를 호출하여 스택 추적을 나타내는 데이터 구조를 얻을 수 있습니다.

get_stack(*f*, *t*)

Return a list of (frame, lineno) tuples in a stack trace, and a size.

The most recently called frame is last in the list. The size is the number of frames below the frame where the debugger was invoked.

format_stack_entry(*frame_lineno*, *lprefix*=': ')

`(frame, lineno)` 튜플인 스택 항목에 대한 정보가 포함된 문자열을 반환합니다. 반환 문자열은 다음을 포함합니다:

- 프레임을 포함하는 규범적 파일명.
- 함수 이름이나 `"<lambda>"`.
- 입력 인자.
- 반환 값.
- 코드 줄 (있으면).

클라이언트가 문자열로 지정된 *statement*를 디버깅하기 위해 디버거를 사용하려면 다음 두 가지 메서드를 호출할 수 있습니다.

run(*cmd*, *globals=None*, *locals=None*)

`exec()` 함수를 통해 실행된 문장을 디버그합니다. *globals*의 기본값은 `__main__.__dict__` 이고, *locals*의 기본값은 *globals*입니다.

runeval(*expr*, *globals=None*, *locals=None*)

`eval()` 함수를 통해 실행된 표현식을 디버그합니다. *globals*와 *locals*는 `run()`과 같은 의미입니다.

runctx(*cmd*, *globals*, *locals*)

이전 버전과의 호환성을 위해. `run()` 메서드를 호출합니다.

runcall(*func*, */*, **args*, ***kwds*)

단일 함수 호출을 디버그하고, 결과를 반환합니다.

마지막으로, 모듈은 다음과 같은 함수를 정의합니다:

bdb.**checkfuncname**(*b*, *frame*)

`Breakpoint` *b*가 설정된 방식에 따라, 여기서 중단해야 하면 `True`를 반환합니다.

줄 번호를 통해 설정되었으면, `b.line`이 *frame*의 것과 같은지 확인합니다. 함수 이름을 통해 중단점이 설정되었으면, 우리는 올바른 *frame*(올바른 함수)에 있는지와 그것의 첫 번째 실행 가능한 줄에 있는지 확인해야 합니다.

bdb.**effective**(*file*, *line*, *frame*)

Return `(active breakpoint, delete temporary flag)` or `(None, None)` as the breakpoint to act upon.

The *active breakpoint* is the first entry in `bplist` for the (`file`, `line`) (which must exist) that is `enabled`, for which `checkfuncname()` is true, and that has neither a false `condition` nor positive `ignore` count. The *flag*, meaning that a temporary breakpoint should be deleted, is `False` only when the `cond` cannot be evaluated (in which case, `ignore` count is ignored).

If no such entry exists, then `(None, None)` is returned.

bdb.**set_trace**()

호출자의 프레임에서 *Bdb* 인스턴스로 디버깅을 시작합니다.

27.3 faulthandler — 파이썬 트레이스백 덤프

Added in version 3.3.

이 모듈은 결함(fault) 시, 시간 초과 후 또는 사용자 시그널에 파이썬 트레이스백을 명시적으로 덤프하는 함수를 포함합니다. *SIGSEGV*, *SIGFPE*, *SIGABRT*, *SIGBUS* 및 *SIGILL* 시그널에 대한 결함 처리기를 설치하려면 *faulthandler.enable()*를 호출하십시오. PYTHONFAULTHANDLER 환경 변수를 설정하거나 -X faulthandler 명령 줄 옵션을 사용하여 시작할 때 활성화할 수도 있습니다.

결함 처리기는 Apport나 윈도우 결함 처리기(Windows fault handler)와 같은 시스템 결함 처리기와 호환됩니다. 이 모듈은 sigaltstack() 함수를 사용할 수 있으면 시그널 처리기에 대체 스택을 사용합니다. 이것은 스택 오버플로에서조차 트레이스백을 덤프할 수 있도록 합니다.

결함 처리기는 치명적일 때 호출되므로 시그널 안전한 함수만 사용할 수 있습니다 (예를 들어, 힙에 메모리를 할당할 수 없습니다). 이 제한 때문에 일반적인 파이썬 트레이스백에 비해 트레이스백 덤프는 최소화됩니다:

- ASCII만 지원됩니다. 인코딩 시 backslashreplace 에러 처리기가 사용됩니다.
- 각 문자열은 500자로 제한됩니다.
- 파일명, 함수 이름 및 줄 번호만 표시됩니다. (소스 코드 없음)
- 100프레임과 100스레드로 제한됩니다.
- 순서가 뒤집힙니다: 가장 최근의 호출이 먼저 표시됩니다.

기본적으로, 파이썬 트레이스백은 *sys.stderr*에 기록됩니다. 트레이스백을 보려면, 응용 프로그램이 터미널에서 실행되어야 합니다. 로그 파일을 *faulthandler.enable()*로 전달할 수도 있습니다.

모듈은 C로 구현되어 있으므로, 충돌 시나 파이썬이 교착 상태에 빠질 때 트레이스백을 덤프할 수 있습니다.

파이썬 개발 모드는 파이썬 시작 시 *faulthandler.enable()*을 호출합니다.

> **더 보기**
>
> **Module *pdb***
> Interactive source code debugger for Python programs.
>
> **Module *traceback***
> Standard interface to extract, format and print stack traces of Python programs.

27.3.1 트레이스백 덤프하기

faulthandler.**dump_traceback**(*file=sys.stderr*, *all_threads=True*)

모든 스레드의 트레이스백을 *file*로 덤프합니다. *all_threads*가 False면, 현재 스레드만 덤프합니다.

> **더 보기**
>
> *traceback.print_tb()*, which can be used to print a traceback object.

버전 3.5에서 변경: 이 함수에 파일 기술자를 전달하는 지원이 추가되었습니다.

27.3.2 Dumping the C stack

Added in version 3.14.

`faulthandler.dump_c_stack(file=sys.stderr)`

Dump the C stack trace of the current thread into *file*.

If the Python build does not support it or the operating system does not provide a stack trace, then this prints an error in place of a dumped C stack.

C Stack Compatibility

If the system does not support the C-level `backtrace(3)` or `dladdr1(3)`, then C stack dumps will not work. An error will be printed instead of the stack.

Additionally, some compilers do not support *CPython' s* implementation of C stack dumps. As a result, a different error may be printed instead of the stack, even if the operating system supports dumping stacks.

> **참고**
>
> Dumping C stacks can be arbitrarily slow, depending on the DWARF level of the binaries in the call stack.

27.3.3 결함 처리기 상태

`faulthandler.enable(file=sys.stderr, all_threads=True, c_stack=True)`

결함 처리기를 활성화합니다: `SIGSEGV`, `SIGFPE`, `SIGABRT`, `SIGBUS` 및 `SIGILL` 시그널에 대한 처리기를 설치하여 파이썬 트레이스백을 덤프합니다. *all_threads* 가 `True` 면 실행 중인 모든 스레드에 대한 트레이스백을 생성합니다. 그렇지 않으면, 현재 스레드만 덤프합니다.

file 은 결함 처리기가 비활성화될 때까지 열려 있어야 합니다: 파일 기술자 관련 문제를 참조하십시오.

If *c_stack* is `True`, then the C stack trace is printed after the Python traceback, unless the system does not support it. See `dump_c_stack()` for more information on compatibility.

버전 3.5에서 변경: 이 함수에 파일 기술자를 전달하는 지원이 추가되었습니다.

버전 3.6에서 변경: 윈도우에서는, 윈도우 예외 (Windows exception) 처리기도 설치됩니다.

버전 3.10에서 변경: The dump now mentions if a garbage collector collection is running if *all_threads* is true.

버전 3.14에서 변경: Only the current thread is dumped if the *GIL* is disabled to prevent the risk of data races.

버전 3.14에서 변경: The dump now displays the C stack trace if *c_stack* is true.

`faulthandler.disable()`

결함 처리기를 비활성화합니다: `enable()` 로 설치된 시그널 처리기를 제거합니다.

`faulthandler.is_enabled()`

결함 처리기가 활성화되었는지 검사합니다.

27.3.4 시간 초과 후에 트레이스백 덤프하기

`faulthandler.dump_traceback_later(timeout, repeat=False, file=sys.stderr, exit=False)`

timeout 초의 시간제한 후, 또는 *repeat* 가 `True` 면 매 *timeout* 초마다, 모든 스레드의 트레이스백을 덤프합니다. *exit* 가 `True` 면, 트레이스백을 덤프한 후 status=1 로 `_exit()` 를 호출합니다. (`_exit()` 가 프로세스를 즉시 종료함에 유의하십시오. 파일 버퍼를 플러시 하는 것과 같은 정리 작업을 수행하지 않습니다.) 함수가 두 번 호출되면, 새 호출은 이전 매개 변수를 대체하고 시간제한을 다시 설정합니다. 타이머는 1초 미만의 해상도를 갖습니다.

file 은 트레이스백이 덤프 되거나 `cancel_dump_traceback_later()` 가 호출될 때까지 열려 있어야 합니다: 파일 기술자 관련 문제를 참조하십시오.

이 함수는 워치독(watchdog) 스레드를 사용하여 구현됩니다.

버전 3.5에서 변경: 이 함수에 파일 기술자를 전달하는 지원이 추가되었습니다.

버전 3.7에서 변경: 이 함수는 이제 항상 사용할 수 있습니다.

faulthandler.**cancel_dump_traceback_later**()

마지막 *dump_traceback_later()* 호출을 취소합니다.

27.3.5 사용자 시그널에 트레이스백 덤프하기

faulthandler.**register**(*signum*, *file=sys.stderr*, *all_threads=True*, *chain=False*)

사용자 시그널을 등록합니다: *signum* 시그널에 대한 처리기를 설치해서, 모든 스레드, 또는 *all_threads* 가 `False`면 현재 스레드의, 트레이스백을 *file*로 덤프합니다. chain이 `True`면 이전 처리기를 호출합니다.

*file*은 시그널이 *unregister()*로 등록 해지 될 때까지 열려 있어야 합니다: 파일 기술자 관련 문제를 참조하십시오.

윈도우에서는 사용할 수 없습니다.

버전 3.5에서 변경: 이 함수에 파일 기술자를 전달하는 지원이 추가되었습니다.

faulthandler.**unregister**(*signum*)

사용자 시그널을 등록 해지합니다: *register()*로 설치된 *signum* 시그널 처리기를 제거합니다. 시그널이 등록되었으면 `True`를 반환하고, 그렇지 않으면 `False`를 반환합니다.

윈도우에서는 사용할 수 없습니다.

27.3.6 파일 기술자 관련 문제

enable(), *dump_traceback_later()* 및 *register()*는 *file* 인자의 파일 기술자를 유지합니다. 파일이 닫히고 파일 기술자가 새 파일에 의해 다시 사용되거나, *os.dup2()*가 파일 기술자를 바꾸는 데 사용되면, 트레이스백이 다른 파일에 기록됩니다. 파일을 바꿀 때마다 이 함수들을 다시 호출하십시오.

27.3.7 예제

리눅스에서 결함 처리기를 활성화하거나 그렇지 않았을 때의 세그멘테이션 결함 예제:

```
$ python -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700 (most recent call first):
  File "/opt/python/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>

Current thread's C stack trace (most recent call first):
  Binary file "/opt/python/python", at _Py_DumpStack+0x42 [0x5b27f7d7147e]
  Binary file "/opt/python/python", at +0x32dcbd [0x5b27f7d85cbd]
  Binary file "/opt/python/python", at +0x32df8a [0x5b27f7d85f8a]
  Binary file "/usr/lib/libc.so.6", at +0x3def0 [0x77b73226bef0]
  Binary file "/usr/lib/libc.so.6", at +0x17ef9c [0x77b7323acf9c]
  Binary file "/opt/python/build/lib.linux-x86_64-3.14/_ctypes.cpython-314d-x86_64-
↪linux-gnu.so", at +0xcdf6 [0x77b7315dddf6]
  Binary file "/usr/lib/libffi.so.8", at +0x7976 [0x77b73158f976]
  Binary file "/usr/lib/libffi.so.8", at +0x413c [0x77b73158c13c]
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  Binary file "/usr/lib/libffi.so.8", at ffi_call+0x12e [0x77b73158ef0e]
  Binary file "/opt/python/build/lib.linux-x86_64-3.14/_ctypes.cpython-314d-x86_64-
↪linux-gnu.so", at +0x15a33 [0x77b7315e6a33]
  Binary file "/opt/python/build/lib.linux-x86_64-3.14/_ctypes.cpython-314d-x86_64-
↪linux-gnu.so", at +0x164fa [0x77b7315e74fa]
  Binary file "/opt/python/build/lib.linux-x86_64-3.14/_ctypes.cpython-314d-x86_64-
↪linux-gnu.so", at +0xc624 [0x77b7315dd624]
  Binary file "/opt/python/python", at _PyObject_MakeTpCall+0xce [0x5b27f7b73883]
  Binary file "/opt/python/python", at +0x11bab6 [0x5b27f7b73ab6]
  Binary file "/opt/python/python", at PyObject_Vectorcall+0x23 [0x5b27f7b73b04]
  Binary file "/opt/python/python", at _PyEval_EvalFrameDefault+0x490c␣
↪[0x5b27f7cbb302]
  Binary file "/opt/python/python", at +0x2818e6 [0x5b27f7cd98e6]
  Binary file "/opt/python/python", at +0x281aab [0x5b27f7cd9aab]
  Binary file "/opt/python/python", at PyEval_EvalCode+0xc5 [0x5b27f7cd9ba3]
  Binary file "/opt/python/python", at +0x255957 [0x5b27f7cad957]
  Binary file "/opt/python/python", at +0x255ab4 [0x5b27f7cadab4]
  Binary file "/opt/python/python", at _PyEval_EvalFrameDefault+0x6c3e␣
↪[0x5b27f7cbd634]
  Binary file "/opt/python/python", at +0x2818e6 [0x5b27f7cd98e6]
  Binary file "/opt/python/python", at +0x281aab [0x5b27f7cd9aab]
  Binary file "/opt/python/python", at +0x11b6e1 [0x5b27f7b736e1]
  Binary file "/opt/python/python", at +0x11d348 [0x5b27f7b75348]
  Binary file "/opt/python/python", at +0x11d626 [0x5b27f7b75626]
  Binary file "/opt/python/python", at PyObject_Call+0x20 [0x5b27f7b7565e]
  Binary file "/opt/python/python", at +0x32a67a [0x5b27f7d8267a]
  Binary file "/opt/python/python", at +0x32a7f8 [0x5b27f7d827f8]
  Binary file "/opt/python/python", at +0x32ac1b [0x5b27f7d82c1b]
  Binary file "/opt/python/python", at Py_RunMain+0x31 [0x5b27f7d82ebe]
  <truncated rest of calls>
Segmentation fault
```

27.4 pdb — 파이썬 디버거

소스 코드: Lib/pdb.py

---

pdb 모듈은 파이썬 프로그램을 위한 대화형 소스 코드 디버거를 정의합니다. 소스 라인 단계의 중단점 (breakpoint) 및 단계 실행 (single stepping) 설정, 스택 프레임 검사, 소스 코드 목록, 그리고 모든 스택 프레임의 컨텍스트에서 임의의 파이썬 코드 평가를 지원합니다. 또한 포스트-모템 (post-mortem) 디버깅을 지원하며, 프로그램 제어 하에서도 호출될 수 있습니다.

이 디버거는 확장이 가능합니다 – 디버거는 실제로 *Pdb* 클래스로 정의됩니다. 현재 문서화되어 있진 않지만, 소스를 읽어보시면 쉽게 이해하실 수 있습니다. 확장 인터페이스는 *bdb* 와 *cmd* 모듈을 활용합니다.

더 보기

모듈 *faulthandler*
Used to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal.

모듈 *traceback*
Standard interface to extract, format and print stack traces of Python programs.

디버거로 진입하는 일반적인 사용법은:

```
import pdb; pdb.set_trace()
```

또는:

```
breakpoint()
```

위 코드를 디버거로 진입하고 싶은 구간에 추가한 다음, 프로그램을 실행하면 됩니다. 그런 다음 이 문장 뒤에 오는 코드를 단계별로 실행하고, *continue* 명령을 사용하여 디버거 없이 프로그램을 계속 실행할 수 있습니다.

버전 3.7에서 변경: 내장 *breakpoint()* 가, 기본값으로 호출될 때는, `import pdb; pdb.set_trace()` 를 대신해서 사용할 수 있습니다.

```
def double(x):
    breakpoint()
    return x * 2
val = 3
print(f"{val} * 2 is {double(val)}")
```

The debugger' s prompt is `(Pdb)`, which is the indicator that you are in debug mode:

```
> ...(2)double()
-> breakpoint()
(Pdb) p x
3
(Pdb) continue
3 * 2 is 6
```

버전 3.3에서 변경: *readline* 모듈을 통한 탭-완성은 명령과 명령 인자에 사용할 수 있습니다, 예를 들면, 현재 전역 및 지역 이름들은 `p` 명령의 인자로 제공됩니다. 다른 스크립트를 디버그하기 위해 명령 줄에서 *pdb*를 호출할 수도 있습니다. 예를 들면:

```
python -m pdb [-c command] (-m module | -p pid | pyfile) [args ...]
```

모듈로 호출하는 경우, 디버깅 중인 프로그램이 비정상적으로 종료되면 pdb는 자동으로 포스트-모템(post-mortem) 디버깅을 시작합니다. 포스트-모템 디버깅이 끝나면 (또는 프로그램이 정상적으로 종료되면), pdb는 프로그램을 재시작합니다. 자동 재시작은 중단점과 같은 pdb의 상태를 유지하고 대부분의 경우 프로그램 종료 시 디버거를 종료하는 것보다 유용합니다.

-c, --command <command>

`.pdbrc` 파일에 주어진 것처럼 명령을 실행하기; 디버거 명령들 을 확인해보세요.

버전 3.2에서 변경: `-c` 옵션을 추가했습니다.

-m <module>

`python -m`과 비슷한 방식으로 모듈을 실행하기. 스크립트와 마찬가지로, 디버거는 모듈의 첫 번째 줄 바로 전에 실행을 일시정지합니다.

버전 3.7에서 변경: `-m` 옵션을 추가했습니다.

-p, --pid <pid>

Attach to the process with the specified PID.

Added in version 3.14.

To attach to a running Python process for remote debugging, use the `-p` or `--pid` option with the target process' s PID:

```
python -m pdb -p 1234
```

> **참고**
>
> Attaching to a process that is blocked in a system call or waiting for I/O will only work once the next bytecode instruction is executed or when the process receives a signal.

디버거 제어하에 문장을 실행하는 일반적인 사용법은 다음과 같습니다:

```
>>> import pdb
>>> def f(x):
...     print(1 / x)
>>> pdb.run("f(2)")
> <string>(1)<module>()
(Pdb) continue
0.5
>>>
```

에러가 발생하는 프로그램을 검사하는 일반적인 사용법은 다음과 같습니다:

```
>>> import pdb
>>> def f(x):
...     print(1 / x)
...
>>> f(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
ZeroDivisionError: division by zero
>>> pdb.pm()
> <stdin>(2)f()
(Pdb) p x
0
(Pdb)
```

버전 3.13에서 변경: The implementation of **PEP 667** means that name assignments made via `pdb` will immediately affect the active scope, even when running inside an *optimized scope*.

이 모듈은 다음과 같은 함수를 정의합니다; 각 함수는 조금 다른 방식으로 디버거로 진입합니다:

`pdb.`**`run`**(*statement*, *globals=None*, *locals=None*)

디버거 제어하에 *statement* (주어진 문자열 또는 코드 객체)를 실행합니다. 코드가 실행하기 전에 디버거 프롬프트가 나타납니다; 중단점을 지정하고 `continue`를 입력하거나, `step` 또는 `next`를 통해 문장을 단계별로 살펴볼 수 있습니다. (이 모든 명령은 아래에 설명되어 있습니다.) 선택적 인자 *globals* 와 *locals*는 코드가 실행되는 환경을 구체적으로 명시합니다; 기본적으로는 이 모듈의 딕셔너리 `__main__` 이 사용됩니다. (내장 함수 `exec()` 또는 `eval()` 에 대한 설명 참조.)

`pdb.`**`runeval`**(*expression*, *globals=None*, *locals=None*)

디버거 제어하에 문자열 또는 코드 객체로 주어진 *expression*을 평가합니다. `runeval()` 이 반환될 때, *expression*의 값을 반환합니다. 그렇지 않으면 이 함수는 `run()`과 유사한 함수입니다.

`pdb.`**`runcall`**(*function*, **args*, ***kwds*)

주어진 인자와 함께 *function* (문자열이 아닌, 함수 또는 메서드 객체)를 호출합니다. `runcall()` 이 반환될 때, 함수 호출로 반환된 값을 반환합니다. 디버거 프롬프트는 함수에 진입하자마자 나타납니다.

`pdb.`**`set_trace`**(**, *header=None*, *commands=None*)

Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails). If given, *header* is printed to the console just before debugging begins. The *commands* argument, if given, is a list of commands to execute when the debugger starts.

버전 3.7에서 변경: 키워드 전용 인자 *header*.

버전 3.13에서 변경: `set_trace()` will enter the debugger immediately, rather than on the next line of code to be executed.

Added in version 3.14: The *commands* argument.

awaitable pdb.**set_trace_async**(*, *header=None*, *commands=None*)

async version of `set_trace()`. This function should be used inside an async function with `await`.

```
async def f():
    await pdb.set_trace_async()
```

`await` statements are supported if the debugger is invoked by this function.

Added in version 3.14.

pdb.**post_mortem**(*t=None*)

주어진 예외나 트레이스백 객체의 포스트-모템(post-mortem) 디버깅으로 진입합니다. 만약 값이 주어지지 않았다면, 현재 처리되고 있는 예외를 사용거나, 없다면 `ValueError`를 발생시킵니다.

버전 3.13에서 변경: Support for exception objects was added.

pdb.**pm**()

`sys.last_exc` 에서 찾은 예외의 포스트-모템(post-mortem) 디버깅으로 진입합니다.

pdb.**set_default_backend**(*backend*)

There are two supported backends for pdb: `'settrace'` and `'monitoring'`. See `bdb.Bdb` for details. The user can set the default backend to use if none is specified when instantiating `Pdb`. If no backend is specified, the default is `'settrace'`.

> **참고**
>
> `breakpoint()` and `set_trace()` will not be affected by this function. They always use `'monitoring'` backend.

Added in version 3.14.

pdb.**get_default_backend**()

Returns the default backend for pdb.

Added in version 3.14.

`run*` 함수와 `set_trace()`는 `Pdb` 클래스를 인스턴스 화하고 같은 이름의 메서드를 호출하는 에일리어스(alias) 입니다. 더 많은 기능에 액세스하려면, 아래를 참고하여 직접 하셔야 합니다:

class pdb.**Pdb**(*completekey='tab', stdin=None, stdout=None, skip=None, nosigint=False, readrc=True, mode=None, backend=None, colorize=False*)

`Pdb` 는 디버거 클래스입니다.

completekey, *stdin* 그리고 *stdout* 인자는 내부 `cmd.Cmd` 클래스로 전달됩니다; 자세한 설명은 해당 클래스에서 확인할 수 있습니다.

skip 인자가 주어진다면, 반드시 글로브-스타일(glob-style) 모듈 이름 패턴의 이터러블(iterable) 이어야 합니다. 디버거는 이 패턴 중 하나와 일치하는 모듈에서 시작되는 프레임 단계로 들어가지 않습니다.[1]

기본적으로, Pdb는 사용자가 `continue` 명령을 내릴 때, SIGINT 신호(사용자가 콘솔에서 `Ctrl-C`를 누를 때 전송되는 신호) 에 대한 핸들러를 설정합니다. 사용자는 `Ctrl-C`를 눌러서 디버거를 벗어날 수 있습니다. 만약 Pdb가 SIGINT 핸들러를 건드리지 않길 원한다면 *nosigint* 설정을 참으로 변경하면 됩니다.

readrc 인자는 기본적으로 참이고 Pdb가 파일 시스템으로부터 .pdbrc를 불러올지 여부를 제어합니다.

[1] 프레임이 특정 모듈에서 시작되는 것으로 간주하는지 여부는 프레임 전역에 있는 `__name__`에 의해 결정됩니다.

The *mode* argument specifies how the debugger was invoked. It impacts the workings of some debugger commands. Valid values are `'inline'` (used by the breakpoint() builtin), `'cli'` (used by the command line invocation) or `None` (for backwards compatible behaviour, as before the *mode* argument was added).

The *backend* argument specifies the backend to use for the debugger. If `None` is passed, the default backend will be used. See `set_default_backend()`. Otherwise the supported backends are `'settrace'` and `'monitoring'`.

The *colorize* argument, if set to `True`, will enable colorized output in the debugger, if color is supported. This will highlight source code displayed in pdb.

*skip*으로 추적하기 위한 호출 예시:

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

인자 없이 감사 이벤트 `pdb.Pdb`를 발생시킵니다.

버전 3.1에서 변경: *skip* 매개 변수를 추가했습니다.

버전 3.2에서 변경: *nosigint* 매개 변수를 추가했습니다. 이전에는, Pdb가 SIGINT 핸들러를 설정하지 않았습니다.

버전 3.6에서 변경: *readrc* 인자.

Added in version 3.14: Added the *mode* argument.

Added in version 3.14: Added the *backend* argument.

Added in version 3.14: Added the *colorize* argument.

버전 3.14에서 변경: Inline breakpoints like `breakpoint()` or `pdb.set_trace()` will always stop the program at calling frame, ignoring the *skip* pattern (if any).

run(*statement*, *globals=None*, *locals=None*)

runeval(*expression*, *globals=None*, *locals=None*)

runcall(*function*, **args*, ***kwds*)

set_trace()

위에 설명된 함수에 대한 설명서를 참조하시면 됩니다.

27.4.1 디버거 명령들

디버거가 인식할 수 있는 명령이 아래 나열되어 있습니다. 대부분의 명령은 한두 문자로 단축될 수 있습니다; 예를 들면, `h(elp)` 는 `h` 또는 `help`로 help 명령을 입력할 때 사용할 수 있습니다. (하지만 `he`, `hel`, `H`, `Help` 또는 `HELP`는 사용할 수 없습니다.) 인자는 반드시 명령과 공백(스페이스나 탭)으로 분리되어야 합니다. 선택적 인자는 명령 문법에서 대괄호(`[]`)로 묶여있습니다; 대괄호는 입력하지 않습니다. 명령 문법에서 대체 가능한 인자는 세로 바(`|`)로 분리되어 있습니다.

빈 줄을 입력하면 마지막으로 입력된 명령이 반복됩니다. 예외: 만약 마지막 명령이 `list` 명령이면, 다음 11줄이 나열됩니다.

디버거가 인식하지 못하는 명령은 파이썬 문장으로 가정하고 디버깅 중인 프로그램의 컨텍스트에서 실행됩니다. 파이썬 문장 앞에 느낌표(`!`)를 붙여 사용할 수도 있습니다. 이 방법은 디버깅 중인 프로그램을 검사하는 강력한 방법입니다. 변수를 변경하거나 함수를 호출하는 것도 가능합니다. 이 문장에서 예외가 발생하면, 예외 명은 출력되지만 디버거의 상태는 변경되지 않습니다.

버전 3.13에서 변경: Expressions/Statements whose prefix is a pdb command are now correctly identified and executed.

디버거는 에일리어스을 지원합니다. 에일리어스는 검사 중인 컨텍스트에서 특정 수준의 적응성을 허용하는 매개변수를 가질 수 있습니다.

한 줄에 여러 명령은 `;;` 로 구분하여 입력할 수 있습니다. (단일 `;` 는 파이썬 파서로 전달되는 한 줄에서, 여러 명령을 구분하기 위한 분리 기호이므로 사용되지 않습니다.) 명령을 똑똑하게 분리하진 못합니다; 입력의 맨 처음 `;;` 에서 나뉘며, 따옴표로 묶인 문자열 중간에 있더라도 나눠집니다. 이중 세미콜론이 있는 문자열의 해결 방법은 묵시적 문자열 이어붙이기 `';'';'` 나 `";""";"`를 사용하는 것입니다.

To set a temporary global variable, use a *convenience variable*. A *convenience variable* is a variable whose name starts with `$`. For example, `$foo = 1` sets a global variable `$foo` which you can use in the debugger session. The *convenience variables* are cleared when the program resumes execution so it' s less likely to interfere with your program compared to using normal variables like `foo = 1`.

There are four preset *convenience variables*:

- `$_frame`: the current frame you are debugging
- `$_retval`: 프레임이 반환 중이면 반환 값
- `$_exception`: the exception if the frame is raising an exception
- `$_asynctask`: the asyncio task if pdb stops in an async function

Added in version 3.12: Added the *convenience variable* feature.

Added in version 3.14: Added the `$_asynctask` convenience variable.

만약 파일 `.pdbrc` 가 사용자의 홈 디렉터리 또는 현재 디렉터리에 있으면, 디버거 프롬프트에서 입력된 것처럼 `'utf-8'` 인코딩으로 읽히고 실행됩니다. 빈 줄과 `#`로 시작하는 줄은 무시됩니다. 이것은 특히 에일리어스에 유용합니다. 만약 두 파일이 모두 존재하면, 홈 디렉터리에 있는 파일이 먼저 읽히고 거기에 정의된 에일리어스는 로컬 파일에 의해 무시될 수 있습니다.

버전 3.2에서 변경: `.pdbrc` 는 `continue`와 `next`같이 디버깅을 계속하는 명령을 포함할 수 있습니다. 이전에는, 이런 명령이 효과가 없었습니다.

버전 3.11에서 변경: `.pdbrc` is now read with `'utf-8'` encoding. Previously, it was read with the system locale encoding.

h(elp) `[command]`

인자가 없는 경우에는, 사용 가능한 명령 리스트를 출력합니다. *command* 인자가 주어진 경우에는, 해당 명령의 도움말을 출력합니다. `help pdb` 는 전체 문서(`pdb` 모듈의 독스트링)를 표시합니다. *command* 인자는 반드시 식별자이어야 하므로, `!` 명령의 도움말을 얻기 위해서 `help exec` 가 꼭 입력되어야 합니다.

w(here) `[count]`

Print a stack trace, with the most recent frame at the bottom. if *count* is 0, print the current frame entry. If *count* is negative, print the least recent - *count* frames. If *count* is positive, print the most recent *count* frames. An arrow (>) indicates the current frame, which determines the context of most commands.

버전 3.14에서 변경: *count* argument is added.

d(own) `[count]`

스택 트레이스에서 현재 프레임을 *count* (기본 1) 단계 아래로 (새로운 프레임으로) 이동합니다.

u(p) `[count]`

스택 트레이스에서 현재 프레임을 *count* (기본 1) 단계 위로 (이전 프레임으로) 이동합니다.

b(reak) `[([filename:]lineno | function) [, condition]]`

With a *lineno* argument, set a break at line *lineno* in the current file. The line number may be prefixed with a *filename* and a colon, to specify a breakpoint in another file (possibly one that hasn' t been loaded yet). The file is searched on `sys.path`. Acceptable forms of *filename* are `/abspath/to/file.py`, `relpath/file.py`, `module` and `package.module`.

function 인자가 주어진 경우, 함수 내에서 첫 번째 실행 가능한 문장에서 브레이크를 설정합니다. *function*은 현재 이름 공간에서 함수로 평가되는 모든 표현식이 될 수 있습니다.

두 번째 인자가 있는 경우, 중단점을 적용하기 전에 표현식이 반드시 참이어야 합니다.

인자가 없다면, 각 중단점, 중단점에 도달한 횟수, 현재까지 무시 횟수, 그리고 관련 조건(있는 경우)을 포함한 모든 중단지점을 나열합니다.

Each breakpoint is assigned a number to which all the other breakpoint commands refer.

tbreak `[([filename:]lineno | function) [, condition]]`

한번 도달하면 제거되는 임시중단점입니다. 인자는 `break`과 동일합니다.

cl(ear) [filename:lineno | bpnumber ...]

filename:lineno 인자가 주어진 경우, 해당 줄에 있는 모든 중단점을 제거합니다. 공백으로 구분된 중단점 번호 배열이 주어진 경우, 해당 중단점을 제거합니다. 인자가 없는 경우, 모든 중단지점을 재차 확인 후 제거합니다.

disable bpnumber [bpnumber ...]

공백으로 구분된 중단점 번호로 해당 중단점을 비활성화합니다. 중단점을 비활성화하는 것은 프로그램이 실행을 중단할 수 없다는 것입니다, 하지만 중단점을 제거하는 것과는 달리, 중단점 목록에 남아있으며 (재-)활성화할 수 있습니다.

enable bpnumber [bpnumber ...]

지정된 중단점을 활성화합니다.

ignore bpnumber [count]

해당 중단점 번호를 무시할 횟수를 설정합니다. 만약 *count*가 생략되면, 무시 횟수는 0으로 설정됩니다. 무시 횟수가 0일 때 중단점이 활성화됩니다. 0이 아닐 때는, 중단점에 도달하고 중단점이 비활성화되지 않고 연관 조건이 참일 때마다 *count*가 차감됩니다.

condition bpnumber [condition]

중단점에 새로운 *condition*을 설정합니다, 표현식이 참일 때만 중단점이 적용됩니다. 만약 *condition*이 없다면, 설정되어있던 모든 조건이 제거됩니다; 즉, 중단점에 적용되어있던 조건이 없어집니다.

commands [bpnumber]

중단점 번호 *bpnumber* 에 대한 명령을 지정합니다. 명령 목록은 다음 줄에 나타나게 됩니다. 명령을 종료하려면 `end`만 입력하면 됩니다. 예를 들면:

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

중단점에 지정된 모든 명령을 제거하려면, `commands` 입력 후에 바로 `end`를 입력하면 됩니다; 즉, 아무 명령을 설정하지 않는 것입니다.

bpnumber 인자가 주어지지 않으면, `commands`는 마지막 중단점 묶음을 참조하게 됩니다.

중단점 명령을 활용해서 프로그램을 다시 시작할 수도 있습니다. *continue* 명령이나, *step* 또는 실행을 재개하는 다른 명령을 사용하기만 하면 됩니다.

Specifying any command resuming execution (currently *continue*, *step*, *next*, *return*, *until*, *jump*, *quit* and their abbreviations) terminates the command list (as if that command was immediately followed by end). This is because any time you resume execution (even with a simple next or step), you may encounter another breakpoint—which could have its own command list, leading to ambiguities about which list to execute.

If the list of commands contains the `silent` command, or a command that resumes execution, then the breakpoint message containing information about the frame is not displayed.

버전 3.14에서 변경: Frame information will not be displayed if a command that resumes execution is present in the command list.

s(tep)

현재 줄을 실행하고, 멈출 수 있는 가장 첫 번째 줄(호출되는 함수 또는 현재 함수의 다음 줄) 에서 멈춥니다.

n(ext)

현재 함수의 다음 줄에 도달하거나, 반환할 때까지 계속 실행합니다. *next*와 *step*의 차이점은 *step*은 호출된 함수 안에서 멈추고, *next*는 호출된 함수를 재빠르게 실행하고 현재 함수의 바로 다음 줄에서만 멈춥니다.

unt(il) [lineno]

인자가 없는 경우에는, 현재 줄 번호보다 높은 줄 번호에 도달할 때까지 계속 실행합니다.

*lineno*가 주어진 경우에는, *lineno*보다 크거나 같은 줄에 도달할 때까지 계속 실행합니다. 두 경우 모두 현재 프레임이 반환될 때 멈춥니다.

버전 3.2에서 변경: 줄 번호를 명시적으로 줄 수 있도록 허용합니다.

r(eturn)

현재 함수가 반환될 때까지 계속 실행합니다.

c(ont(inue))

중단점을 마주칠 때까지 계속 실행합니다.

j(ump) `lineno`

다음으로 실행될 줄을 설정할 수 있습니다. 프레임의 맨 마지막에서만 실행이 가능합니다. 이전 줄로 돌아가 코드를 재실행하거나, 실행을 원치 않는 코드를 건너뛸 수 있습니다.

중요한 점은 이 명령은 언제나 실행할 수 있진 않습니다 – `for` 루프 내부로 들어가거나 `finally` 절을 건너뛰는 것은 불가능합니다.

l(ist) `[first[, last]]`

현재 파일의 소스 코드를 나열합니다. 인자가 없는 경우, 현재 줄 주위로 11줄을 나열하거나 이전 줄을 이어서 나열합니다. 인자로 `.`을 입력한 경우, 현재 줄 주위로 11줄을 나열합니다. 한 인자만 주어진 경우, 해당 줄 주위로 11줄을 나열합니다. 두 인자가 주어진 경우, 두 인자 사이의 모든 줄을 나열합니다; 만약 두 번째 인자가 첫 번째 인자보다 작은 경우, 첫 번째 인자로부터 나열하는 줄 수로 인식합니다.

현재 프레임에서 현재 위치는 `->`로 표시됩니다. 예외를 디버깅할 때, 예외가 최초로 발생하거나 전파된 줄이 현재 줄과 다른 경우에는 `>>`로 표시됩니다.

버전 3.2에서 변경: `>>`표시를 추가했습니다.

ll | `longlist`

현재 함수나 프레임의 소스 코드 전체를 나열합니다. 참고할만한 줄은 `list`처럼 표시됩니다.

Added in version 3.2.

a(rgs)

현재 함수의 인자와 그들의 현재 값을 출력합니다.

p `expression`

현재 컨텍스트에서 *expression*을 평가하고 값을 출력합니다.

> **참고**
>
> 디버거 명령은 아니지만, `print()`도 사용될 수 있습니다 — 이때는 파이썬의 `print()` 함수를 실행하게 됩니다.

pp `expression`

`p` 명령과 비슷하지만, *expression*의 값을 `pprint` 모듈을 활용하여 보기 좋은 형태로 출력합니다.

whatis `expression`

*expression*의 형(type)을 출력합니다.

source `expression`

*expression*의 소스 코드를 가져와서 보여줍니다.

Added in version 3.2.

display `[expression]`

현재 프레임에서 실행이 중지될 때마다, *expression*의 값이 변경된 경우 표시합니다.

*expression*이 주어지지 않은 경우, 현재 프레임에서 표시되는 모든 표현식을 나열합니다.

> **참고**
>
> Display evaluates *expression* and compares to the result of the previous evaluation of *expression*, so when the result is mutable, display may not be able to pick up the changes.

예제:

```
lst = []
breakpoint()
pass
lst.append(1)
print(lst)
```

Display won' t realize `lst` has been changed because the result of evaluation is modified in place by `lst.append(1)` before being compared:

```
> example.py(3)<module>()
-> pass
(Pdb) display lst
display lst: []
(Pdb) n
> example.py(4)<module>()
-> lst.append(1)
(Pdb) n
> example.py(5)<module>()
-> print(lst)
(Pdb)
```

You can do some tricks with copy mechanism to make it work:

```
> example.py(3)<module>()
-> pass
(Pdb) display lst[:]
display lst[:]: []
(Pdb) n
> example.py(4)<module>()
-> lst.append(1)
(Pdb) n
> example.py(5)<module>()
-> print(lst)
display lst[:]: [1]  [old: []]
(Pdb)
```

Added in version 3.2.

undisplay [expression]

현재 프레임에서 *expression*을 더는 표시하지 않습니다. *expression*이 주어지지 않은 경우, 현재 프레임에서 표시되는 모든 표현식을 제거합니다.

Added in version 3.2.

interact

현재 스코프에 대한 지역 및 전역 이름 공간에서 초기화된 새 전역 이름 공간에서 (*code* 모듈을 활용하는) 대화형 인터프리터를 시작합니다. 인터프리터를 종료하고 디버거로 돌아가려면 `exit()` 나 `quit()` 를 사용하세요.

> **참고**

> As `interact` creates a new dedicated namespace for code execution, assignments to variables will not affect the original namespaces. However, modifications to any referenced mutable objects will be reflected in the original namespaces as usual.

Added in version 3.2.

버전 3.13에서 변경: `exit()` and `quit()` can be used to exit the *interact* command.

버전 3.13에서 변경: *interact* directs its output to the debugger' s output channel rather than *sys.stderr*.

alias [name [command]]

*command*를 실행하는 *name*이라 불리는 에일리어스를 생성합니다. *command*는 따옴표로 감싸지 않아야 합니다. 대체할 수 있는 파라미터는 `%1`, `%2`, … 및 `%9` 로 표시되지만, `%*` 는 모든 파라미터로 대체됩니다. 만약 *command*가 생략되면, 현재 *name*의 에일리어스가 표시됩니다. 만약 아무 인자가 주어지지 않으면, 모든 에일리어스가 나열됩니다.

에일리어스는 중첩될 수 있고 pdb 프롬프트 내에서 정당하게 입력할 수 있는 모든 것을 담을 수 있습니다. 주의할 점은 pdb 내부 명령들이 에일리어스에 의해 오버라이드 될 수 있습니다. 그 명령은 에일리어스가 없어질 때까지 사용할 수 없게 됩니다. 에일리어싱은 명령 줄의 첫 번째 단어에 회귀적으로 적용됩니다; 나머지 단어들은 적용되지 않습니다.

`.pdbrc`파일에 추가되면 특히 유용한 두 에일리어스 예시:

```
# 인스턴스 변수를 인쇄합니다 (사용법 "pi classInst")
alias pi for k in %1.__dict__.keys(): print(f"%1.{k} = {%1.__dict__[k]}")
# self 의 인스턴스 변수를 인쇄합니다
alias ps pi self
```

unalias name

지정된 에일리어스 *name*을 제거합니다.

! statement

현재 스택 프레임의 컨텍스트에서 단일 *statement*를 실행합니다. 문장의 첫 단어가 디버거 명령이 아닌 경우, 느낌표는 제외해도 됩니다, 예를 들면:

```
(Pdb) ! n=42
(Pdb)
```

전역 변수를 설정하려면 실행하려는 명령과 동일한 줄 맨 앞에 `global`문장을 붙이면 됩니다, 예를 들면:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [args ...]

restart [args ...]

디버그 된 파이썬 프로그램을 재시작합니다. 만약 *args*가 주어지면, *shlex*으로 나뉘게 되고 결과는 새 *sys.argv*로 사용됩니다. 이전 기록, 중단점, 행동 그리고 디버거 옵션은 유지됩니다. *restart*는 *run*의 에일리어스입니다.

버전 3.14에서 변경: *run* and *restart* commands are disabled when the debugger is invoked in `'inline'` mode.

q(uit)

Quit from the debugger. The program being executed is aborted. An end-of-file input is equivalent to *quit*.

A confirmation prompt will be shown if the debugger is invoked in `'inline'` mode. Either `y`, `Y`, `<Enter>` or `EOF` will confirm the quit.

버전 3.14에서 변경: A confirmation prompt will be shown if the debugger is invoked in `'inline'` mode. After the confirmation, the debugger will call *sys.exit()* immediately, instead of raising *bdb.BdbQuit* in the next trace event.

debug code

code(현재 환경에서 실행될 임의의 표현식이나 문장)를 단계별로 수행하는 재귀적 디버거에 진입합니다.

retval

현재 함수의 마지막 반환에 대한 반환 값을 인쇄합니다.

exceptions [excnumber]

List or jump between chained exceptions.

When using `pdb.pm()` or `Pdb.post_mortem(...)` with a chained exception instead of a traceback, it allows the user to move between the chained exceptions using `exceptions` command to list exceptions, and `exceptions <number>` to switch to that exception.

예제:

```
def out():
    try:
        middle()
    except Exception as e:
        raise ValueError("reraise middle() error") from e

def middle():
    try:
        return inner(0)
    except Exception as e:
        raise ValueError("Middle fail")

def inner(x):
    1 / x

 out()
```

calling `pdb.pm()` will allow to move between exceptions:

```
> example.py(5)out()
-> raise ValueError("reraise middle() error") from e

(Pdb) exceptions
  0 ZeroDivisionError('division by zero')
  1 ValueError('Middle fail')
> 2 ValueError('reraise middle() error')

(Pdb) exceptions 0
> example.py(16)inner()
-> 1 / x

(Pdb) up
> example.py(10)middle()
-> return inner(0)
```

Added in version 3.13.

27.5 파이썬 프로파일러

소스 코드: Lib/profile.py 및 Lib/pstats.py

27.5.1 프로파일러 소개

*cProfile*과 *profile*은 파이썬 프로그램의 결정론적 프로파일링 (*deterministic profiling*)을 제공합니다. 프로파일 (*profile*)은 프로그램의 여러 부분이 얼마나 자주 그리고 얼마나 오랫동안 실행되었는지를 기술하는 통계 집합입니다. 이러한 통계는 *pstats* 모듈을 통해 보고서로 포매팅 할 수 있습니다.

파이썬 표준 라이브러리는 같은 프로파일링 인터페이스의 두 가지 구현을 제공합니다:

1. *cProfile*이 대부분 사용자에게 권장됩니다; 오래 실행되는 프로그램을 프로파일링하는 데 적합한 합리적인 부하를 주는 C 확장입니다. Brett Rosen과 Ted Czotter가 제공한 `lsprof`를 기반으로 합니다.
2. *profile*은 순수 파이썬 모듈이고 이 인터페이스를 *cProfile*이 모방했습니다. 하지만, 프로파일링 되는 프로그램에 상당한 부하를 추가합니다. 어떤 방식으로 프로파일러를 확장하려고 한다면, 이 모듈을 사용하면 작업이 더 쉬울 수 있습니다. Jim Roskind가 원래 설계하고 작성했습니다.

> **참고**
>
> 프로파일러 모듈은 벤치마킹 목적(이를 위해서는 합리적으로 정확한 결과를 주는 *timeit*이 있습니다)이 아니라 주어진 프로그램에 대한 실행 프로파일을 제공하도록 설계되었습니다. 이것은 특히 C 코드에 대한 파이썬 코드 벤치마킹에 적용됩니다: 프로파일러는 파이썬 코드에 부하를 가하지만, C 수준 함수에는 그렇지 않아서 C 코드는 모든 파이썬 코드보다 빨라 보입니다.

27.5.2 즉석 사용자 설명서

이 섹션은 "설명서를 읽고 싶지 않은" 사용자를 위해 제공됩니다. 매우 간단한 개요를 제공하며, 사용자가 기존 응용 프로그램에서 프로파일링을 빠르게 수행할 수 있도록 합니다.

단일 인자를 취하는 함수를 프로파일링하려면, 이렇게 할 수 있습니다:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

(시스템에서 *cProfile*을 사용할 수 없으면 대신 *profile*을 사용하십시오.)

위의 작업은 *re.compile()*을 실행하고 다음과 같은 프로파일 결과를 인쇄합니다:

```
214 function calls (207 primitive calls) in 0.002 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.002    0.002 {built-in method builtins.exec}
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 __init__.py:250(compile)
     1    0.000    0.000    0.001    0.001 __init__.py:289(_compile)
     1    0.000    0.000    0.000    0.000 _compiler.py:759(compile)
     1    0.000    0.000    0.000    0.000 _parser.py:937(parse)
     1    0.000    0.000    0.000    0.000 _compiler.py:598(_code)
     1    0.000    0.000    0.000    0.000 _parser.py:435(_parse_sub)
```

첫 번째 줄은 214개의 호출이 관찰되었음을 나타냅니다. 이 호출 중 207개는 프리미티브(*primitive*)였으며, 이는 호출이 재귀를 통해 유발되지 않았음을 의미합니다. 다음 줄: `Ordered by: cumulative time`, 은 출력이 `cumtime` 값으로 정렬되었음을 나타냅니다. 열 제목은 다음과 같습니다:

ncalls
: 호출 수.

tottime
: 주어진 함수에서 소비된 총 시간 (서브 함수 호출에 든 시간은 제외합니다)

percall
tottime을 ncalls로 나눈 몫

cumtime
이 함수와 모든 서브 함수에서 소요된 누적 시간 (호출에서 종료까지). 이 수치는 재귀 함수에서도 정확합니다.

percall
cumtime을 프리미티브 호출로 나눈 몫

filename:lineno(function)
각 함수의 해당 데이터를 제공합니다 – 파일명:줄 번호(함수)

첫 번째 열에 두 개의 숫자가 있으면 (예를 들어 3/1), 함수가 재귀 되었음을 의미합니다. 두 번째 값은 프리미티브 호출 수이고 앞엣것은 총 호출 수입니다. 함수가 재귀 되지 않으면, 이 두 값은 같으며, 한 숫자만 인쇄됩니다.

프로파일 실행의 끝에 출력을 인쇄하는 대신, run() 함수에 파일명을 지정하여 결과를 파일에 저장할 수 있습니다:

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

pstats.Stats 클래스는 파일에서 프로파일 결과를 읽고 다양한 방식으로 포맷합니다.

*cProfile*과 *profile*을 스크립트로 호출하여 다른 스크립트를 프로파일링 할 수도 있습니다. 예를 들면:

```
python -m cProfile [-o output_file] [-s sort_order] (-m module | myscript.py)
```

-o <output_file>
stdout 대신 파일에 프로파일 결과를 씁니다.

-s <sort_order>
출력을 정렬할 *sort_stats()* 정렬 값 중 하나를 지정합니다. 이는 *-o*가 제공되지 않은 경우에만 적용됩니다.

-m <module>
스크립트 대신 모듈이 프로파일링 되도록 지정합니다.

Added in version 3.7: -m 옵션을 *cProfile*에 추가했습니다.

Added in version 3.8: -m 옵션을 *profile*에 추가했습니다.

pstats 모듈의 *Stats* 클래스에는 프로파일 결과 파일에 저장된 데이터를 조작하고 인쇄하기 위한 다양한 메서드가 있습니다:

```
import pstats
from pstats import SortKey
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

strip_dirs() 메서드는 모든 모듈 이름에서 외부 경로를 제거했습니다. *sort_stats()* 메서드는 인쇄되는 표준 모듈/줄/이름 문자열에 따라 모든 항목을 정렬했습니다. *print_stats()* 메서드는 모든 통계를 인쇄했습니다. 다음과 같은 정렬 호출을 시도할 수 있습니다:

```
p.sort_stats(SortKey.NAME)
p.print_stats()
```

첫 번째 호출은 실제로 함수 이름으로 목록을 정렬하고, 두 번째 호출은 통계를 인쇄합니다. 다음은 몇 가지 흥미로운 실험 호출입니다:

```
p.sort_stats(SortKey.CUMULATIVE).print_stats(10)
```

이것은 함수에서의 누적 시간을 기준으로 프로파일을 정렬한 다음, 가장 중요한 10개의 줄만 인쇄합니다. 시간이 걸리는 알고리즘을 이해하려면, 위의 줄을 사용하십시오.

어떤 함수가 많이 반복되고 많은 시간이 걸리는지 알고 싶다면, 다음을 수행하여:

```
p.sort_stats(SortKey.TIME).print_stats(10)
```

각 함수 내에서 소비한 시간에 따라 정렬한 다음, 상위 10개 함수에 대한 통계를 인쇄하십시오.

다음과 같은 것도 시도해 볼 수 있습니다:

```
p.sort_stats(SortKey.FILENAME).print_stats('__init__')
```

이렇게 하면 모든 통계가 파일 이름으로 정렬된 다음, 클래스 초기화(init) 메서드에 대한 통계만 인쇄됩니다 (이들의 철자가 `__init__`이기 때문입니다). 마지막 예로, 다음을 시도해 볼 수 있습니다:

```
p.sort_stats(SortKey.TIME, SortKey.CUMULATIVE).print_stats(.5, 'init')
```

이 줄은 주 시간 키와 누적 시간 보조 키로 통계를 정렬한 다음, 일부 통계를 인쇄합니다. 구체적으로, 목록을 먼저 원래 크기의 50%(`.5`)로 줄인 다음, `init`를 포함하는 줄만 유지되고, 그 서브 서브 목록이 인쇄됩니다.

어떤 함수가 위의 함수를 호출했는지 궁금하다면, 이제 다음과 같이 할 수 있습니다 (`p`는 여전히 마지막 기준에 따라 정렬됩니다):

```
p.print_callers(.5, 'init')
```

그러면 나열된 각 함수에 대한 호출자 목록을 얻습니다.

더 많은 기능을 원하면, 매뉴얼을 읽거나, 다음 함수가 무엇인지 추측하십시오:

```
p.print_callees()
p.add('restats')
```

스크립트로 호출될 때, *pstats* 모듈은 프로파일 덤프를 읽고 검사하기 위한 통계 브라우저입니다. 간단한 줄 지향 인터페이스(*cmd*를 사용하여 구현되었습니다)와 대화식 도움말이 있습니다.

27.5.3 `profile`과 `cProfile` 모듈 레퍼런스

*profile*과 *cProfile* 모듈은 모두 다음 함수를 제공합니다:

`profile.`**`run`**(*command*, *filename=None*, *sort=-1*)

이 함수는 *exec()* 함수에 전달할 수 있는 단일 인자와 선택적 파일 이름을 취합니다. 모든 경우에 이 루틴은 다음을 실행합니다:

```
exec(command, __main__.__dict__, __main__.__dict__)
```

그리고 실행으로부터 프로파일링 통계를 수집합니다. 파일 이름이 없으면, 이 함수는 자동으로 *Stats* 인스턴스를 만들고 간단한 프로파일링 보고서를 인쇄합니다. 정렬 값이 지정되면, 이 *Stats* 인스턴스로 전달되어 결과 정렬 방법을 제어합니다.

`profile.`**`runctx`**(*command*, *globals*, *locals*, *filename=None*, *sort=-1*)

이 함수는 *run()*과 유사하며, *command* 문자열에 대한 전역(globals)과 지역(locals) 매핑 제공하기 위한 인자가 추가되었습니다. 이 루틴은 다음을 실행합니다:

```
exec(command, globals, locals)
```

그리고 위의 *run()* 함수에서와같이 프로파일링 통계를 수집합니다.

class `profile.`**`Profile`**(*timer=None*, *timeunit=0.0*, *subcalls=True*, *builtins=True*)

이 클래스는 일반적으로 `cProfile.run()` 함수가 제공하는 것보다 프로파일링에 대한 더 세밀한 제어가 필요할 때만 사용됩니다.

timer 인자를 통해 코드를 실행하는 데 걸리는 시간을 측정하기 위한 사용자 정의 타이머를 제공할 수 있습니다. 현재 시각을 나타내는 단일 숫자를 반환하는 함수여야 합니다. 숫자가 정수이면, *timeunit* 는 각 시간 단위의 지속 시간을 지정하는 승수를 지정합니다. 예를 들어, 타이머가 밀리초 단위로 측정된 시간을 반환하면 시간 단위는 `.001` 입니다.

`Profile` 클래스를 직접 사용하면 프로파일 데이터를 파일에 쓰지 않고도 프로파일 결과를 포맷할 수 있습니다:

```
import cProfile, pstats, io
from pstats import SortKey
pr = cProfile.Profile()
pr.enable()
# ... 뭔가 합니다 ...
pr.disable()
s = io.StringIO()
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

`Profile` 클래스는 컨텍스트 관리자로도 사용될 수 있습니다 (`cProfile` 모듈에서만 지원됩니다. 컨텍스트 관리자 형을 참조하십시오):

```
import cProfile

with cProfile.Profile() as pr:
    # ... 뭔가 합니다 ...

    pr.print_stats()
```

버전 3.8에서 변경: 컨텍스트 관리자 지원이 추가되었습니다.

enable()

프로파일링 데이터 수집을 시작합니다. `cProfile`에만 있습니다.

disable()

프로파일링 데이터 수집을 중지합니다. `cProfile`에만 있습니다.

create_stats()

프로파일링 데이터 수집을 중지하고 결과를 내부적으로 현재 프로파일로 기록합니다.

print_stats(*sort=-1*)

현재 프로파일을 기반으로 `Stats` 객체를 만들고 결과를 stdout에 인쇄합니다.

The *sort* parameter specifies the sorting order of the displayed statistics. It accepts a single key or a tuple of keys to enable multi-level sorting, as in `Stats.sort_stats`.

Added in version 3.13: `print_stats()` now accepts a tuple of keys.

dump_stats(*filename*)

현재 프로파일의 결과를 *filename*에 씁니다.

run(*cmd*)

`exec()`를 통해 cmd를 프로파일 합니다.

runctx(*cmd*, *globals*, *locals*)

지정된 전역과 지역 환경으로 `exec()`를 통해 cmd를 프로파일 합니다.

runcall(*func*, */*, **args*, ***kwargs*)

`func(*args, **kwargs)`를 프로파일 합니다

프로파일링은 호출된 명령/함수가 실제로 반환하는 경우에만 작동함에 유의하십시오. 인터프리터가 종료되면 (예를 들어 호출된 명령/함수 실행 중 `sys.exit()` 호출을 통해) 아무런 프로파일링 결과도 인쇄되지 않습니다.

27.5.4 `Stats` 클래스

프로파일러 데이터의 분석은 `Stats` 클래스를 사용하여 수행됩니다.

class `pstats.`**`Stats`**(**filenames or profile, stream=sys.stdout*)

이 클래스 생성자는 *filename*(또는 파일명의 리스트)이나 `Profile` 인스턴스에서 "통계 객체"의 인스턴스를 만듭니다. 출력은 *stream*에 의해 지정된 스트림으로 인쇄됩니다.

위의 생성자에 의해 선택된 파일은 해당 버전의 `profile`이나 `cProfile`에 의해 만들어졌어야 합니다. 구체적으로, 이 프로파일러의 향후 버전에서 보장되는 파일 호환성은 없으며, 다른 프로파일러에서 생성된 파일이나 다른 운영 체제에서 실행되는 같은 프로파일러의 실행과 호환되지 않습니다. 여러 파일이 제공되면, 동일한 함수에 대한 모든 통계가 통합되므로, 여러 프로세스에 대한 전체 뷰를 단일 보고서에서 고려할 수 있습니다. 추가 파일을 기존 `Stats` 객체의 데이터와 결합해야 하면, `add()` 메서드를 사용할 수 있습니다.

파일에서 프로파일 데이터를 읽는 대신, `cProfile.Profile`이나 `profile.Profile` 객체를 프로파일 데이터 소스로 사용할 수 있습니다.

`Stats` 객체에는 다음과 같은 메서드가 있습니다:

strip_dirs()

`Stats` 클래스에 대한 이 메서드는 파일 이름에서 모든 선행 경로 정보를 제거합니다. 80열 이내에 (가깝게) 맞게 출력물의 크기를 줄이는 데 매우 유용합니다. 이 메서드는 객체를 수정하고, 제거된 정보는 손실됩니다. 제거 조작을 수행한 후, 객체는 객체 초기화와 로드 직후와 마찬가지로 "임의의" 순서로 항목을 가진 것으로 간주합니다. `strip_dirs()`로 인해 두 함수 이름이 구별할 수 없게 되면 (같은 파일 이름의 같은 줄에 있고, 함수 이름도 같습니다), 이 두 항목에 대한 통계는 단일 항목으로 누적됩니다.

add(**filenames*)

`Stats` 클래스의 이 메서드는 추가 프로파일링 정보를 현재 프로파일링 객체에 누적합니다. 인자는 해당 버전의 `profile.run()`이나 `cProfile.run()`으로 만들어진 파일명을 참조해야 합니다. 동일한 이름을 가진 (파일, 줄, 이름) 함수에 대한 통계는 자동으로 단일 함수 통계에 축적됩니다.

dump_stats(*filename*)

`Stats` 객체에 로드된 데이터를 *filename*이라는 파일에 저장합니다. 파일이 없으면 만들어지고, 이미 존재하면 덮어씁니다. 이것은 `profile.Profile`과 `cProfile.Profile` 클래스에 있는 같은 이름의 메서드와 동등합니다.

sort_stats(**keys*)

이 메서드는 제공된 기준에 따라 `Stats` 객체를 정렬하여 수정합니다. 인자는 정렬 기준을 식별하는 문자열이나 SortKey 열거형일 수 있습니다 (예: `'time'`, `'name'`, `SortKey.TIME` 또는 `SortKey.NAME`). SortKey 열거형 인자는 문자열 인자보다 안정적이고 에러가 적다는 점에서 문자열 인자보다 유리합니다.

둘 이상의 키가 제공되면, 그 앞에 선택된 모든 키가 같을 때 추가 키가 보조 기준으로 사용됩니다. 예를 들어, `sort_stats(SortKey.NAME, SortKey.FILE)`은 함수 이름에 따라 모든 항목을 정렬하고, 함수 이름이 같으면 파일 이름으로 정렬합니다.

문자열 인자의 경우, 약어가 모호하지 않은 한, 모든 키 이름에 약어를 사용할 수 있습니다.

유효한 문자열과 SortKey는 다음과 같습니다:

유효한 문자열 인자	유효한 열거형 인자	의미
'calls'	SortKey.CALLS	호출 수
'cumulative'	SortKey.CUMULATIVE	누적 시간
'cumtime'	해당 없음	누적 시간
'file'	해당 없음	파일 이름
'filename'	SortKey.FILENAME	파일 이름
'module'	해당 없음	파일 이름
'ncalls'	해당 없음	호출 수
'pcalls'	SortKey.PCALLS	프리미티브 호출 수
'line'	SortKey.LINE	줄 번호
'name'	SortKey.NAME	함수 이름
'nfl'	SortKey.NFL	이름/파일/줄
'stdname'	SortKey.STDNAME	표준 이름
'time'	SortKey.TIME	내부 시간
'tottime'	해당 없음	내부 시간

통계의 모든 정렬은 내림차순이고 (가장 시간이 오래 걸리는 항목을 앞에 놓습니다), 이름, 파일 및 줄 번호 검색은 오름차순(알파벳 순서) 임에 유의하십시오. `SortKey.NFL`과 `SortKey.STDNAME` 간의 미묘한 차이점은 표준 이름이 인쇄된 이름의 일종이라는 것입니다. 즉, 포함된 줄 번호가 이상한 방식으로 비교됩니다. 예를 들어, 줄 3, 20 및 40은 (파일 이름이 같으면) 문자열 순서 20, 3 및 40으로 나타납니다. 반면에 `SortKey.NFL`은 줄 번호를 숫자로 비교합니다. 실제로, `sort_stats(SortKey.NFL)` 은 `sort_stats(SortKey.NAME, SortKey.FILENAME, SortKey.LINE)` 과 같습니다.

이전 버전과의 호환성을 위해, 숫자 인자 `-1`, `0`, `1` 및 `2`가 허용됩니다. 이들은 각각 `'stdname'`, `'calls'`, `'time'` 및 `'cumulative'`로 해석됩니다. 이 이전 스타일 형식(숫자)을 사용하면, 오직 하나의 정렬 키(숫자키)만 사용되며, 추가 인자는 조용히 무시됩니다.

Added in version 3.7: SortKey 열거형을 추가했습니다.

reverse_order()

Stats 클래스에 대한 이 메서드는 객체 내 기본 리스트의 순서를 뒤집습니다. 기본적으로 오름차순 대 내림차순은 선택한 정렬 키에 따라 올바르게 선택됨에 유의하십시오.

print_stats(**restrictions*)

Stats 클래스에 대한 이 메서드는 *profile.run()* 정의에 설명된 대로 보고서를 인쇄합니다.

인쇄 순서는 객체에서 수행된 마지막 *sort_stats()* 연산을 기반으로 합니다 (*add()*와 *strip_dirs()*의 경고가 적용됩니다).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a string that will be interpreted as a regular expression (to pattern match the standard name that is printed). If several restrictions are provided, then they are applied sequentially. For example:

```
print_stats(.1, 'foo:')
```

는 먼저 인쇄를 목록의 처음 10%로 제한한 다음, 파일 이름 `.*foo:`의 일부인 함수만 인쇄합니다. 대조적으로, 명령:

```
print_stats('foo:', .1)
```

은 파일 이름이 `.*foo:` 인 모든 함수로 목록을 제한한 다음, 그중 처음 10%만 인쇄합니다.

print_callers(**restrictions*)

Stats 클래스에 대한 이 메서드는 프로파일 된 데이터베이스에 있는 각 함수를 호출한 모든 함수의 목록을 인쇄합니다. 순서는 *print_stats()*에서 제공한 순서와 동일하며, 제한 인자의 정의도 동일합니다. 각 호출자는 개별 줄로 보고됩니다. 통계를 생성한 프로파일러에 따라 형식이 약간 다릅니다:

- `profile`을 사용하면, 각 호출자 뒤에 숫자가 괄호 안에 표시되어 이 특정 호출이 몇 번이나 되었는지 표시됩니다. 편의를 위해, 두 번째 괄호로 묶지 않은 숫자는 오른쪽에 나오는 함수에서 소비한 누적 시간을 반복합니다.
- `cProfile`을 사용하면, 각 호출자 앞에 세 숫자가 나옵니다: 이 특정 호출이 발생한 횟수, 이 특정 호출자가 호출한 동안 현재 함수에서 소비 한 총 및 누적 시간.

print_callees(**restrictions*)

`Stats` 클래스에 대한 이 메서드는 표시된 함수에 의해 호출된 모든 함수의 목록을 인쇄합니다. 이러한 호출 방향 반전(호출한 대 호출된)을 제외하고, 인자와 순서는 `print_callers()` 메서드와 같습니다.

get_stats_profile()

이 메서드는 함수 이름을 FunctionProfile 인스턴스로 매핑하는 StatsProfile 인스턴스를 반환합니다. 각 FunctionProfile 인스턴스에는 함수 실행 시간, 호출 횟수 등의 함수의 프로파일 관련 정보가 있습니다.

Added in version 3.9: 다음과 같은 데이터 클래스(dataclasses)를 추가했습니다: StatsProfile, FunctionProfile. 다음과 같은 함수를 추가했습니다: get_stats_profile.

27.5.5 결정론적 프로파일링이란 무엇입니까?

결정론적 프로파일링(*Deterministic profiling*)이라는 용어는 모든 함수 호출, 함수 반환 및 예외 이벤트가 모니터링되고, (사용자 코드가 실행되는 시간 동안) 이러한 이벤트 사이의 간격에 대한 정확한 시간 측정이 이루어진다는 사실을 반영하기 위한 것입니다. 반면에, 통계적 프로파일링(*statistical profiling*)(이 모듈에서는 수행하지 않습니다)은 유효 명령어 포인터를 무작위로 샘플링하여 시간이 소비되는 위치를 추론합니다. 후자의 기술은 전통적으로 (코드를 계측할 필요가 없기 때문에) 오버헤드가 적지만, 시간이 어디에서 소비되는지에 대한 상대적 표시만 제공합니다.

파이썬에서는, 실행 중에 인터프리터가 활성화되어있어서, 결정론적 프로파일링을 수행하기 위해 인스트루먼트 된 코드(instrumented code)가 필요하지 않습니다. 파이썬은 각 이벤트에 대해 자동으로 훅(*hook*)(선택적 콜백)을 제공합니다. 또한, 파이썬의 인터프리터 적인 성격은 실행에 이미 많은 오버헤드를 추가하는 경향이 있어서, 결정론적 프로파일링은 일반적인 응용 프로그램에서 작은 처리 오버헤드만 추가하는 경향이 있습니다. 결과적으로 결정론적 프로파일링은 그다지 비싸지 않으면서도, 파이썬 프로그램의 실행에 대한 광범위한 실행 시간 통계를 제공합니다.

호출 수 통계를 사용하여 코드의 버그(놀랄만한 횟수)를 식별하고, 가능한 인라인 확장 지점(높은 호출 횟수)을 식별할 수 있습니다. 내부 시간 통계를 사용하여 신중하게 최적화해야 하는 "핫 루프(hot loops)"를 식별할 수 있습니다. 누적 시간 통계를 사용하여 알고리즘 선택에서의 고수준 에러를 식별할 수 있습니다. 이 프로파일러에서의 누적 시간의 특이한 처리는 알고리즘의 재귀 구현에 대한 통계를 반복 구현과 직접 비교할 수 있도록 함에 유의하십시오.

27.5.6 한계

타이밍 정보의 정확성과 관련하여 한 가지 제약이 있습니다. 정확성과 관련해서는 결정론적 프로파일러에 근본적인 문제가 있습니다. 가장 명백한 제약은 하부 "시계"가 (일반적으로) 약 .001 초의 속도로만 눈금이 변하는 것입니다. 따라서 하부 시계보다 더 정확한 측정은 없습니다. 충분한 측정을 수행하면, "에러"가 평균이 되어 사라지는 경향이 있습니다. 불행히도, 이 첫 번째 에러를 제거하면 두 번째 에러 원인이 발생합니다.

두 번째 문제는 이벤트가 디스패치 된 시점부터 프로파일러가 시간을 얻기 위해 호출하는 것이 실제로 시계의 상태를 얻기까지 "시간이 걸린다"는 것입니다. 마찬가지로, 프로파일러 이벤트 핸들러를 빠져나갈 때 시계값이 획득된 (그런 다음 저장됩니다) 시간부터, 사용자의 코드가 다시 실행될 때까지 어떤 지연이 발생합니다. 결과적으로, 여러 번 호출되거나, 많은 함수를 호출하는 함수는 일반적으로 이 에러를 누적합니다. 이러한 방식으로 누적되는 에러는 일반적으로 시계 정확도(1 눈금 미만)보다 작지만, 누적될 수 있어서 매우 중요해집니다.

오버헤드가 낮은 `cProfile`보다 `profile`에서 이 문제가 더 중요합니다. 이러한 이유로, `profile`은 특정 플랫폼에 대해 자신을 보정하는 방법을 제공하여 이 에러를 확률적으로 (평균적으로) 제거할 수 있습니다. 프로파일러가 보정된 후에는, 더 정확하지만 (최소한 최소 자승의 의미에서), 때로는 음수를 생성합니다 (호출 횟수가 예외적으로 낮고, 확률의 신들이 당신에게 등을 돌리면 :-).) 프로파일에서 음수를 보아도

너무 놀라지 마십시오. 프로파일러를 보정한 경우에*만* 나타나야 하며, 결과는 실제로 보정하지 않은 것보다 낫습니다.

27.5.7 보정

profile 모듈의 프로파일러는 각 이벤트 처리 시간에서 상수를 빼서 시간 함수 호출의 오버헤드를 보상하고, 결과를 잘 보관합니다. 기본적으로, 상수는 0입니다. 다음 절차는 주어진 플랫폼에 대해 더 나은 상수를 얻는 데 사용될 수 있습니다 (한계를 참조하십시오).

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

이 메서드는 인자가 제공한 횟수의 파이썬 호출을, 직접하고 프로파일러하에서 다시 하면서, 두 두 시간을 측정합니다. 그런 다음 프로파일러 이벤트 당 숨겨진 오버헤드를 계산하여 부동 소수점으로 반환합니다. 예를 들어, 맥 OS를 실행하는 1.8Ghz 인텔 코어 i5에서, 그리고 파이썬의 time.process_time()을 타이머로 사용할 때, 매직 넘버는 약 4.04e-6입니다.

이 반복의 목적은 상당히 일관된 결과를 얻는 것입니다. 컴퓨터가 매우 빠르거나, 타이머 함수의 해상도가 좋지 않으면, 일관된 결과를 얻기 위해 100000이나 심지어 1000000을 전달해야 할 수 있습니다.

일관된 대답을 얻었을 때, 세 가지 방법으로 사용할 수 있습니다:

```
import profile

# 1. 이후에 만들어지는 모든 Profile 인스턴스에 계산된 바이어스를 적용합니다.
profile.Profile.bias = your_computed_bias

# 2. 특정 Profile 인스턴스에 계산된 바이어스를 적용합니다.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. 인스턴스 생성자에 계산된 바이어스를 지정합니다.
pr = profile.Profile(bias=your_computed_bias)
```

선택해야 한다면, 더 작은 상수를 선택하는 것이 좋습니다, 그러면 결과가 프로파일 통계에서 결과가 "덜 자주" 음수로 표시됩니다.

27.5.8 사용자 정의 타이머 사용하기

현재 시각을 결정하는 방법을 변경하려면 (예를 들어, 벽시계 시간이나 소요된 프로세스 시간을 사용하도록 만들려면), `Profile` 클래스 생성자에게 원하는 타이밍 함수를 전달합니다:

```
pr = profile.Profile(your_time_func)
```

결과 프로파일러는 `your_time_func`를 호출합니다. *profile.Profile*이나 `cProfile.Profile` 중 어느 것을 사용하느냐에 따라, `your_time_func`의 반환 값은 다르게 해석됩니다:

profile.Profile

`your_time_func`는 단일 숫자를 반환하거나, 또는 (*os.times()*가 반환하는 것과 같이) 합계가 현재 시각인 숫자 리스트를 반환해야 합니다. 함수가 단일 시간 숫자를 반환하거나, 반환된 숫자의 리스트 길이가 2이면, 특히 빠른 버전의 디스패치 루틴을 얻게 됩니다.

여러분이 선택한 타이머 함수에 대해 프로파일러 클래스를 보정해야 합니다 (보정을 참조하십시오). 대부분의 기계에서, 단일 정숫값을 반환하는 타이머는 프로파일링 중 낮은 오버헤드 측면에서 최상의 결과를 제공합니다. (*os.times()*는 부동 소수점 값의 튜플을 반환하므로 꽤 나쁩니다). 더 좋은 타이머를 가장 깨끗한 방식으로 대체하려면, 클래스를 파생하고 적절한 보정 상수와 함께 타이머 호출을 가장 잘 처리하는 대체 디스패치 메서드를 배선하십시오.

cProfile.Profile

your_time_func는 단일 숫자를 반환해야 합니다. 정수를 반환하면, 시간 단위의 실제 지속 시간을 지정하는 두 번째 인자로 클래스 생성자를 호출할 수도 있습니다. 예를 들어, your_integer_time_func가 밀리초 단위로 측정된 시간을 반환하면, 다음과 같이 Profile 인스턴스를 구성합니다:

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

cProfile.Profile 클래스를 보정할 수 없어서, 사용자 정의 타이머 함수는 주의해서 사용해야 하고 가능한 한 빨라야 합니다. 사용자 정의 타이머로 최상의 결과를 얻으려면, 내부 _lsprof 모듈의 C 소스에 하드 코딩해야 할 수도 있습니다.

파이썬 3.3은 *time*에 프로세스나 벽시계 시간을 정확하게 측정하는 데 사용할 수 있는 몇 가지 새로운 함수를 추가합니다. 예를 들어, *time.perf_counter()*를 참조하십시오.

27.6 timeit — 작은 코드 조각의 실행 시간 측정

소스 코드: Lib/timeit.py

이 모듈은 파이썬 코드의 작은 조각의 시간을 측정하는 간단한 방법을 제공합니다. 명령 줄 인터페이스뿐만 아니라 콜러블도 있습니다. 실행 시간을 측정에 따르는 흔한 함정들을 피할 수 있습니다. O' Reilly가 출판한 *Python Cookbook* 2판에 있는 Tim Peters의 "Algorithms" 장의 개요도 참조하십시오.

27.6.1 기본 예제

다음 예제에서는 명령 줄 인터페이스를 사용하여 세 가지 다른 표현식을 비교하는 방법을 보여줍니다:

```
$ python -m timeit "'-'.join(str(n) for n in range(100))"
10000 loops, best of 5: 30.2 usec per loop
$ python -m timeit "'-'.join([str(n) for n in range(100)])"
10000 loops, best of 5: 27.5 usec per loop
$ python -m timeit "'-'.join(map(str, range(100)))"
10000 loops, best of 5: 23.2 usec per loop
```

이것은 파이썬 인터페이스로는 다음과 같이 할 수 있습니다:

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
```

콜러블을 파이썬 인터페이스로 전달할 수도 있습니다:

```
>>> timeit.timeit(lambda: "-".join(map(str, range(100))), number=10000)
0.19665591977536678
```

그러나 *timeit()*은 명령 줄 인터페이스가 사용될 때만 반복 횟수를 자동으로 결정합니다. 예제 절에서 고급 예제를 찾을 수 있습니다.

27.6.2 파이썬 인터페이스

이 모듈은 세 개의 편리 함수와 하나의 공용 클래스를 정의합니다:

timeit.**timeit**(*stmt='pass', setup='pass', timer=<default timer>, number=1000000, globals=None*)

지정된 문장, *setup* 코드 및 *timer* 함수로 `Timer` 인스턴스를 만들고, *number* 실행으로 `timeit()` 메서드를 실행합니다. 선택적 *globals* 인자는 코드를 실행할 이름 공간을 지정합니다.

버전 3.5에서 변경: 선택적 *globals* 매개 변수가 추가되었습니다.

timeit.**repeat**(*stmt='pass', setup='pass', timer=<default timer>, repeat=5, number=1000000, globals=None*)

주어진 문장, *setup* 코드 및 *timer* 함수로 `Timer` 인스턴스를 생성하고, 주어진 *repeat* 카운트와 *number* 실행으로 `repeat()` 메서드를 실행합니다. 선택적 *globals* 인자는 코드를 실행할 이름 공간을 지정합니다.

버전 3.5에서 변경: 선택적 *globals* 매개 변수가 추가되었습니다.

버전 3.7에서 변경: *repeat*의 기본값이 3에서 5로 변경되었습니다.

timeit.**default_timer**()

기본 타이머, 항상 time.perf_counter() 이고, 초 단위의 float를 반환합니다. 대안인 time.perf_counter_ns 는 나노초 단위의 정수를 반환합니다.

버전 3.3에서 변경: 이제 `time.perf_counter()`가 기본 타이머입니다.

class timeit.**Timer**(*stmt='pass', setup='pass', timer=<timer function>, globals=None*)

작은 코드 조각의 실행 속도를 측정하기 위한 클래스.

생성자는 시간 측정될 문장, 설정에 사용되는 추가 문장 및 타이머 함수를 받아들입니다. 두 문장의 기본값은 `'pass'`입니다; 타이머 함수는 플랫폼에 따라 다릅니다 (모듈 독스트링을 참조하십시오). *stmt*와 *setup*은 여러 줄에 걸친 문자열 리터럴을 포함하지 않는 한 ; 나 줄 바꿈으로 구분된 여러 개의 문장을 포함 할 수도 있습니다. 문장은 기본적으로 timeit의 이름 공간 내에서 실행됩니다; 이 동작은 *globals*에 이름 공간을 전달하여 제어 할 수 있습니다.

첫 번째 문장의 실행 시간을 측정하려면, `timeit()` 메서드를 사용하십시오. `repeat()`와 `autorange()` 메서드는 `timeit()`을 여러 번 호출하는 편리 메서드입니다.

*setup*의 실행 시간은 전체 측정 실행 시간에서 제외됩니다.

*stmt*와 *setup* 매개 변수는 인자 없이 호출 할 수 있는 객체를 받아들일 수도 있습니다. 이렇게 하면 `timeit()`에 의해 실행될 타이머 함수에 그들에 대한 호출을 포함시킵니다. 이때는 여분의 함수 호출로 인해 타이밍 오버헤드가 약간 더 커집니다.

버전 3.5에서 변경: 선택적 *globals* 매개 변수가 추가되었습니다.

timeit(*number=1000000*)

주 문장의 *number* 실행의 시간을 측정합니다. setup 문장을 한 번 실행한 다음, 주 문장을 여러 번 실행하는 데 걸리는 시간을 반환합니다. 기본 타이머는 초 단위의 float를 반환합니다. 인자는 루프를 통과하는 횟수이며, 기본값은 백만입니다. 주 문장, setup 문장 및 사용할 타이머 함수는 생성자에 전달됩니다.

> **참고**
>
> 기본적으로, `timeit()`은 시간 측정 중에 가비지 수거를 일시적으로 끕니다. 이 방법의 장점은 독립적인 시간 측정이 더 잘 비교될 수 있다는 것입니다. 단점은 GC가 측정되는 함수의 성능에서 중요한 요소가 될 수 있다는 것입니다. 그렇다면, GC를 *setup* 문자열의 첫 번째 문장에서 다시 활성화할 수 있습니다. 예를 들면:
>
> ```
> timeit.Timer('for i in range(10): oct(i)', 'gc.enable()').timeit()
> ```

autorange(*callback=None*)

`timeit()`를 호출하는 횟수를 자동으로 결정합니다.

이 함수는 총 시간이 0.2초 이상이 될 때까지 `timeit()`을 반복적으로 호출하고, 최종 (루프 수, 해당 루프 수에 소요된 시간)을 반환하는 편리 함수입니다. 실행 시간이 적어도 0.2초가 될 때까지 1, 2, 5, 10, 20, 50 … 로 루프 수를 증가시키면서 `timeit()`을 호출합니다.

callback 이 주어지고 None 이 아니면, 각 시도 다음에 두 개의 인자로 호출합니다: `callback(number, time_taken)`.

Added in version 3.6.

repeat(*repeat=5, number=1000000*)

`timeit()` 을 몇 번 호출합니다.

이것은 반복적으로 `timeit()` 을 호출하여 결과 리스트를 반환하는 편리 함수입니다. 첫 번째 인자는 `timeit()` 을 호출할 횟수를 지정합니다. 두 번째 인자는 `timeit()` 에 대한 *number* 인자를 지정합니다.

> **참고**
>
> 결과 벡터로부터 평균과 표준 편차를 계산하고 이를 보고하고 싶을 수 있습니다. 하지만, 이것은 별로 유용하지 않습니다. 일반적으로, 가장 낮은 값이 여러분의 기계가 주어진 코드 조각을 얼마나 빨리 실행할 수 있는지에 대한 하한값을 제공합니다; 결과 벡터의 더 높은 값은 일반적으로 파이썬의 속도 변동성 때문이 아니라, 시간 측정의 정확성을 방해하는 다른 프로세스에 의해 발생합니다. 따라서 결과의 `min()` 이 여러분이 관심을 기울여야 할 유일한 숫자일 것입니다. 그 후에, 전체 벡터를 살펴보고 통계보다는 상식을 적용해야 합니다.

버전 3.7에서 변경: *repeat* 의 기본값이 3에서 5로 변경되었습니다.

print_exc(*file=None*)

측정되는 코드로부터의 트레이스백을 인쇄하는 도우미.

일반적인 사용:

```
t = Timer(...)        # try/except 바깥
try:
    t.timeit(...)     # 또는 t.repeat(...)
except Exception:
    t.print_exc()
```

표준 트레이스백에 비해 장점은 컴파일된 템플릿의 소스 행이 표시된다는 것입니다. 선택적 *file* 인자는 트레이스백을 보내야 할 곳을 지시합니다; 기본값은 `sys.stderr` 입니다.

27.6.3 명령 줄 인터페이스

명령 줄에서 프로그램으로 호출할 때, 다음 형식이 사용됩니다:

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-p] [-v] [-h] [statement ...]
```

이때 다음 옵션을 지원합니다:

-n N, **--number**=N

'statement' 를 실행하는 횟수

-r N, **--repeat**=N

타이머 반복 횟수 (기본값 5)

-s S, **--setup**=S

초기에 한 번 실행될 문장 (기본값 pass)

-p, **--process**

벽시계 시간이 아니라 프로세스 시간을 측정합니다, 기본값인 `time.perf_counter()` 대신 `time.process_time()` 을 사용합니다

Added in version 3.3.

-u, --unit=U

타이머 출력의 시간 단위를 지정합니다; nsec, usec, msec 또는 sec 중에서 선택할 수 있습니다.

Added in version 3.5.

-v, --verbose

날 시간 측정 결과를 인쇄합니다; 더 많은 자릿수 정밀도를 위해서는 반복하십시오

-h, --help

짧은 사용법 메시지를 출력하고 종료합니다

여러 줄의 문장은 각 줄을 별도의 statement 인자로 지정하여 제공할 수 있습니다; 들여쓰기 된 줄은 인자를 따옴표로 묶고 선행 공백을 사용하면 됩니다. 여러 개의 *-s* 옵션도 비슷하게 취급됩니다.

*-n*이 주어지지 않으면, 총 시간이 최소 0.2초가 될 때까지 시퀀스 1, 2, 5, 10, 20, 50, … 에서 증가하는 숫자를 시도하여 적절한 루프 수가 계산됩니다.

default_timer() 측정은 같은 기계에서 실행되는 다른 프로그램의 영향을 받을 수 있으므로, 정확한 시간 측정이 필요할 때 가장 좋은 방법은 시간 측정을 몇 번 반복하고 가장 좋은 시간을 사용하는 것입니다. 이 작업에는 *-r* 옵션이 좋습니다; 대부분의 경우 기본값인 5번 반복으로 충분할 것입니다. *time.process_time()*을 사용하여 CPU 시간을 측정 할 수 있습니다.

참고

pass 문을 실행하는 것과 관련된 어떤 기준 오버헤드가 있습니다. 여기에 있는 코드는 그것을 숨기려고 시도하지는 않지만, 여러분은 신경 써야 합니다. 기준 오버헤드는 인자 없이 프로그램을 호출하여 측정할 수 있으며, 파이썬 버전마다 다를 수 있습니다.

27.6.4 예제

처음에 한 번만 실행되는 setup 문을 제공 할 수 있습니다:

```
$ python -m timeit -s "text = 'sample string'; char = 'g'" "char in text"
5000000 loops, best of 5: 0.0877 usec per loop
$ python -m timeit -s "text = 'sample string'; char = 'g'" "text.find(char)"
1000000 loops, best of 5: 0.342 usec per loop
```

In the output, there are three fields. The loop count, which tells you how many times the statement body was run per timing loop repetition. The repetition count ('best of 5') which tells you how many times the timing loop was repeated, and finally the time the statement body took on average within the best repetition of the timing loop. That is, the time the fastest repetition took divided by the loop count.

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

Timer 클래스와 그 메서드를 사용하여 같은 작업을 수행 할 수 있습니다:

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40183617287970225, 0.37027556854118704, 0.38344867356679524, 0.3712595970846668,
↪ 0.37866875250654886]
```

다음 예제는 여러 줄을 포함하는 표현식의 시간을 측정하는 방법을 보여줍니다. 여기서 우리는 누락되었거나 존재하는 객체 어트리뷰트를 검사하는데 *hasattr()*과 try/except를 사용하는 비용을 비교합니다:

```
$ python -m timeit "try:" "  str.__bool__" "except AttributeError:" "  pass"
20000 loops, best of 5: 15.7 usec per loop
$ python -m timeit "if hasattr(str, '__bool__'): pass"
50000 loops, best of 5: 4.26 usec per loop

$ python -m timeit "try:" "  int.__bool__" "except AttributeError:" "  pass"
200000 loops, best of 5: 1.43 usec per loop
$ python -m timeit "if hasattr(int, '__bool__'): pass"
100000 loops, best of 5: 2.23 usec per loop
```

```
>>> import timeit
>>> # 어트리뷰트가 없습니다
>>> s = """\
... try:
...     str.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # 어트리뷰트가 있습니다
>>> s = """\
... try:
...     int.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

timeit 모듈이 여러분이 정의한 함수에 액세스하도록 하려면, import 문이 포함된 *setup* 매개 변수를 전달하면 됩니다:

```
def test():
    """멍청한 테스트 함수"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

또 다른 옵션은 *globals()*를 *globals* 매개 변수로 전달해서, 여러분의 현재 전역 이름 공간에서 코드가 실행되도록 하는 것입니다. 임포트를 개별적으로 지정하는 것보다 편리 할 수 있습니다:

```
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]', globals=globals()))
```

27.7 trace — 파이썬 문장 실행 추적

소스 코드: Lib/trace.py

---

trace 모듈을 사용하면 프로그램 실행을 추적하고, 주석 처리된 문장 커버리지 리스트를 생성하고, 호출자/피호출자 관계를 인쇄하고, 프로그램 실행 중 호출되는 함수를 나열할 수 있습니다. 다른 프로그램이나 명령 줄에서 사용할 수 있습니다.

더 보기

Coverage.py
브랜치 커버리지와 같은 고급 기능과 함께 HTML 출력을 제공하는 널리 사용되는 제삼자 커버리지 도구.

27.7.1 명령 줄 사용법

trace 모듈은 명령 줄에서 호출할 수 있습니다. 다음과 같이 간단할 수도 있습니다

```
python -m trace --count -C . somefile.py ...
```

이것은 `somefile.py`를 실행하고 실행 중에 임포트 한 모든 파이썬 모듈들의 주석이 달린 리스트를 현재 디렉터리에 생성합니다.

--help
사용법을 표시하고 종료합니다.

--version
모듈의 버전을 표시하고 종료합니다.

Added in version 3.8: 실행 가능한 모듈을 실행할 수 있는 `--module` 옵션이 추가되었습니다.

주요 옵션

*trace*를 호출할 때 다음 옵션 중 적어도 하나를 지정해야 합니다. *--listfuncs* 옵션은 *--trace* 나 *--count* 옵션과 함께 사용할 수 없습니다. *--listfuncs*이 제공되면, *--count* 나 *--trace*는 허용되지 않으며, 반대의 경우도 마찬가지입니다.

-c, --count
프로그램 완료 시 각 문장이 실행된 횟수를 보여주는 주석이 달린 리스팅 파일 집합을 생성합니다. 아래의 *--coverdir*, *--file* 및 *--no-report*도 참조하십시오.

-t, --trace
줄이 실행될 때마다 표시합니다.

-l, --listfuncs
프로그램을 실행하여 실행되는 함수들을 표시합니다.

-r, --report
--count 와 *--file* 옵션을 사용한 이전 프로그램 실행에서 주석이 달린 목록을 생성합니다. 이것은 어떤 코드도 실행하지 않습니다.

-T, --trackcalls
프로그램을 실행하여 노출된 호출 관계를 표시합니다.

수정자

-f, --file=<file>

여러 추적 실행에서 실행 횟수를 누적할 파일의 이름. *--count* 옵션과 함께 사용해야 합니다.

-C, --coverdir=<dir>

보고서 파일이 저장되는 디렉터리. `package.module`에 대한 커버리지 보고서는 *dir/package/module*`.cover` 파일에 기록됩니다.

-m, --missing

주석이 달린 리스팅을 작성할 때, `>>>>>>`로 실행되지 않은 줄을 표시합니다.

-s, --summary

--count 나 *--report*를 사용할 때, 처리된 각 파일에 대한 요약을 표준출력으로 기록합니다.

-R, --no-report

주석이 달린 리스팅을 생성하지 않습니다. 이것은 *--count*를 사용하여 여러 번 실행한 다음, 마지막에 주석이 달린 리스팅의 단일 집합을 생성하려고 할 때 유용합니다.

-g, --timing

프로그램이 시작된 이후의 시간을 각 줄 앞에 붙입니다. 추적하는 동안에만 사용됩니다.

필터

이 옵션들은 여러 번 반복 될 수 있습니다.

--ignore-module=<mod>

주어진 각 모듈 이름과 (패키지라면) 그 서브 모듈을 무시합니다. 인자는 쉼표로 구분된 이름 목록일 수 있습니다.

--ignore-dir=<dir>

명명된 디렉터리와 하위 디렉터리의 모든 모듈과 패키지를 무시합니다. 인자는 *os.pathsep*으로 구분된 디렉터리 목록일 수 있습니다.

27.7.2 프로그래밍 인터페이스

class trace.**Trace**(*count=1, trace=1, countfuncs=0, countcallers=0, ignoremods=(), ignoredirs=(), infile=None, outfile=None, timing=False*)

단일 문장이나 표현식의 실행을 추적할 객체를 만듭니다. 모든 매개 변수는 선택 사항입니다. *count*는 줄 번호의 카운팅을 활성화합니다. *trace*는 줄 실행 추적을 활성화합니다. *countfuncs*는 실행 중에 호출된 함수들의 리스팅을 활성화합니다. *countcallers*는 호출 관계 추적을 활성화합니다. *ignoremods*는 무시할 모듈이나 패키지의 리스트입니다. *ignoredirs*는 들어있는 모듈이나 패키지를 무시해야 하는 디렉터리의 리스트입니다. *infile*은 저장된 카운트 정보를 읽을 파일 이름입니다. *outfile*는 갱신된 카운트 정보를 쓰는 파일의 이름입니다. *timing*는 추적이 시작될 때에 상대적인 타임스탬프 표시를 활성화합니다.

run(*cmd*)

명령을 실행하고 현재 추적 매개 변수를 사용하여 실행에서 통계를 수집합니다. *cmd*는 *exec()*로 전달하는데 적합한 문자열이나 코드 객체여야 합니다.

runctx(*cmd, globals=None, locals=None*)

정의된 전역과 지역 환경에서, 명령을 실행하고 현재 추적 매개 변수를 사용하여 실행에서 통계를 수집합니다. 정의되지 않으면, *globals* 와 *locals*의 기본값은 빈 딕셔너리입니다.

runfunc(*func, /, *args, **kwds*)

현재 추적 매개 변수를 갖는 *Trace* 객체의 제어하에 주어진 인자로 *func*를 호출합니다.

results()

주어진 *Trace* 인스턴스에 대한 모든 이전 `run`, `runctx` 및 `runfunc` 호출의 누적 결과를 포함하는 *CoverageResults* 객체를 반환합니다. 누적된 추적 결과를 재설정하지 않습니다.

class trace.**CoverageResults**

*Trace.results()*에 의해 만들어진 커버리지 결과를 위한 컨테이너. 사용자가 직접 만들어서는 안 됩니다.

update(*other*)

다른 *CoverageResults* 객체의 데이터를 병합합니다.

write_results(*show_missing=True, summary=False, coverdir=None, *, ignore_missing_files=False*)

커버리지 결과를 기록합니다. 실행 카운트가 없는 줄을 표시하려면 *show_missing*을 설정하십시오. 모듈당 커버리지 요약을 출력에 포함 시키려면 *summary*를 설정하십시오. *coverdir*은 커버리지 결과 파일이 출력될 디렉터리를 지정합니다. None이면, 각 소스 파일의 결과가 해당 디렉터리에 위치합니다.

If *ignore_missing_files* is True, coverage counts for files that no longer exist are silently ignored. Otherwise, a missing file will raise a *FileNotFoundError*.

버전 3.13에서 변경: Added *ignore_missing_files* parameter.

프로그래밍 인터페이스의 사용을 보여주는 간단한 예제:

```
import sys
import trace

# Trace 객체를 만들고, 무시할 대상과, 추적할지, 줄을 셀지 또는 둘 다 수행할지를 지정합니
다.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# 주어진 tracer를 사용해서 새 명령을 실행합니다
tracer.run('main()')

# 보고서를 만들어, 현재 디렉터리에 출력을 넣습니다
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

27.8 tracemalloc — 메모리 할당 추적

Added in version 3.4.

소스 코드: Lib/tracemalloc.py

---

tracemalloc 모듈은 파이썬이 할당한 메모리 블록을 추적하는 디버그 도구입니다. 다음 정보를 제공합니다:

- 객체가 할당된 곳의 트레이스백
- 파일명과 줄 번호별로 할당된 메모리 블록에 대한 통계: 할당된 메모리 블록의 총 크기, 수 및 평균 크기
- 메모리 누수를 탐지하기 위해 두 스냅샷의 차이점 계산

파이썬이 할당한 대부분의 메모리 블록을 추적하려면, PYTHONTRACEMALLOC 환경 변수를 1로 설정하거나, -X tracemalloc 명령 줄 옵션을 사용하여 모듈을 가능한 한 빨리 시작해야 합니다. *tracemalloc.start()* 함수는 실행 시간에 호출되어 파이썬 메모리 할당 추적을 시작할 수 있습니다.

기본적으로, 할당된 메모리 블록의 트레이스는 가장 최근의 프레임 만 저장합니다 (1프레임). 시작 시 25 프레임을 저장하려면: PYTHONTRACEMALLOC 환경 변수를 25로 설정하거나, -X tracemalloc=25 명령 줄 옵션을 사용하십시오.

27.8.1 예

상위 10개 표시

가장 많은 메모리를 할당하는 10개의 파일을 표시합니다:

```
import tracemalloc

tracemalloc.start()

# ... 여러분의 응용 프로그램을 실행합니다 ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

파이썬 테스트 스위트의 출력 예:

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315,␣
↪average=108 B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

파이썬이 모듈에서 4855 KiB 데이터 (바이트 코드와 상수)를 로드했으며 *collections* 모듈이 *namedtuple* 형을 빌드하기 위해 244 KiB를 할당했음을 알 수 있습니다.

추가 옵션은 *Snapshot.statistics()*를 참조하십시오.

차이 계산

두 개의 스냅샷을 취하고 차이점을 표시합니다:

```
import tracemalloc
tracemalloc.start()
# ... 여러분의 응용 프로그램을 시작합니다 ...

snapshot1 = tracemalloc.take_snapshot()
# ... 메모리 누수를 일으키는 함수를 호출합니다 ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

파이썬 테스트 스위트의 일부 테스트를 실행하기 전/후의 출력 예:

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369),
↪ average=117 B
/usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB), count=8106 (+8106),␣
↪average=119 B
/usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298 KiB), count=589 (+589),
↪ average=519 B
<frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB), count=7423 (+1526),␣
↪average=139 B
/usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB), count=1334 (+1334),␣
↪average=86 B
/usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0 KiB), count=1 (+1),␣
↪average=96.0 KiB
/usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB), count=109 (+109),␣
↪average=784 B
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143␣
↪(+143), average=557 B
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969␣
↪(+969), average=76 B
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126),␣
↪average=546 B
```

우리는 파이썬이 `8173 KiB`의 모듈 데이터(바이트 코드와 상수)를 로드했으며, 이전 스냅샷을 취할 때인 테스트 전에 로드된 것보다 `4428 KiB` 더 많은 것을 볼 수 있습니다. 마찬가지로, `linecache` 모듈은 트레이스백을 포맷하기 위해 `940 KiB`의 파이썬 소스 코드를 캐시 했는데, 이전 스냅샷 이후의 모든 것입니다.

시스템에 사용 가능한 메모리가 거의 없으면, 스냅샷을 오프라인으로 분석하기 위해 `Snapshot.dump()` 메서드로 디스크에 스냅샷을 기록할 수 있습니다. 그런 다음 `Snapshot.load()` 메서드를 사용하여 스냅샷을 다시 로드하십시오.

메모리 블록의 트레이스백 얻기

가장 큰 메모리 블록의 트레이스백을 표시하는 코드:

```
import tracemalloc

# 25프레임을 저장합니다
tracemalloc.start(25)

# ... 여러분의 응용 프로그램을 실행합니다 ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# 가장 큰 메모리 블록을 꺼냅니다
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)
```

파이썬 테스트 스위트의 출력 예 (트레이스백은 25프레임으로 제한되었습니다):

```
903 memory blocks: 870.1 KiB
  File "<frozen importlib._bootstrap>", line 716
  File "<frozen importlib._bootstrap>", line 1036
  File "<frozen importlib._bootstrap>", line 934
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/doctest.py", line 101
    import pdb
  File "<frozen importlib._bootstrap>", line 284
  File "<frozen importlib._bootstrap>", line 938
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/test/support/__init__.py", line 1728
    import doctest
  File "/usr/lib/python3.4/test/test_pickletools.py", line 21
    support.run_doctest(pickletools)
  File "/usr/lib/python3.4/test/regrtest.py", line 1276
    test_runner()
  File "/usr/lib/python3.4/test/regrtest.py", line 976
    display_failure=not verbose)
  File "/usr/lib/python3.4/test/regrtest.py", line 761
    match_tests=ns.match_tests)
  File "/usr/lib/python3.4/test/regrtest.py", line 1563
    main()
  File "/usr/lib/python3.4/test/__main__.py", line 3
    regrtest.main_in_temp_cwd()
  File "/usr/lib/python3.4/runpy.py", line 73
    exec(code, run_globals)
  File "/usr/lib/python3.4/runpy.py", line 160
    "__main__", fname, loader, pkg_name)
```

대부분의 메모리가 모듈에서 데이터(바이트 코드와 상수)를 로드하기 위해 *importlib* 모듈에 할당되었음을 알 수 있습니다: `870.1 KiB`. 트레이스백은 *importlib*가 가장 최근에 데이터를 로드한 위치입니다: *doctest* 모듈의 `import pdb` 줄. 새 모듈이 로드되면 트레이스백이 변경될 수 있습니다.

예쁜 탑(top)

`<frozen importlib._bootstrap>`과 `<unknown>` 파일을 무시하고, 예쁜 출력으로 가장 많은 메모리를 할당하는 10개의 줄을 표시하는 코드:

```
import linecache
import os
import tracemalloc

def display_top(snapshot, key_type='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(key_type)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        print("#%s: %s:%s: %.1f KiB"
              % (index, frame.filename, frame.lineno, stat.size / 1024))
        line = linecache.getline(frame.filename, frame.lineno).strip()
        if line:
            print('    %s' % line)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()

# ... 여러분의 응용 프로그램을 실행합니다 ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

파이썬 테스트 스위트의 출력 예:

```
Top 10 lines
#1: Lib/base64.py:414: 419.8 KiB
    _b85chars2 = [(a + b) for a in _b85chars for b in _b85chars]
#2: Lib/base64.py:306: 419.8 KiB
    _a85chars2 = [(a + b) for a in _a85chars for b in _a85chars]
#3: collections/__init__.py:368: 293.6 KiB
    exec(class_definition, namespace)
#4: Lib/abc.py:133: 115.2 KiB
    cls = super().__new__(mcls, name, bases, namespace)
#5: unittest/case.py:574: 103.1 KiB
    testMethod()
#6: Lib/linecache.py:127: 95.4 KiB
    lines = fp.readlines()
#7: urllib/parse.py:476: 71.8 KiB
    for a in _hexdig for b in _hexdig}
#8: <string>:5: 62.0 KiB
#9: Lib/_weakrefset.py:37: 60.0 KiB
    self.data = set()
#10: Lib/base64.py:142: 59.8 KiB
    _b32tab2 = [a + b for a in _b32tab for b in _b32tab]
6220 other: 3602.8 KiB
Total allocated size: 5303.1 KiB
```

추가 옵션은 *Snapshot.statistics()*를 참조하십시오.

모든 추적한 메모리 블록의 현재 및 최대 크기를 기록

다음 코드는 `0 + 1 + 2 + ...`와 같은 두 개의 합계를 비효율적으로 계산하는데, 이 숫자들의 리스트를 만듭니다. 이 리스트는 일시적으로 많은 메모리를 소비합니다. *get_traced_memory()*와 *reset_peak()*를 사용하여 계산 중 최대 메모리 사용량뿐만 아니라 합이 계산된 후의 작은 메모리 사용량도 관찰할 수 있습니다:

```
import tracemalloc

tracemalloc.start()

# 예제 코드: 커다란 임시 리스트로 합계를 계산합니다
large_sum = sum(list(range(100000)))

first_size, first_peak = tracemalloc.get_traced_memory()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
tracemalloc.reset_peak()

# 예제 코드: 작은 임시 리스트로 합계를 계산합니다
small_sum = sum(list(range(1000)))

second_size, second_peak = tracemalloc.get_traced_memory()

print(f"{first_size=}, {first_peak=}")
print(f"{second_size=}, {second_peak=}")
```

출력:

```
first_size=664, first_peak=3592984
second_size=804, second_peak=29704
```

*reset_peak()*를 사용하면 `small_sum`을 계산하는 동안의 최대 사용량을 *start()* 호출 이후 메모리 블록의 전체 최대 크기보다 훨씬 작더라도 정확하게 기록 할 수 있습니다. *reset_peak()*를 호출하지 않으면, `second_peak`는 여전히 `large_sum` 계산의 최대가 됩니다 (즉, `first_peak`와 같습니다). 이 경우, 두 최댓값은 모두 최종 메모리 사용량보다 훨씬 높아서, 최적화할 수 있음을 제안합니다 (*list*에 대한 불필요한 호출을 제거하고, `sum(range(...))` 이라고 작성하여).

27.8.2 API

함수

`tracemalloc.`**`clear_traces`**`()`

파이썬이 할당한 메모리 블록의 트레이스를 지웁니다.

*stop()*도 참조하십시오.

`tracemalloc.`**`get_object_traceback`**`(`*obj*`)`

파이썬 객체 *obj*가 할당된 위치의 트레이스백을 가져옵니다. *Traceback* 인스턴스를 반환하거나, *tracemalloc* 모듈이 메모리 할당을 추적하지 않고 있거나 객체 할당을 추적하지 않았으면 `None`을 반환합니다.

*gc.get_referrers()*와 *sys.getsizeof()* 함수도 참조하십시오.

`tracemalloc.`**`get_traceback_limit`**`()`

트레이스의 트레이스백에 저장된 최대 프레임 수를 가져옵니다.

한계를 얻으려면 *tracemalloc* 모듈이 메모리 할당을 추적하고 있어야 합니다, 그렇지 않으면 예외가 발생합니다.

한계는 *start()* 함수에 의해 설정됩니다.

`tracemalloc.`**`get_traced_memory`**`()`

tracemalloc 모듈이 추적하는 메모리 블록의 현재 크기와 최대 크기를 튜플로 가져옵니다: `(current: int, peak: int)`.

`tracemalloc.`**`reset_peak`**`()`

tracemalloc 모듈이 추적하는 메모리 블록의 최대 크기를 현재 크기로 설정합니다.

tracemalloc 모듈이 파이썬 메모리 할당을 추적하고 있지 않으면 아무것도 하지 않습니다.

이 함수는 기록된 최대 크기만 수정하며, *clear_traces()*와 달리 어떤 추적도 수정하거나 지우지 않습니다. *reset_peak()*를 호출하기 전에 *take_snapshot()*로 찍은 스냅숏은 호출 후 찍은 스냅숏과 의미 있게 비교할 수 있습니다.

*get_traced_memory()*도 참조하십시오.

Added in version 3.9.

tracemalloc.**get_tracemalloc_memory**()

메모리 블록의 트레이스를 저장하는 데 사용된 *tracemalloc* 모듈의 메모리 사용량을 바이트 단위로 가져옵니다. *int*를 반환합니다.

tracemalloc.**is_tracing**()

tracemalloc 모듈이 파이썬 메모리 할당을 추적하고 있으면 True, 그렇지 않으면 False.

*start()*와 *stop()* 함수도 참조하십시오.

tracemalloc.**start**(*nframe:* int = *1*)

파이썬 메모리 할당 추적을 시작합니다: 파이썬 메모리 할당자(allocator)에 훅을 설치합니다. 수집된 트레이스의 트레이스백은 *nframe* 개의 프레임으로 제한됩니다. 기본적으로, 메모리 블록의 트레이스는 가장 최근의 프레임 만 저장합니다: 제한은 1 입니다. *nframe*은 1 보다 크거나 같아야 합니다.

Traceback.total_nframe 어트리뷰트를 보면 트레이스백을 구성한 원래의 총 프레임 수를 계속 읽을 수 있습니다.

1 개보다 더 많은 프레임을 저장하는 것은 'traceback'으로 그룹화된 통계를 계산하거나 누적 통계를 계산할 때만 유용합니다: *Snapshot.compare_to()*와 *Snapshot.statistics()* 메서드를 참조하십시오.

더 많은 프레임을 저장하면 *tracemalloc* 모듈의 메모리와 CPU 오버헤드가 증가합니다. *get_tracemalloc_memory()* 함수를 사용하여 *tracemalloc* 모듈이 사용하는 메모리양을 측정하십시오.

PYTHONTRACEMALLOC 환경 변수(PYTHONTRACEMALLOC=NFRAME)와 -X tracemalloc=NFRAME 명령 줄 옵션을 사용하여 시작 시 추적을 시작할 수 있습니다.

stop(), *is_tracing()* 및 *get_traceback_limit()* 함수도 참조하십시오.

tracemalloc.**stop**()

파이썬 메모리 할당 추적을 중지합니다: 파이썬 메모리 할당자에서 훅을 제거합니다. 또한 파이썬이 할당한 메모리 블록의 이전에 수집된 모든 트레이스를 지웁니다.

트레이스를 지우기 전에 *take_snapshot()* 함수를 호출하여 트레이스의 스냅샷을 취하십시오.

start(), *is_tracing()* 및 *clear_traces()* 함수도 참조하십시오.

tracemalloc.**take_snapshot**()

파이썬이 할당한 메모리 블록의 트레이스의 스냅샷을 취합니다. 새로운 *Snapshot* 인스턴스를 반환합니다.

tracemalloc 모듈이 메모리 할당 추적을 시작하기 전에 할당된 메모리 블록은 스냅샷에 포함되지 않습니다.

트레이스의 트레이스백은 *get_traceback_limit()* 개의 프레임으로 제한됩니다. 더 많은 프레임을 저장하려면 *start()* 함수의 *nframe* 매개 변수를 사용하십시오.

스냅샷을 취하기 위해서는 *tracemalloc* 모듈이 메모리 할당을 추적하고 있어야 합니다, *start()* 함수를 참조하십시오.

get_object_traceback() 함수도 참조하십시오.

DomainFilter

class tracemalloc.**DomainFilter**(*inclusive:* bool, *domain:* int)

주소 공간(도메인)별로 메모리 블록의 트레이스를 필터링합니다.

Added in version 3.6.

inclusive

*inclusive*가 True 이면 (포함), 주소 공간 *domain*에 할당된 메모리 블록을 일치시킵니다.

*inclusive*가 False 이면 (제외), 주소 공간 *domain*에 할당되지 않은 메모리 블록을 일치시킵니다.

domain

메모리 블록의 주소 공간 (int). 읽기 전용 프로퍼티.

Filter

class tracemalloc.**Filter**(*inclusive:* bool, *filename_pattern:* str, *lineno:* int = *None*, *all_frames:* bool = *False*, *domain:* int = *None*)

메모리 블록의 트레이스를 필터링합니다.

*filename_pattern*의 문법은 `fnmatch.fnmatch()` 함수를 참조하십시오. `'.pyc'` 파일 확장자가 `'.py'`로 대체됩니다.

예:

- `Filter(True, subprocess.__file__)`은 `subprocess` 모듈의 트레이스만 포함합니다
- `Filter(False, tracemalloc.__file__)`은 `tracemalloc` 모듈의 트레이스를 제외합니다
- `Filter(False, "<unknown>")`은 빈 트레이스백을 제외합니다

버전 3.5에서 변경: `'.pyo'` 파일 확장자는 더는 `'.py'`로 대체되지 않습니다.

버전 3.6에서 변경: `domain` 어트리뷰트를 추가했습니다.

domain

메모리 블록의 주소 공간 (`int`나 `None`).

tracemalloc은 도메인 `0`을 사용하여 파이썬의 메모리 할당을 추적합니다. C 확장은 다른 도메인을 사용하여 다른 리소스를 추적 할 수 있습니다.

inclusive

*inclusive*가 `True`(포함)이면, 줄 번호 `lineno`에서 이름이 `filename_pattern`과 일치하는 파일에서 할당된 메모리 블록만 일치시킵니다.

*inclusive*가 `False`(제외)이면, 줄 번호 `lineno`에서 이름이 `filename_pattern`과 일치하는 파일에 할당된 메모리 블록을 무시합니다.

lineno

필터의 줄 번호 (`int`). *lineno*가 `None`이면, 필터는 모든 줄 번호와 일치합니다.

filename_pattern

필터의 파일명 패턴 (`str`). 읽기 전용 프로퍼티.

all_frames

*all_frames*가 `True`이면, 트레이스백의 모든 프레임이 검사됩니다. *all_frames*가 `False`이면, 가장 최근 프레임 만 검사됩니다.

트레이스백 한계가 `1`이면 이 어트리뷰트가 적용되지 않습니다. `get_traceback_limit()` 함수와 `Snapshot.traceback_limit` 어트리뷰트를 참조하십시오.

Frame

class tracemalloc.**Frame**

트레이스백의 프레임.

`Traceback` 클래스는 `Frame` 인스턴스의 시퀀스입니다.

filename

파일명 (`str`).

lineno

줄 번호 (`int`).

Snapshot

class tracemalloc.**Snapshot**

파이썬이 할당한 메모리 블록의 트레이스의 스냅샷.

`take_snapshot()` 함수는 스냅샷 인스턴스를 만듭니다.

compare_to(*old_snapshot: Snapshot, key_type: str, cumulative: bool = False*)

이전 스냅샷과의 차이점을 계산합니다. *key_type* 별로 그룹화된 `StatisticDiff` 인스턴스의 정렬된 리스트로 통계를 가져옵니다.

*key_type*과 *cumulative* 매개 변수에 대해서는 `Snapshot.statistics()` 메서드를 참조하십시오.

결과는 다음 값에 따라 내림차순으로 정렬됩니다: `StatisticDiff.size_diff`의 절댓값, `StatisticDiff.size`, `StatisticDiff.count_diff`의 절댓값, `Statistic.count` 그런 다음 `StatisticDiff.traceback`.

dump(*filename*)

스냅샷을 파일에 씁니다.

스냅샷을 다시 로드하려면 `load()`를 사용하십시오.

filter_traces(*filters*)

필터링 된 `traces` 시퀀스로 새 `Snapshot` 인스턴스를 만듭니다. *filters*는 `DomainFilter`와 `Filter` 인스턴스의 리스트입니다. *filters*가 빈 리스트면, traces의 사본으로 새 `Snapshot` 인스턴스를 반환합니다.

모든 포함 필터가 한 번에 적용되며, 아무런 포함 필터도 일치하지 않으면 트레이스는 무시됩니다. 하나 이상의 제외 필터가 일치하면 트레이스는 무시됩니다.

버전 3.6에서 변경: `DomainFilter` 인스턴스도 이제 *filters*에서 허용됩니다.

classmethod load(*filename*)

파일에서 스냅샷을 로드합니다.

`dump()`도 참조하십시오.

statistics(*key_type: str, cumulative: bool = False*)

key_type 별로 그룹화된 `Statistic` 인스턴스의 정렬된 리스트로 통계를 가져옵니다:

key_type	설명
`'filename'`	파일명
`'lineno'`	파일명과 줄 번호
`'traceback'`	트레이스백

*cumulative*가 `True`이면, 가장 최근의 프레임뿐만 아니라, 트레이스의 트레이스백의 모든 프레임에 대한 메모리 블록의 크기와 개수를 누적합니다. 누적 모드는 *key_type*이 `'filename'`과 `'lineno'`와 같을 때만 사용할 수 있습니다.

결과는 다음 값에 따라 내림차순으로 정렬됩니다: `Statistic.size`, `Statistic.count` 그런 다음 `Statistic.traceback`.

traceback_limit

`traces`의 트레이스백에 저장된 최대 프레임 수: 스냅샷을 취할 때 `get_traceback_limit()`의 결과.

traces

파이썬이 할당한 모든 메모리 블록의 트레이스: `Trace` 인스턴스의 시퀀스.

시퀀스의 순서는 정의되지 않았습니다. 정렬된 통계 리스트를 얻으려면 `Snapshot.statistics()` 메서드를 사용하십시오.

Statistic

class tracemalloc.**Statistic**

메모리 할당 통계.

`Snapshot.statistics()`는 `Statistic` 인스턴스의 리스트를 반환합니다.

`StatisticDiff` 클래스도 참조하십시오.

count

메모리 블록 수 (int).

size

총 메모리 블록의 바이트 단위 크기 (int).

traceback

메모리 블록이 할당된 곳의 트레이스백, *Traceback* 인스턴스.

StatisticDiff

class tracemalloc.**StatisticDiff**

기존 *Snapshot* 인스턴스와 새 인스턴스 간의 메모리 할당에 대한 통계적 차이.

*Snapshot.compare_to()*는 *StatisticDiff* 인스턴스의 리스트를 반환합니다. *Statistic* 클래스도 참조하십시오.

count

새 스냅샷의 메모리 블록 수 (int): 새 스냅샷에서 메모리 블록이 해제되었으면 0.

count_diff

이전 스냅샷과 새 스냅샷 간의 메모리 블록 수의 차이 (int): 메모리 블록이 새 스냅샷에 할당되었으면 0.

size

새 스냅샷에서 총 메모리 블록의 바이트 단위 크기 (int): 새 스냅샷에서 메모리 블록이 해제되었으면 0.

size_diff

이전 스냅샷과 새 스냅샷 사이의 총 메모리 블록 크기의 바이트 단위 차이 (int): 메모리 블록이 새 스냅샷에 할당되었으면 0.

traceback

메모리 블록이 할당된 곳의 트레이스백, *Traceback* 인스턴스.

Trace

class tracemalloc.**Trace**

메모리 블록의 트레이스.

Snapshot.traces 어트리뷰트는 *Trace* 인스턴스의 시퀀스입니다.

버전 3.6에서 변경: *domain* 어트리뷰트를 추가했습니다.

domain

메모리 블록의 주소 공간 (int). 읽기 전용 프로퍼티.

tracemalloc은 도메인 0을 사용하여 파이썬의 메모리 할당을 추적합니다. C 확장은 다른 도메인을 사용하여 다른 리소스를 추적 할 수 있습니다.

size

메모리 블록의 바이트 단위 크기 (int).

traceback

메모리 블록이 할당된 곳의 트레이스백, *Traceback* 인스턴스.

Traceback

class tracemalloc.**Traceback**

가장 오래된 프레임에서 가장 최근 프레임 순으로 정렬된 *Frame* 인스턴스의 시퀀스.

트레이스백은 적어도 1프레임을 포함합니다. tracemalloc 모듈이 프레임을 가져오지 못하면, 줄 번호 0의 파일명 "<unknown>"이 사용됩니다.

스냅샷을 취할 때, 트레이스의 트레이스백은 *get_traceback_limit()* 프레임으로 제한됩니다. *take_snapshot()* 함수를 참조하십시오. 트레이스백의 원래 프레임 수는 *Traceback.total_nframe* 어트리뷰트에 저장됩니다. 이를 통해 트레이스백 제한으로 인해 트레이스백이 잘렸는지 알 수 있습니다.

Trace.traceback 어트리뷰트는 *Traceback* 인스턴스의 인스턴스입니다.

버전 3.7에서 변경: 프레임은 이제 가장 최근에서 가장 오래된 것 대신, 가장 오래된 것에서 가장 최근으로 정렬됩니다.

total_nframe

자르기 전에 트레이스백을 구성한 총 프레임 수. 정보가 없으면 이 어트리뷰트를 None으로 설정할 수 있습니다.

버전 3.9에서 변경: *Traceback.total_nframe* 어트리뷰트가 추가되었습니다.

format(*limit=None*, *most_recent_first=False*)

줄의 리스트로 트레이스백을 포맷합니다. 소스 코드에서 줄을 꺼내는데 *linecache* 모듈을 사용합니다. *limit*가 설정되면, *limit*가 양수일 때 가장 최신 *limit* 개의 프레임을 포맷합니다. 그렇지 않으면, 가장 오래된 abs(limit) 개의 프레임을 포맷합니다. *most_recent_first*가 True이면, 포맷된 프레임의 순서가 반대로 되어, 가장 최근의 프레임을 마지막이 아니라 처음에 반환합니다.

*format()*에 줄 넘김 문자가 포함되지 않는다는 점을 제외하고, *traceback.format_tb()* 함수와 유사합니다.

예:

```
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

출력:

```
Traceback (most recent call first):
  File "test.py", line 9
    obj = Object()
  File "test.py", line 12
    tb = tracemalloc.get_object_traceback(f())
```

CHAPTER 28

소프트웨어 패키징 및 배포

이 라이브러리는 파이썬 소프트웨어를 게시하고 설치하는 데 도움을 줍니다. 이 모듈은 파이썬 패키지 색인과 함께 작동하도록 설계되었지만, 로컬 색인 서버와 함께 사용할 수도, 색인 서버 없이 사용할 수도 있습니다.

28.1 ensurepip — pip 설치 프로그램 부트스트랩

Added in version 3.4.

소스 코드: Lib/ensurepip

---

ensurepip 패키지는 pip 설치 프로그램을 기존의 파이썬 설치나 가상 환경으로 부트스트랩 하는데 필요한 지원을 제공합니다. 이 부트스트랩 접근 방식은 pip가 자체 배포 주기가 있는 독립적인 프로젝트이며, 최신 사용 가능한 안정 버전이 CPython 참조 인터프리터의 유지 보수와 기능 배포에 번들로 제공된다는 사실을 반영합니다.

대부분, 파이썬의 최종 사용자는 이 모듈을 직접 호출할 필요가 없습니다 (pip는 기본적으로 부트스트랩 되어있어야 하기 때문입니다). 하지만, 파이썬을 설치할 때 (또는 가상 환경을 만들 때) pip를 건너뛰었거나 그 후에 명시적으로 pip를 제거했다면 필요할 수 있습니다.

> **참고**
>
> 이 모듈은 인터넷에 접속하지 않습니다. pip를 부트스트랩 하는 데 필요한 모든 구성 요소는 패키지의 내부 부품으로 포함됩니다.

> **더 보기**
>
> **installing-index**
> 파이썬 패키지를 설치하기 위한 최종 사용자 지침서
>
> **PEP 453: 파이썬 설치에서 pip의 명시적 부트스트랩**
> 이 모듈의 원래 근거와 사양.

가용성: not Android, not iOS, not WASI.

이 모듈은 모바일 플랫폼이나 웹어셈블리 플랫폼에서 지원되지 않습니다.

28.1.1 명령 줄 인터페이스

명령 줄 인터페이스는 인터프리터의 `-m` 스위치를 사용하여 호출됩니다.

가장 간단한 호출은 이렇습니다:

```
python -m ensurepip
```

이 호출은 아직 설치되지 않았으면 `pip`를 설치하지만, 그렇지 않으면 아무것도 하지 않습니다. `pip`의 설치 버전이 적어도 `ensurepip`에서 제공하는 것 만큼 최신 버전이 되도록 하려면, `--upgrade` 옵션을 전달하십시오:

```
python -m ensurepip --upgrade
```

기본적으로, `pip`는 현재 가상 환경(활성화되었다면)이나 시스템 사이트 패키지(활성 가상 환경이 없으면)에 설치됩니다. 설치 위치는 두 개의 추가 명령 줄 옵션을 통해 제어할 수 있습니다:

--root <dir>

현재 활성화된 가상 환경의 루트(있다면)나 현재 파이썬 설치의 기본 루트 대신, 지정된 루트 디렉터리에 상대적으로 `pip`를 설치합니다.

--user

`pip`를 현재 파이썬 설치에 전역적으로 설치하지 않고 사용자 사이트 패키지 디렉터리에 설치합니다 (이 옵션은 활성 가상 환경에서는 허용되지 않습니다).

기본적으로, `pipX` 와 `pipX.Y` 스크립트가 설치됩니다 (여기서 X.Y는 `ensurepip`를 호출하는 데 사용된 파이썬 버전을 나타냅니다). 설치된 스크립트는 두 개의 추가 명령 줄 옵션을 통해 제어할 수 있습니다:

--altinstall

대안 설치가 요청되면, `pipX` 스크립트가 설치되지 않습니다.

--default-pip

"기본 pip" 설치가 요청되면, 두 개의 일반 스크립트에 더해 `pip` 스크립트가 설치됩니다.

두 스크립트 선택 옵션을 모두 제공하면 예외가 발생합니다.

28.1.2 모듈 API

*`ensurepip`*는 프로그래밍 방식으로 사용하기 위해 두 가지 함수를 제공합니다:

`ensurepip.`**`version`**`()`

환경을 부트스트랩 할 때 설치될 pip의 버전을 지정하는 문자열을 반환합니다.

`ensurepip.`**`bootstrap`**`(`*root=None, upgrade=False, user=False, altinstall=False, default_pip=False, verbosity=0*`)`

`pip`를 현재나 지정된 환경으로 부트스트랩 합니다.

*root*는 상대 경로로 설치할 대안 루트 디렉터리를 지정합니다. *root*가 `None`이면, 설치는 현재 환경의 기본 설치 위치를 사용합니다.

*upgrade*는 이미 설치된 이전 버전의 `pip`를 제공되는 버전으로 업그레이드할지를 나타냅니다.

*user*는 전역으로 설치하는 대신 사용자 구성을 사용할지를 나타냅니다.

기본적으로, `pipX` 및 `pipX.Y` 스크립트가 설치됩니다 (여기서 X.Y는 현재 버전의 파이썬을 나타냅니다).

*altinstall*가 설정되면, `pipX`가 설치되지 않습니다.

*default_pip*가 설정되면, 두 개의 일반 스크립트에 더해 `pip`가 설치됩니다.

altinstall 과 *default_pip*를 모두 설정하면 *`ValueError`*가 발생합니다.

*verbosity*는 부트스트랩 연산에서 *`sys.stdout`*로 출력하는 수준을 제어합니다.

인자 `root`로 감사 이벤트(*auditing event*) `ensurepip.bootstrap`을 발생시킵니다.

> **참고**
>
> 부트스트랩 프로세스에는 `sys.path` 와 `os.environ` 모두에 부작용이 있습니다. 대신 자식 프로세스에서 명령 줄 인터페이스를 호출하면 이러한 부작용을 피할 수 있습니다.

> **참고**
>
> 부트스트랩 프로세스는 `pip`에 필요한 추가 모듈을 설치할 수 있지만, 다른 소프트웨어는 이러한 종속성이 기본적으로 항상 존재한다고 가정해서는 안 됩니다 (`pip`의 차후 버전에서 제거될 수 있기 때문입니다).

28.2 venv — 가상 환경 생성

Added in version 3.3.

소스 코드: Lib/venv/

The `venv` module supports creating lightweight "virtual environments", each with their own independent set of Python packages installed in their `site` directories. A virtual environment is created on top of an existing Python installation, known as the virtual environment' s "base" Python, and by default is isolated from the packages in the base environment, so that only those explicitly installed in the virtual environment are available. See *Virtual Environments* and `site`' s *virtual environments documentation* for more information.

가상 환경 내에서 사용될 때, pip와 같은 일반적인 설치 도구는 명시적으로 지정하지 않아도 파이썬 패키지를 가상 환경에 설치합니다.

A virtual environment is (amongst other things):

- Used to contain a specific Python interpreter and software libraries and binaries which are needed to support a project (library or application). These are by default isolated from software in other virtual environments and Python interpreters and libraries installed in the operating system.
- Contained in a directory, conventionally named `.venv` or `venv` in the project directory, or under a container directory for lots of virtual environments, such as `~/.virtualenvs`.
- Not checked into source control systems such as Git.
- Considered as disposable – it should be simple to delete and recreate it from scratch. You don' t place any project code in the environment.
- Not considered as movable or copyable – you just recreate the same environment in the target location.

파이썬 가상 환경에 대한 자세한 배경 정보는 **PEP 405**를 참조하십시오.

> **더 보기**
>
> Python Packaging User Guide: Creating and using virtual environments

가용성: not Android, not iOS, not WASI.

이 모듈은 모바일 플랫폼이나 웹어셈블리 플랫폼에서 지원되지 않습니다.

28.2.1 가상 환경 만들기

가상 환경은 venv 모듈을 실행해서 만들어집니다:

```
python -m venv /path/to/new/virtual/environment
```

This creates the target directory (including parent directories as needed) and places a pyvenv.cfg file in it with a home key pointing to the Python installation from which the command was run. It also creates a bin (or Scripts on Windows) subdirectory containing a copy or symlink of the Python executable (as appropriate for the platform or arguments used at environment creation time). It also creates a lib/pythonX.Y/site-packages subdirectory (on Windows, this is Lib\site-packages). If an existing directory is specified, it will be re-used.

버전 3.5에서 변경: 이제 가상 환경을 만들 때 venv를 사용하는 것이 좋습니다.

Deprecated since version 3.6, removed in version 3.8: **pyvenv**는 파이썬 3.3 및 3.4 용 가상 환경을 만드는 데 권장되는 도구였으며, 파이썬 3.5 에서 venv를 직접 실행하는 방식으로 대체되었습니다.

윈도우에서는, 다음과 같이 venv 명령을 호출하십시오:

```
PS> python -m venv C:\path\to\new\virtual\environment
```

명령에 -h를 사용하면 사용 가능한 옵션이 표시됩니다:

```
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip] [--prompt PROMPT] [--upgrade-deps]
            [--without-scm-ignore-files]
            ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target directories.

Once an environment has been created, you may wish to activate it, e.g. by
sourcing an activate script in its bin directory.
```

ENV_DIR

A required argument specifying the directory to create the environment in.

--system-site-packages

Give the virtual environment access to the system site-packages directory.

--symlinks

Try to use symlinks rather than copies, when symlinks are not the default for the platform.

--copies

Try to use copies rather than symlinks, even when symlinks are the default for the platform.

--clear

Delete the contents of the environment directory if it already exists, before environment creation.

--upgrade

Upgrade the environment directory to use this version of Python, assuming Python has been upgraded in-place.

--without-pip

Skips installing or upgrading pip in the virtual environment (pip is bootstrapped by default).

--prompt <PROMPT>

Provides an alternative prompt prefix for this environment.

--upgrade-deps

Upgrade core dependencies (pip) to the latest version in PyPI.

--without-scm-ignore-files

Skips adding SCM ignore files to the environment directory (Git is supported by default).

버전 3.4에서 변경: 기본적으로 pip을 설치합니다. `--without-pip` 와 `--copies` 옵션을 추가했습니다.

버전 3.4에서 변경: 이전 버전에서는, `--clear` 나 `--upgrade` 옵션이 제공되지 않았을 때, 대상 디렉터리가 이미 존재하면 에러가 발생했습니다.

버전 3.9에서 변경: PyPI에 있는 최신으로 pip + setuptools를 업그레이드하는 `--upgrade-deps` 옵션을 추가했습니다.

버전 3.12에서 변경: `setuptools` is no longer a core venv dependency.

버전 3.13에서 변경: Added the `--without-scm-ignore-files` option.

버전 3.13에서 변경: `venv` now creates a `.gitignore` file for Git by default.

> **참고**
>
> 심볼릭 링크가 윈도우에서 지원되지만, 추천하지는 않습니다. 특히 파일 탐색기에서 `python.exe`를 더블 클릭하면 심볼릭 링크를 열심히 따라가고(resolve) 가상 환경은 무시됩니다.

> **참고**
>
> 마이크로소프트 윈도우에서는 사용자에 대한 실행 정책을 설정하여 `Activate.ps1` 스크립트를 활성화해야 할 수도 있습니다. 다음 PowerShell 명령을 실행하여 이를 수행할 수 있습니다:
>
> ```
> PS C:\> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
> ```
>
> 자세한 정보는 About Execution Policies를 참조하십시오.

만들어진 `pyvenv.cfg` 파일에는 `include-system-site-packages` 키도 포함되어 있으며, `venv`가 `--system-site-packages` 옵션으로 실행되면 `true`로 설정되고, 그렇지 않으면 `false`로 설정됩니다.

`--without-pip` 옵션을 주지 않는 한, 가상 환경으로 `pip`을 부트스트랩 하기 위해 *ensurepip*가 호출됩니다.

`venv`에 여러 경로를 지정할 수 있습니다. 이때 지정된 옵션에 따라 제공된 각 경로에서 같은 가상 환경이 만들어집니다.

28.2.2 How venvs work

When a Python interpreter is running from a virtual environment, *sys.prefix* and *sys.exec_prefix* point to the directories of the virtual environment, whereas *sys.base_prefix* and *sys.base_exec_prefix* point to those of the base Python used to create the environment. It is sufficient to check `sys.prefix != sys.base_prefix` to determine if the current interpreter is running from a virtual environment.

A virtual environment may be "activated" using a script in its binary directory (`bin` on POSIX; `Scripts` on Windows). This will prepend that directory to your `PATH`, so that running **python** will invoke the environment' s Python interpreter and you can run installed scripts without having to use their full path. The invocation of the activation script is platform-specific (*<venv>* must be replaced by the path to the directory containing the virtual environment):

플랫폼	셸	가상 환경을 활성화하는 명령
POSIX	bash/zsh	`$ source <venv>/bin/activate`
	fish	`$ source <venv>/bin/activate.fish`
	csh/tcsh	`$ source <venv>/bin/activate.csh`
	pwsh	`$ <venv>/bin/Activate.ps1`
윈도우	cmd.exe	`C:\> <venv>\Scripts\activate.bat`
	PowerShell	`PS C:\> <venv>\Scripts\Activate.ps1`

Added in version 3.4: **fish** 와 **csh** 활성화 스크립트.

Added in version 3.8: PowerShell Core 지원을 위해 POSIX에 설치된 PowerShell 활성화 스크립트.

가상 환경을 구체적으로 활성화할 필요는 없습니다, 파이썬을 호출할 때 그 환경의 파이썬 인터프리터를 가리키는 전체 경로를 지정하기만 하면 되기 때문입니다. 또한, 환경에 설치된 모든 스크립트는 활성화하지 않고도 실행 가능해야 합니다.

In order to achieve this, scripts installed into virtual environments have a "shebang" line which points to the environment' s Python interpreter, `#!/<path-to-venv>/bin/python`. This means that the script will run with that interpreter regardless of the value of `PATH`. On Windows, "shebang" line processing is supported if you have the launcher installed. Thus, double-clicking an installed script in a Windows Explorer window should run it with the correct interpreter without the environment needing to be activated or on the `PATH`.

가상 환경이 활성화되면, `VIRTUAL_ENV` 환경 변수가 환경의 경로로 설정됩니다. 가상 환경을 사용하기 위해 명시적으로 활성화할 필요가 없기 때문에, `VIRTUAL_ENV`로 가상 환경이 사용 중인지를 판단할 수 없습니다.

경고

Because scripts installed in environments should not expect the environment to be activated, their shebang lines contain the absolute paths to their environment' s interpreters. Because of this, environments are inherently non-portable, in the general case. You should always have a simple means of recreating an environment (for example, if you have a requirements file `requirements.txt`, you can invoke `pip install -r requirements.txt` using the environment' s `pip` to install all of the packages needed by the environment). If for any reason you need to move the environment to a new location, you should recreate it at the desired location and delete the one at the old location. If you move an environment because you moved a parent directory of it, you should recreate the environment in its new location. Otherwise, software installed into the environment may not work as expected.

셸에서 `deactivate`를 입력하여 가상 환경을 비활성화할 수 있습니다. 정확한 메커니즘은 플랫폼에 따라 다르고 내부 구현 상세입니다 (보통, 스크립트나 셸 함수가 사용됩니다).

28.2.3 API

위에서 설명한 고수준 메서드는 제삼자 가상 환경 작성자가 필요에 따라 환경을 사용자 정의할 수 있는 메커니즘을 제공하는 간단한 API를 사용합니다: `EnvBuilder` 클래스.

class venv.**EnvBuilder**(*system_site_packages=False, clear=False, symlinks=False, upgrade=False, with_pip=False, prompt=None, upgrade_deps=False, *, scm_ignore_files=frozenset()*)

`EnvBuilder` 클래스는 인스턴스를 만들 때 다음 키워드 인자를 받아들입니다:

- *system_site_packages* – 시스템 파이썬 site-packages가 환경에서 사용 가능해야 함을 나타내는 논릿값입니다 (기본값은 `False`).
- *clear* – 참이면, 환경을 만들기 전에, 대상 디렉터리에 존재하는 내용을 지우도록 하는 논릿값.
- *symlinks* – 파이썬 바이너리를 복사하는 대신 심볼릭 링크하려고 시도할지를 나타내는 논릿값입니다.
- *upgrade* – 참이면 실행 중인 파이썬으로 기존 환경을 업그레이드하도록 하는 논릿값 - 파이썬이 그 자리에서 업그레이드되었을 때 사용됩니다 (기본값은 `False`).
- *with_pip* – 참이면 가상 환경에 pip이 설치되도록 하는 논릿값입니다. `--default-pip` 옵션과 함께 `ensurepip`를 사용합니다.
- *prompt* – 가상 환경이 활성화된 후 사용할 문자열입니다 (기본값은 환경의 디렉터리 이름이 사용됨을 뜻하는 `None`입니다). 특수한 문자열 `"."`이 제공되면, 현재 디렉터리의 기본 이름 (basename)이 프롬프트로 사용됩니다.
- *upgrade_deps* – PyPI에서 기반 venv 모듈을 최신으로 갱신합니다
- *scm_ignore_files* – Create ignore files based for the specified source control managers (SCM) in the iterable. Support is defined by having a method named `create_{scm}_ignore_file`. The only value supported by default is `"git"` via `create_git_ignore_file()`.

버전 3.4에서 변경: `with_pip` 매개 변수 추가

버전 3.6에서 변경: `prompt` 매개 변수 추가

버전 3.9에서 변경: `upgrade_deps` 매개 변수 추가

버전 3.13에서 변경: `scm_ignore_files` 매개 변수 추가

EnvBuilder may be used as a base class.

create(*env_dir*)

가상 환경을 담을 대상 디렉터리(절대 또는 현재 디렉터리에 대한 상대)를 지정해서 가상 환경을 만듭니다. `create` 메서드는 지정된 디렉터리에 환경을 만들거나 적절한 예외를 발생시킵니다.

EnvBuilder 클래스의 `create` 메서드는 서브 클래스가 사용자 정의할 수 있는 훅을 보여줍니다:

```
def create(self, env_dir):
    """
    디렉터리에 가상화된 파이썬 환경을 만듭니다.
    env_dir은 안에 환경을 만들 대상 디렉터리입니다.
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
    self.setup_scripts(context)
    self.post_setup(context)
```

메서드 *ensure_directories()*, *create_configuration()*, *setup_python()*, *setup_scripts()* 및 *post_setup()* 각각을 재정의할 수 있습니다.

ensure_directories(*env_dir*)

환경 디렉터리와 이미 존재하지 않는 필요한 모든 서브 디렉터리를 만들고, 문맥 객체를 돌려줍니다. 문맥 객체는 다른 메서드에서 사용하기 위한 어트리뷰트(가령 경로)를 담고 있을 뿐입니다. *EnvBuilder*가 `clear=True` 인자로 만들어지면, 환경 디렉터리의 내용이 지워진 후 모든 필요한 서브 디렉터리가 재생성됩니다.

The returned context object is a *types.SimpleNamespace* with the following attributes:

- `env_dir` - The location of the virtual environment. Used for `__VENV_DIR__` in activation scripts (see *install_scripts()*).
- `env_name` - The name of the virtual environment. Used for `__VENV_NAME__` in activation scripts (see *install_scripts()*).
- `prompt` - The prompt to be used by the activation scripts. Used for `__VENV_PROMPT__` in activation scripts (see *install_scripts()*).
- `executable` - The underlying Python executable used by the virtual environment. This takes into account the case where a virtual environment is created from another virtual environment.
- `inc_path` - The include path for the virtual environment.
- `lib_path` - The purelib path for the virtual environment.
- `bin_path` - 가상 환경의 스크립트 경로.
- `bin_name` - The name of the script path relative to the virtual environment location. Used for `__VENV_BIN_NAME__` in activation scripts (see *install_scripts()*).
- `env_exe` - The name of the Python interpreter in the virtual environment. Used for `__VENV_PYTHON__` in activation scripts (see *install_scripts()*).
- `env_exec_cmd` - The name of the Python interpreter, taking into account filesystem redirections. This can be used to run Python in the virtual environment.

버전 3.11에서 변경: The *venv sysconfig installation scheme* is used to construct the paths of the created directories.

버전 3.12에서 변경: The attribute `lib_path` was added to the context, and the context object was documented.

create_configuration(*context*)

환경에 `pyvenv.cfg` 구성 파일을 만듭니다.

setup_python(*context*)

환경에 파이썬 실행 파일의 복사본이나 심볼릭 링크를 만듭니다. POSIX 시스템에서, 특정 실행 파일 `python3.x`가 사용되면, 해당 이름의 파일이 이미 존재하지 않는 한 `python` 과 `python3` 심볼릭 링크가 해당 실행 파일을 가리키도록 만들어집니다.

setup_scripts(*context*)

플랫폼에 적합한 활성화 스크립트를 가상 환경에 설치합니다.

upgrade_dependencies(*context*)

환경에서 핵심 venv 종속성 패키지 (현재 pip)를 업그레이드합니다. 이것은 환경에서 셸로 `pip` 실행 파일을 실행하여 수행됩니다.

Added in version 3.9.

버전 3.12에서 변경: setuptools is no longer a core venv dependency.

post_setup(*context*)

제삼자 구현에서 재정의하여 가상 환경에 패키지를 사전 설치하거나 다른 생성 후 단계를 수행할 수 있는 메서드입니다.

install_scripts(*context*, *path*)

이 메서드는 가상 환경에 사용자 정의 스크립트를 설치하는 데 도움을 주는데, 서브 클래스의 *setup_scripts()* 나 *post_setup()*에서 호출할 수 있습니다.

*path*는 `common`, `posix`, `nt` 서브 디렉터리를 포함해야 하는 디렉터리 경로입니다. 각 디렉터리에는 환경의 `bin` 디렉터리로 들어갈 스크립트가 들어 있습니다. `common`과 *os.name*에 해당하는 디렉터리의 내용은 자리 표시자의 일부 텍스트 치환 후 복사됩니다:

- `__VENV_DIR__`은 환경 디렉터리의 절대 경로로 치환됩니다.
- `__VENV_NAME__`은 환경 이름(환경 디렉터리의 최종 경로 세그먼트)으로 치환됩니다.
- `__VENV_PROMPT__`는 프롬프트(괄호로 묶인 환경 이름과 그 뒤의 스페이스)로 치환됩니다
- `__VENV_BIN_NAME__`은 bin 디렉터리의 이름(`bin` 이나 `Scripts`)으로 치환됩니다.
- `__VENV_PYTHON__`은 환경의 실행 파일의 절대 경로로 치환됩니다.

디렉터리는 존재하는 것이 허용됩니다 (기존 환경이 업그레이드될 때).

create_git_ignore_file(*context*)

Creates a `.gitignore` file within the virtual environment that causes the entire directory to be ignored by the Git source control manager.

Added in version 3.13.

버전 3.7.2에서 변경: 윈도우는 이제 실제 바이너리를 복사하는 대신 `python[w].exe`를 위한 리디렉터 스크립트를 사용합니다. 3.7.2 에서만, 소스 트리의 빌드에서 실행하지 않는 한 *setup_python()* 아무 작업도 수행하지 않습니다.

버전 3.7.3에서 변경: 윈도우는 *setup_scripts()* 대신 *setup_python()*의 일부로 리디렉터 스크립트를 복사합니다. 이것은 3.7.2는 해당하지 않습니다. 심볼릭 링크를 사용하면, 원래 실행 파일이 링크됩니다.

모듈 수준의 편리 함수도 있습니다:

venv.**create**(*env_dir*, *system_site_packages=False*, *clear=False*, *symlinks=False*, *with_pip=False*, *prompt=None*, *upgrade_deps=False*, **, scm_ignore_files=frozenset()*)

주어진 키워드 인자로 `EnvBuilder`를 만들고, *env_dir* 인자로 `create()` 메서드를 호출합니다.

Added in version 3.3.

버전 3.4에서 변경: *with_pip* 매개 변수 추가

버전 3.6에서 변경: *prompt* 매개 변수 추가

버전 3.9에서 변경: *upgrade_deps* 매개 변수 추가

버전 3.13에서 변경: *scm_ignore_files* 매개 변수 추가

28.2.4 EnvBuilder 확장 예제

다음 스크립트는 생성된 가상 환경에 setuptools 와 pip을 설치하는 서브 클래스를 구현하여 `EnvBuilder`를 확장하는 방법을 보여줍니다:

```
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv

class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    이 빌더는 setuptools 와 pip을 설치하여 만들어진 가상 환경에 다른 패키지를 pip이나
    easy_install 할 수 있도록 합니다.

    :param nodist: 참이면, 만들어진 가상 환경에 setuptools와 pip을
                   설치하지 않습니다.
    :param nopip: 참이면, 만들어진 가상 환경에 pip을 설치하지
                  않습니다.
    :param progress: setuptools 나 pip이 설치되면, progress 콜러블을 전달하여 설치 진행
                     상황을 감시할 수 있습니다. 지정하면, 두 개의 인자로 호출됩니다: 진
행률을
                     나타내는 문자열과 문자열의 출처를 나타내는 context. context 인자는
                     세 가지 값 중 하나를 가질 수 있습니다: virtualize() 자체에서
                     호출되었음을 나타내는 'main'과, 앱 설치에 사용된 자식 프로세스의 출
력
                     스트림에서 줄을 읽어서 얻었음을 나타내는 'stdout'과 'stderr'.

                     콜러블을 지정하지 않으면, sys.stderr에 기본 진행 정보가 출력됩니다.
    """

    def __init__(self, *args, **kwargs):
        self.nodist = kwargs.pop('nodist', False)
        self.nopip = kwargs.pop('nopip', False)
        self.progress = kwargs.pop('progress', None)
        self.verbose = kwargs.pop('verbose', False)
        super().__init__(*args, **kwargs)

    def post_setup(self, context):
        """
        만들어지는 가상 환경에 미리 설치될 필요가 있는 것들을 설정합니다.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        :param context: 처리 중인 가상 환경 생성 요청에 대한 정보.
        """
        os.environ['VIRTUAL_ENV'] = context.env_dir
        if not self.nodist:
            self.install_setuptools(context)
        # setuptools 없이 pip을 설치할 수 없습니다
        if not self.nopip and not self.nodist:
            self.install_pip(context)

    def reader(self, stream, context):
        """
        자식 프로세스의 출력 스트림에서 줄을 읽고 (지정되었으면) progress 콜러블로 전달하
거나 진행
        정보를 sys.stderr에 씁니다.
        """
        progress = self.progress
        while True:
            s = stream.readline()
            if not s:
                break
            if progress is not None:
                progress(s, context)
            else:
                if not self.verbose:
                    sys.stderr.write('.')
                else:
                    sys.stderr.write(s.decode('utf-8'))
                sys.stderr.flush()
        stream.close()

    def install_script(self, context, name, url):
        _, _, path, _, _, _ = urlparse(url)
        fn = os.path.split(path)[-1]
        binpath = context.bin_path
        distpath = os.path.join(binpath, fn)
        # 가상 환경의 바이너리 폴더에 스크립트를 내려받습니다
        urlretrieve(url, distpath)
        progress = self.progress
        if self.verbose:
            term = '\n'
        else:
            term = ''
        if progress is not None:
            progress('Installing %s ...%s' % (name, term), 'main')
        else:
            sys.stderr.write('Installing %s ...%s' % (name, term))
            sys.stderr.flush()
        # 가상 환경에 설치합니다
        args = [context.env_exe, fn]
        p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
        t1 = Thread(target=self.reader, args=(p.stdout, 'stdout'))
        t1.start()
        t2 = Thread(target=self.reader, args=(p.stderr, 'stderr'))
        t2.start()
        p.wait()
        t1.join()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        t2.join()
        if progress is not None:
            progress('done.', 'main')
        else:
            sys.stderr.write('done.\n')
        # 정리 - 더는 필요 없습니다
        os.unlink(distpath)

    def install_setuptools(self, context):
        """
        가상 환경에 setuptools를 설치합니다.

        :param context: 처리 중인 가상 환경 생성 요청에 대한 정보.
        """
        url = "https://bootstrap.pypa.io/ez_setup.py"
        self.install_script(context, 'setuptools', url)
        # clear up the setuptools archive which gets downloaded
        pred = lambda o: o.startswith('setuptools-') and o.endswith('.tar.gz')
        files = filter(pred, os.listdir(context.bin_path))
        for f in files:
            f = os.path.join(context.bin_path, f)
            os.unlink(f)

    def install_pip(self, context):
        """
        Install pip in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = 'https://bootstrap.pypa.io/get-pip.py'
        self.install_script(context, 'pip', url)


def main(args=None):
    import argparse

    parser = argparse.ArgumentParser(prog=__name__,
                                     description='Creates virtual Python '
                                                 'environments in one or '
                                                 'more target '
                                                 'directories.')
    parser.add_argument('dirs', metavar='ENV_DIR', nargs='+',
                        help='A directory in which to create the '
                             'virtual environment.')
    parser.add_argument('--no-setuptools', default=False,
                        action='store_true', dest='nodist',
                        help="Don't install setuptools or pip in the "
                             "virtual environment.")
    parser.add_argument('--no-pip', default=False,
                        action='store_true', dest='nopip',
                        help="Don't install pip in the virtual "
                             "environment.")
    parser.add_argument('--system-site-packages', default=False,
                        action='store_true', dest='system_site',
                        help='Give the virtual environment access to the '
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                              'system site-packages dir.')
    if os.name == 'nt':
        use_symlinks = False
    else:
        use_symlinks = True
    parser.add_argument('--symlinks', default=use_symlinks,
                        action='store_true', dest='symlinks',
                        help='Try to use symlinks rather than copies, '
                             'when symlinks are not the default for '
                             'the platform.')
    parser.add_argument('--clear', default=False, action='store_true',
                        dest='clear', help='Delete the contents of the '
                                           'virtual environment '
                                           'directory if it already '
                                           'exists, before virtual '
                                           'environment creation.')
    parser.add_argument('--upgrade', default=False, action='store_true',
                        dest='upgrade', help='Upgrade the virtual '
                                             'environment directory to '
                                             'use this version of '
                                             'Python, assuming Python '
                                             'has been upgraded '
                                             'in-place.')
    parser.add_argument('--verbose', default=False, action='store_true',
                        dest='verbose', help='Display the output '
                                             'from the scripts which '
                                             'install setuptools and pip.')
    options = parser.parse_args(args)
    if options.upgrade and options.clear:
        raise ValueError('you cannot supply --upgrade and --clear together.')
    builder = ExtendedEnvBuilder(system_site_packages=options.system_site,
                                 clear=options.clear,
                                 symlinks=options.symlinks,
                                 upgrade=options.upgrade,
                                 nodist=options.nodist,
                                 nopip=options.nopip,
                                 verbose=options.verbose)
    for d in options.dirs:
        builder.create(d)

if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)
```

이 스크립트는 온라인에서 내려받을 수도 있습니다.

28.3 zipapp — 실행 가능한 파이썬 zip 아카이브 관리

Added in version 3.5.

소스 코드: Lib/zipapp.py

이 모듈은 파이썬 인터프리터가 직접 실행할 수 있는 파이썬 코드를 포함하는 zip 파일 생성을 관리하는 도구를 제공합니다. 이 모듈은 명령 줄 인터페이스와 파이썬 *API*를 모두 제공합니다.

28.3.1 기본 예

다음 예제는 명령 줄 인터페이스를 사용하여 파이썬 코드가 포함된 디렉터리에서 실행 가능 아카이브를 만드는 방법을 보여줍니다. 실행하면, 아카이브가 아카이브의 모듈 `myapp`에서 `main` 함수를 실행합니다.

```
$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
```

28.3.2 명령 줄 인터페이스

명령 줄에서 프로그램으로 호출될 때, 다음 형식이 사용됩니다:

```
$ python -m zipapp source [options]
```

*source*가 디렉터리면, *source*의 내용으로부터 아카이브를 만듭니다. *source*가 파일이면, 아카이브여야 하며, 대상 아카이브로 복사됩니다 (또는 –info 옵션이 지정되면 셔뱅(shebang) 줄의 내용이 표시됩니다).

다음과 같은 옵션이 이해됩니다:

-o <output>, **--output**=<output>

출력을 *output*이라는 이름의 파일에 씁니다. 이 옵션을 지정하지 않으면, 출력 파일명은 입력 *source*와 같고, 확장자 `.pyz`가 추가됩니다. 명시적인 파일명이 제공되면, 그대로 사용됩니다 (그래서 필요하면 `.pyz` 확장자를 포함해야 합니다).

*source*가 아카이브이면 반드시 출력 파일명을 지정해야 합니다 (그리고 이 경우, *output*은 *source*와 달라야 합니다).

-p <interpreter>, **--python**=<interpreter>

실행할 명령으로 *interpreter*를 지정하여 `#!` 줄을 아카이브에 추가합니다. 또한, POSIX에서, 아카이브를 실행 파일로 만듭니다. 기본값은 `#!` 줄을 쓰지 않고, 파일을 실행 파일로 만들지 않는 것입니다.

-m <mainfn>, **--main**=<mainfn>

아카이브에 *mainfn*을 실행하는 `__main__.py` 파일을 씁니다. *mainfn* 인자는 "pkg.mod:fn" 형식이어야 합니다. 여기서 "pkg.mod"는 아카이브의 패키지/모듈이며, "fn"은 주어진 모듈에 있는 콜러블입니다. `__main__.py` 파일은 그 콜러블을 실행합니다.

아카이브를 복사할 때 *--main*을 지정할 수 없습니다.

-c, --compress

디플레이트(deflate) 메서드로 파일을 압축하여, 출력 파일의 크기를 줄입니다. 기본적으로, 파일은 아카이브에 압축되지 않은 상태로 저장됩니다.

아카이브를 복사할 때 *--compress*는 효과가 없습니다.

Added in version 3.7.

--info

진단 목적으로, 아카이브에 내장된 인터프리터를 표시합니다. 이 경우, 다른 옵션은 무시되고 SOURCE는 디렉터리가 아닌 아카이브여야 합니다.

-h, --help

간단한 사용법 메시지를 인쇄하고 종료합니다.

28.3.3 파이썬 API

이 모듈은 두 개의 편의 함수를 정의합니다:

zipapp.**create_archive**(*source, target=None, interpreter=None, main=None, filter=None, compressed=False*)

*source*로 응용 프로그램 아카이브를 만듭니다. 소스는 다음 중 하나일 수 있습니다:

- 디렉터리 이름, 또는 디렉터리를 참조하는 경로류 객체. 이 경우 해당 디렉터리의 내용으로 새 응용 프로그램 아카이브가 만들어집니다.
- 기존 응용 프로그램 아카이브 파일의 이름, 또는 이러한 파일을 참조하는 경로류 객체. 이 경우 파일이 대상(target)으로 복사됩니다 (*interpreter* 인자에 제공된 값을 반영하도록 수정하면서). 필요하면, 파일 이름에 `.pyz` 확장자가 포함되어야 합니다.
- 바이너리 모드에서 읽기로 열린 파일 객체. 파일의 내용은 응용 프로그램 아카이브여야 하며, 파일 객체는 아카이브의 시작 부분에 위치한 것으로 가정합니다.

target 인자는 결과 아카이브가 기록될 위치를 결정합니다:

- 파일 이름이거나 경로류 객체이면, 아카이브가 그 파일에 기록됩니다.
- 열린 파일 객체면, 그 파일 객체에 아카이브가 기록되며, 파일은 바이너리 모드로 쓰기로 열려 있어야 합니다.
- target이 생략되면 (또는 `None`이면), 소스(source)는 디렉터리여야 하며 대상은 `.pyz` 확장자가 추가된 소스와 이름이 같은 파일이 됩니다.

interpreter 인자는 아카이브가 실행될 파이썬 인터프리터의 이름을 지정합니다. 아카이브 시작 부분에 "셔뱅(shebang)" 줄로 기록됩니다. POSIX에서는 이를 OS가 해석하고, 윈도우에서는 파이썬 런처가 이를 처리합니다. *interpreter*를 생략하면 셔뱅 줄이 기록되지 않습니다. interpreter가 지정되고 target이 파일명이면, target 파일의 실행 가능 비트가 설정됩니다.

main 인자는 아카이브의 메인 프로그램으로 사용될 콜러블의 이름을 지정합니다. 소스가 디렉터리이고 소스에 아직 `__main__.py` 파일이 없을 때만 지정할 수 있습니다. *main* 인자는 "pkg.module:callable" 형식이어야 하며 아카이브는 "pkg.module"을 임포트 하고 인자 없이 지정된 콜러블을 실행해서 실행됩니다. 소스가 디렉터리이고 `__main__.py` 파일을 포함하지 않을 때 *main*을 생략하는 것은 에러입니다, 그렇지 않으면 결과 아카이브가 실행 가능하지 않기 때문입니다.

선택적 *filter* 인자는 추가되는 파일의 (소스 디렉터리에 상대적인) 경로를 나타내는 Path 객체가 전달되는 콜백 함수를 지정합니다. 파일을 추가하려면 `True`를 반환해야 합니다.

선택적 *compressed* 인자는 파일을 압축할지를 결정합니다. `True`로 설정하면, 아카이브의 파일이 디플레이트(deflate) 메서드로 압축됩니다; 그렇지 않으면 파일이 압축되지 않은 상태로 저장됩니다. 기존 아카이브를 복사할 때는 이 인자가 효과가 없습니다.

*source*나 *target*에 파일 객체가 지정되면, create_archive를 호출한 후 파일 객체를 닫는 것은 호출자의 책임입니다.

기존 아카이브를 복사할 때, 제공된 파일 객체에는 `read`와 `readline` 또는 `write` 메서드만 필요합니다. 디렉터리에서 아카이브를 만들 때, 대상이 파일 객체면 `zipfile.ZipFile` 클래스로 전달되며, 해당 클래스에 필요한 메서드를 제공해야 합니다.

버전 3.7에서 변경: *filter*와 *compressed* 매개 변수를 추가했습니다.

zipapp.**get_interpreter**(*archive*)

아카이브 시작 부분에서 `#!` 줄에 지정된 인터프리터를 반환합니다. `#!` 줄이 없으면, *None*을 반환합니다. *archive* 인자는 파일명이나 바이너리 모드로 읽기로 열린 파일류 객체일 수 있습니다. 아카이브가 시작 부분에 위치한 것으로 가정합니다.

28.3.4 예

디렉터리를 아카이브로 패킹하고, 실행합니다.

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

create_archive() 함수를 사용하여 같은 작업을 수행할 수 있습니다:

```
>>> import zipapp
>>> zipapp.create_archive('myapp', 'myapp.pyz')
```

POSIX에서 응용 프로그램을 직접 실행할 수 있게 만들려면, 사용할 인터프리터를 지정하십시오.

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

기존 아카이브에서 셔뱅 줄을 바꾸려면, *create_archive()* 함수를 사용하여 수정된 아카이브를 만드십시오:

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz', '/usr/bin/python3')
```

파일을 제자리에서 갱신하려면, *BytesIO* 객체를 사용하여 메모리에서 치환을 수행한 다음, 나중에 소스를 덮어씁니다. 파일을 제자리에서 덮어쓰면 에러로 인해 원본 파일을 손실할 위험이 있음에 유의하십시오. 이 코드는 이러한 에러에 대한 보호는 없지만, 프로덕션 코드는 이를 방지해야 합니다. 또한, 이 방법은 아카이브가 메모리에 올라올 수 있을 때만 작동합니다:

```
>>> import zipapp
>>> import io
>>> temp = io.BytesIO()
>>> zipapp.create_archive('myapp.pyz', temp, '/usr/bin/python2')
>>> with open('myapp.pyz', 'wb') as f:
>>>     f.write(temp.getvalue())
```

28.3.5 인터프리터 지정하기

인터프리터를 지정한 다음 응용 프로그램 아카이브를 배포한다면, 사용된 인터프리터가 이식성 있는지 확인할 필요가 있음에 유의하십시오. 윈도우 용 파이썬 런처는 가장 일반적인 POSIX `#!` 줄 형식을 지원하지만, 고려해야 할 다른 문제가 있습니다:

- "/usr/bin/env python" (또는 "/usr/bin/python" 과 같은 "python" 명령의 다른 형식)을 사용한다면, 사용자에게 기본적으로 파이썬 2나 파이썬 3이 있을 수 있음을 고려해야 하고, 두 버전에서 작동하도록 코드를 작성하십시오.
- "/usr/bin/env python3" 과 같이 명시적인 버전을 사용하면 해당 버전이 없는 사용자에게는 응용 프로그램이 작동하지 않습니다. (여러분의 코드를 파이썬 2 호환으로 만들지 않았다면 이것이 여러분이 원하는 것일 수 있습니다).
- "파이썬 X.Y 이상" 이라고 말할 방법이 없어서, "/usr/bin/env python3.4" 처럼 정확한 버전을 사용할 때는 주의하십시오. 예를 들어 파이썬 3.5 사용자를 위해서는 셔뱅 줄을 변경해야 합니다.

일반적으로, 여러분의 코드가 파이썬 2나 3중 어느 것으로 작성되었는지에 따라 "/usr/bin/env python2" 나 "/usr/bin/env python3" 을 사용해야 합니다.

28.3.6 zipapp으로 독립형 응용 프로그램 만들기

zipapp 모듈을 사용하면, 시스템에 적합한 버전의 파이썬만 설치되어있는 최종 사용자에게 배포 할 수 있는 필요한 모든 것이 담긴 파이썬 프로그램을 만들 수 있습니다. 이 작업의 핵심은 응용 프로그램 코드와 함께 모든 응용 프로그램의 종속성을 아카이브에 묶는 것입니다.

독립형 아카이브를 만드는 단계는 다음과 같습니다:

1. 정상적으로 디렉터리에 응용 프로그램을 만드십시오, 그러면 `__main__.py` 파일과 모든 지원 응용 프로그램 코드를 포함하는 `myapp` 디렉터리를 얻게 됩니다.
2. pip을 사용하여 모든 응용 프로그램의 종속성을 `myapp` 디렉터리에 설치하십시오:

```
$ python -m pip install -r requirements.txt --target myapp
```

(이것은 requirements.txt 파일에 프로젝트 요구 사항이 있다고 가정합니다 - 그렇지 않으면, pip 명령 줄에 종속성을 수동으로 나열할 수 있습니다).

3. 다음과 같이 응용 프로그램을 패키징하십시오:

```
$ python -m zipapp -p "interpreter" myapp
```

그러면 독립형 실행 파일이 생성되며, 사용 가능한 적절한 인터프리터가 있는 모든 시스템에서 실행할 수 있습니다. 자세한 내용은 인터프리터 지정하기를 참조하십시오. 단일 파일로 사용자에게 제공될 수 있습니다.

유닉스에서, myapp.pyz 파일은 그대로 실행 파일입니다. "평범한" 명령 이름을 선호하면, 파일 이름을 변경하여 .pyz 확장자를 제거할 수 있습니다. 윈도우에서, 파이썬 인터프리터가 설치될 때 .pyz와 .pyzw 파일 확장자를 등록한다는 점에서 myapp.pyz[w] 파일은 실행 파일입니다.

경고

응용 프로그램이 C 확장을 포함하는 패키지에 의존하면, 해당 패키지는 zip 파일에서 실행할 수 없습니다 (이것은 OS 제한 사항인데, OS 로더가 로드 할 수 있으려면 실행 코드는 파일 시스템에 있어야만 합니다). 이 경우, zip 파일에서 해당 종속성을 제외하고, 사용자가 파일을 설치하도록 요구하거나, zip 파일과 함께 제공하고 __main__.py에 코드를 추가하여 sys.path에 압축 해제된 모듈을 포함하는 디렉터리를 포함할 수 있습니다. 이 경우, 대상 아키텍처에 적합한 바이너리를 제공해야 합니다 (그리고 아마도 사용자 컴퓨터를 기반으로 실행 시간에 sys.path에 추가할 올바른 버전을 선택해야 합니다).

28.3.7 파이썬 Zip 응용 프로그램 아카이브 형식

파이썬은 버전 2.6부터 __main__.py 파일이 포함된 zip 파일을 실행할 수 있었습니다. 파이썬에서 실행되려면, 응용 프로그램 아카이브는 응용 프로그램의 진입점으로 실행될 __main__.py 파일이 포함된 표준 zip 파일이어야 합니다. 모든 파이썬 스크립트와 마찬가지로, 스크립트의 부모(이 경우 zip 파일)가 *sys.path*에 배치되므로 zip 파일에서 추가 모듈을 임포트 할 수 있습니다.

zip 파일 형식은 임의의 데이터를 zip 파일 앞에 추가할 수 있도록 합니다. zip 응용 프로그램 형식은 이 기능을 사용하여 표준 POSIX "셔뱅(shebang)" 줄을 파일 앞에 추가합니다 (#!/path/to/interpreter).

따라서 공식적으로 파이썬 zip 응용 프로그램 형식은 다음과 같습니다:

1. 문자 b'#!', 인터프리터 이름, 개행 (b'\n') 문자를 포함하는 선택적 셔뱅(shebang) 줄. 인터프리터 이름은 OS "셔뱅" 처리가 허용하는 모든 것, 또는 윈도우의 파이썬 런처일 수 있습니다. 인터프리터는 윈도우에서는 UTF-8로, POSIX에서는 *sys.getfilesystemencoding()*으로 인코딩되어야 합니다.
2. *zipfile* 모듈에 의해 생성된 표준 zip 파일 데이터. zipfile 내용은 반드시 __main__.py라는 파일을 포함해야 합니다 (zip 파일의 "루트"에 있어야 합니다 - 즉, 서브 디렉터리에 있을 수 없습니다). zip 파일 데이터는 압축되거나 그렇지 않을 수 있습니다.

응용 프로그램 아카이브에 셔뱅 줄이 있으면, POSIX 시스템에서 직접 실행될 수 있도록 실행 파일 비트를 설정할 수 있습니다.

이 모듈의 도구를 사용하여 응용 프로그램 아카이브를 만들 필요는 없습니다 - 모듈은 편의를 위한 것입니다. 하지만 어떤 방법으로 만들었든 위 형식의 아카이브는 파이썬에서 허용됩니다.

CHAPTER 29

파이썬 실행시간 서비스

이 장에서 설명하는 모듈들은 파이썬 인터프리터와 그 환경과의 상호 작용과 관련된 다양한 서비스를 제공합니다. 다음은 개요입니다:

29.1 sys — 시스템 특정 파라미터와 함수

이 모듈은 인터프리터에 의해 사용되거나 유지되는 일부 변수와 인터프리터와 강하게 상호 작용하는 함수에 대한 액세스를 제공합니다. 항상 사용 가능합니다. 명시적으로 달리 언급하지 않는 한, 모든 변수는 읽기 전용입니다.

sys.**abiflags**

표준 `configure` 스크립트를 사용하여 파이썬을 빌드한 POSIX 시스템에서, 이것은 **PEP 3149**에 지정된 ABI 플래그를 포함합니다.

Added in version 3.2.

버전 3.8에서 변경: 기본 플래그는 빈 문자열이 되었습니다 (pymalloc을 위한 `m` 플래그가 제거되었습니다).

가용성: Unix.

sys.**addaudithook**(*hook*)

현재 (서브) 인터프리터의 활성 감사 훅 리스트에 콜러블 *hook*을 추가합니다.

sys.audit() 함수를 통해 감사 이벤트가 발생하면, 각 훅은 추가된 순서대로 이벤트 이름과 인자 튜플로 호출됩니다. `PySys_AddAuditHook()`으로 추가된 네이티브 훅이 먼저 호출되고, 현재 (서브) 인터프리터에서 추가된 훅이 뒤따릅니다. 그런 다음 훅은 이벤트를 로그 하거나, 예외를 발생 시켜 연산을 중단하거나, 프로세스를 완전히 종료 할 수 있습니다.

Note that audit hooks are primarily for collecting information about internal or otherwise unobservable actions, whether by Python or libraries written in Python. They are not suitable for implementing a "sandbox". In particular, malicious code can trivially disable or bypass hooks added using this function. At a minimum, any security-sensitive hooks must be added using the C API `PySys_AddAuditHook()` before initialising the runtime, and any modules allowing arbitrary memory modification (such as *ctypes*) should be completely removed or closely monitored.

*sys.addaudithook()*을 호출하면 스스로 인자 없이 `sys.addaudithook`이라는 이름의 감사 이벤트를 발생시킵니다. 기존 훅에서 *RuntimeError*에서 파생된 예외가 발생하면, 새 훅이 추가되지 않고

예외가 억제됩니다. 결과적으로, 호출자는 기존의 모든 훅을 제어하지 않는 한 훅이 추가되었다고 가정할 수 없습니다.

CPython이 발생시키는 모든 이벤트에 대해서는 감사 이벤트 표를, 원래 디자인 논의는 **PEP 578**을 참조하십시오.

Added in version 3.8.

버전 3.8.1에서 변경: *Exception*에서 파생되었지만 *RuntimeError*가 아닌 예외는 더는 억제되지 않습니다.

추적(tracing)이 활성화되었을 때 (*settrace()*를 참조하십시오), 파이썬 훅은 콜러블에 참값으로 설정된 `__cantrace__` 멤버가 있을 때만 추적합니다. 그렇지 않으면, 추적 함수는 훅을 건너뜁니다.

`sys.`**`argv`**

파이썬 스크립트에 전달된 명령 줄 인자 리스트. `argv[0]`은 스크립트 이름입니다 (전체 경로명인지는 운영 체제에 따라 다릅니다). 인터프리터에 `-c` 명령 줄 옵션을 사용하여 명령이 실행되었으면, `argv[0]`은 문자열 `'-c'`로 설정됩니다. 파이썬 인터프리터에 스크립트 이름이 전달되지 않으면, `argv[0]`은 빈 문자열입니다.

표준 입력이나 명령 줄에 제공된 파일 목록을 루핑 하려면, *fileinput* 모듈을 참조하십시오.

See also *sys.orig_argv*.

> **참고**
>
> 유닉스에서, 명령 줄 인자는 OS에서 바이트열로 전달됩니다. 파이썬은 이것들을 파일 시스템 인코딩과 "surrogateescape" 에러 처리기로 디코딩합니다. 원본 바이트열이 필요할 때, `[os.fsencode(arg) for arg in sys.argv]`로 얻을 수 있습니다.

`sys.`**`audit`**(*event*, **args*)

활성 감사 훅으로 감사 이벤트를 발생시킵니다. *event*는 이벤트를 식별하는 문자열이고, *args*는 이벤트에 대한 추가 정보를 갖는 선택적 인자를 포함할 수 있습니다. 주어진 이벤트에 대한 인자의 수와 형은 공개된 안정 API로 간주하며 배포 간에 수정되지 않아야 합니다.

예를 들어, 한 감사 이벤트의 이름은 `os.chdir`입니다. 이 이벤트에는 요청된 새 작업 디렉터리를 포함하는 *path*라는 인자가 하나 있습니다.

*sys.audit()*는 이벤트 이름과 인자들을 전달하여 기존 감사 훅들을 호출하고, 어떤 훅에서건 발생한 첫 번째 예외를 발생시킵니다. 일반적으로, 예외가 발생하면, 처리되지 않아야 하며 가능한 한 빨리 프로세스를 종료해야 합니다. 이렇게 하면 훅 구현이 특정 이벤트에 반응하는 방법을 결정할 수 있습니다: 단순히 이벤트를 로그 하거나 예외를 발생 시켜 연산을 중단 할 수 있습니다.

훅은 *sys.addaudithook()*이나 `PySys_AddAuditHook()` 함수를 사용하여 추가됩니다.

이 함수의 네이티브 동등 물은 `PySys_Audit()` 입니다. 가능하면 네이티브 함수를 사용하는 것이 좋습니다.

CPython이 발생시키는 모든 이벤트에 대해서는 감사 이벤트 표를 참조하십시오.

Added in version 3.8.

`sys.`**`base_exec_prefix`**

Equivalent to *exec_prefix*, but referring to the base Python installation.

When running under *Virtual Environments*, *exec_prefix* gets overwritten to the virtual environment prefix. *base_exec_prefix*, conversely, does not change, and always points to the base Python installation. Refer to *Virtual Environments* for more information.

Added in version 3.3.

`sys.`**`base_prefix`**

Equivalent to *prefix*, but referring to the base Python installation.

When running under *virtual environment*, `prefix` gets overwritten to the virtual environment prefix. `base_prefix`, conversely, does not change, and always points to the base Python installation. Refer to *Virtual Environments* for more information.

Added in version 3.3.

`sys.`**`byteorder`**

네이티브 바이트 순서의 표시기. 이는 빅 엔디안(최상위 바이트 먼저) 플랫폼에서 `'big'` 값을, 리틀 엔디안(최하위 바이트 먼저) 플랫폼에서 `'little'`을 갖습니다.

`sys.`**`builtin_module_names`**

이 파이썬 인터프리터로 컴파일된 모든 모듈의 이름을 포함하는 문자열의 튜플. (이 정보는 다른 방법으로는 얻을 수 없습니다 — `modules.keys()`는 임포트 된 모듈만 나열합니다.)

See also the `sys.stdlib_module_names` list.

`sys.`**`call_tracing`**(*func*, *args*)

추적이 활성화된 동안, `func(*args)`를 호출합니다. 추적 상태가 저장되고, 나중에 복원됩니다. 이것은 체크 포인트에서 디버거에서 호출되어 다른 코드를 재귀적으로 디버깅이나 프로파일링하기 위한 것입니다.

Tracing is suspended while calling a tracing function set by `settrace()` or `setprofile()` to avoid infinite recursion. `call_tracing()` enables explicit recursion of the tracing function.

`sys.`**`copyright`**

파이썬 인터프리터와 관련된 저작권이 포함된 문자열.

`sys.`**`_clear_type_cache`**`()`

내부 형 캐시를 지웁니다. 형 캐시는 어트리뷰트와 메서드 조회 속도를 높이는 데 사용됩니다. 참조 누수 디버깅 중에 불필요한 참조를 제거하기 위해서 만 이 함수를 사용하십시오.

이 함수는 내부와 특수 목적으로만 사용해야 합니다.

버전 3.13부터 폐지됨: Use the more general `_clear_internal_caches()` function instead.

`sys.`**`_clear_internal_caches`**`()`

Clear all internal performance-related caches. Use this function *only* to release unnecessary references and memory blocks when hunting for leaks.

Added in version 3.13.

`sys.`**`_current_frames`**`()`

함수가 호출될 때 각 스레드의 식별자를 해당 스레드에서 현재 활성화된 최상위 스택 프레임에 매핑하는 딕셔너리를 반환합니다. `traceback` 모듈의 함수는 이러한 프레임을 주면 호출 스택을 빌드할 수 있음에 유의하십시오.

이것은 교착 상태 디버깅에 가장 유용합니다: 이 함수는 교착 상태에 빠진 스레드의 협력을 필요로 하지 않으며, 이러한 스레드의 호출 스택은 교착 상태를 유지하는 한 고정됩니다. 교착 상태가 아닌 스레드에 대해 반환된 프레임은 호출하는 코드가 프레임을 검사할 때까지 해당 스레드의 현재 활동과 관련이 없을 수 있습니다.

이 함수는 내부와 특수 목적으로만 사용해야 합니다.

인자 없이 감사 이벤트 `sys._current_frames`를 발생시킵니다.

`sys.`**`_current_exceptions`**`()`

함수가 호출될 때 각 스레드의 식별자를 해당 스레드에서 현재 활성화된 최상위 예외에 매핑하는 딕셔너리를 반환합니다. 스레드가 현재 예외를 처리하고 있지 않으면, 결과 딕셔너리에 포함되지 않습니다.

This is most useful for statistical profiling.

이 함수는 내부와 특수 목적으로만 사용해야 합니다.

인자 없이 감사 이벤트 `sys._current_exceptions`를 발생시킵니다.

버전 3.12에서 변경: Each value in the dictionary is now a single exception instance, rather than a 3-tuple as returned from `sys.exc_info()`.

`sys.breakpointhook()`

이 훅 함수는 내장 *breakpoint()*에 의해 호출됩니다. 기본적으로, *pdb* 디버거로 떨어뜨리지만, 다른 함수로 설정하여 사용할 디버거를 선택할 수 있습니다.

이 함수의 서명은 그것이 호출하는 것에 의존합니다. 예를 들어, 기본 바인딩(예를 들어 `pdb.set_trace()`)에는 인자가 필요하지 않지만, 추가 인자(위치 및/또는 키워드)를 기대하는 함수에 바인딩할 수 있습니다. 내장 `breakpoint()` 함수는 `*args`와 `**kws`를 그대로 전달합니다. `breakpointhooks()`가 반환하는 것이 `breakpoint()`에서 반환됩니다.

기본 구현은 먼저 환경 변수 `PYTHONBREAKPOINT`를 참조합니다. 이것이 `"0"`으로 설정되면, 이 함수는 즉시 반환합니다; 즉, 아무런 일도 하지 않습니다. 환경 변수가 설정되지 않거나 빈 문자열로 설정되면, `pdb.set_trace()`가 호출됩니다. 그렇지 않으면 이 변수는 파이썬의 점으로 표현한 임포트 표기법(예를 들어 `package.subpackage.module.function`)을 사용하여 실행할 함수의 이름을 지정해야 합니다. 이 경우, `package.subpackage.module`이 임포트 되고 결과 모듈에는 `function()`이라는 이름의 콜러블이 있어야 합니다. 이것이 `*args` 및 `**kws`를 전달하여 실행되며, `function()`이 반환하는 것을 `sys.breakpointhook()`이 내장 *breakpoint()* 함수로 반환합니다.

`PYTHONBREAKPOINT`로 이름이 지정된 콜러블을 임포트 하는 도중에 문제가 발생하면, *RuntimeWarning*이 보고되고 중단점은 무시됨에 유의하십시오.

또한 `sys.breakpointhook()`이 프로그래밍 방식으로 재정의되면, `PYTHONBREAKPOINT`는 참조되지 않음에 유의하십시오.

Added in version 3.7.

`sys._debugmallocstats()`

CPython의 메모리 할당자 상태에 대한 저수준 정보를 stderr에 인쇄합니다.

파이썬이 디버그 모드로 빌드되었으면 (`configure --with-pydebug option`), 값비싼 내부 일관성 검사도 수행합니다.

Added in version 3.3.

이 함수는 CPython 전용입니다. 정확한 출력 형식은 여기에 정의되어 있지 않으며, 변경될 수 있습니다.

`sys.dllhandle`

파이썬 DLL의 핸들을 지정하는 정수.

가용성: Windows.

`sys.displayhook(value)`

*value*가 `None`이 아니면, 이 함수는 `repr(value)`를 `sys.stdout`으로 인쇄하고, *value*를 `builtins._`에 저장합니다. `repr(value)`를 `sys.stdout.errors` 에러 처리기(아마도 `'strict'`)로 `sys.stdout.encoding`으로 인코딩할 수 없으면, `'backslashreplace'` 에러 처리기를 사용하여 `sys.stdout.encoding`으로 인코딩합니다.

`sys.displayhook`은 대화형 파이썬 세션에서 입력한 표현식을 평가한 결과에 대해 호출됩니다. `sys.displayhook`에 다른 1-인자 함수를 대입하여 이러한 값의 표시를 사용자 정의할 수 있습니다.

의사 코드:

```
def displayhook(value):
    if value is None:
        return
    # 재귀를 방지하기 위해 '_' 를 None으로 설정합니다
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

버전 3.2에서 변경: *UnicodeEncodeError* 에 `'backslashreplace'` 에러 처리기를 사용합니다.

`sys.`**`dont_write_bytecode`**

이것이 참이면, 파이썬은 소스 모듈을 임포트 할 때 `.pyc` 파일을 쓰려고 하지 않습니다. 이 값은 `-B` 명령 줄 옵션과 `PYTHONDONTWRITEBYTECODE` 환경 변수에 따라 초기에 `True`나 `False`로 설정되지만, 바이트 코드 파일 생성을 제어하기 위해 직접 설정할 수 있습니다.

`sys.`**`_emscripten_info`**

A *named tuple* holding information about the environment on the *wasm32-emscripten* platform. The named tuple is provisional and may change in the future.

`_emscripten_info.`**`emscripten_version`**

Emscripten version as tuple of ints (major, minor, micro), e.g. `(3, 1, 8)`.

`_emscripten_info.`**`runtime`**

Runtime string, e.g. browser user agent, `'Node.js v14.18.2'`, or `'UNKNOWN'`.

`_emscripten_info.`**`pthreads`**

`True` if Python is compiled with Emscripten pthreads support.

`_emscripten_info.`**`shared_memory`**

`True` if Python is compiled with shared memory support.

가용성: Emscripten.

Added in version 3.11.

`sys.`**`pycache_prefix`**

이것이 설정되면 (`None`이 아니면), 파이썬은 바이트 코드 캐시 `.pyc` 파일을 소스 코드 트리의 `__pycache__` 디렉터리가 아니라, 이 디렉터리를 루트로 하는 병렬 디렉터리 트리에 씁니다 (그리고 그곳에서 읽습니다). 소스 코드 트리의 모든 `__pycache__` 디렉터리는 무시되고 새 `.pyc` 파일이 이 디렉터리 내에 기록됩니다. 따라서 *compileall*을 사전 빌드 단계로 사용한다면, 실행 시간에 사용할 같은 pycache 접두어(있다면)로 실행해야 합니다.

상대 경로는 현재 작업 디렉터리를 기준으로 해석됩니다.

이 값은 초기에 `-X pycache_prefix=PATH` 명령 줄 옵션이나 `PYTHONPYCACHEPREFIX` 환경 변수(명령 줄이 우선합니다)의 값을 기반으로 설정됩니다. 둘 다 설정되지 않으면, `None`입니다.

Added in version 3.8.

`sys.`**`excepthook`**(*type*, *value*, *traceback*)

이 함수는 주어진 트레이스백(traceback)과 예외를 `sys.stderr`에 인쇄합니다.

SystemExit 이외의 예외가 발생하고 잡히지 않을 때, 인터프리터는 세 인자, 예외 클래스, 예외 인스턴스 및 트레이스백 객체로 `sys.excepthook`을 호출합니다. 대화식 세션에서는 제어가 프롬프트로 반환되기 직전에 일어납니다; 파이썬 프로그램에서는 프로그램이 종료되기 직전에 일어납니다. 이러한 최상위 예외 처리는 `sys.excepthook`에 다른 3-인자 함수를 대입하여 사용자 정의할 수 있습니다.

잡히지 않은 예외가 발생할 때 인자 `hook`, `type`, `value`, `traceback`로 감사 이벤트 `sys.excepthook`을 발생시킵니다. 아무런 훅도 설정되지 않았으면, `hook`은 `None` 일 수 있습니다. 훅이 *RuntimeError*에서 파생된 예외를 발생시키면 훅 호출이 억제됩니다. 그렇지 않으면, 감사 훅 예외는 발생시킬 수 없는 것(unraisable)으로 보고되고 `sys.excepthook`이 호출됩니다.

> **더 보기**
>
> *sys.unraisablehook()* 함수는 발생시킬 수 없는 예외(unraisable exception)를 다루고 *threading.excepthook()* 함수는 *threading.Thread.run()*이 발생시킨 예외를 다룹니다.

sys.**__breakpointhook__**

sys.**__displayhook__**

sys.**__excepthook__**

sys.**__unraisablehook__**

이러한 객체는 프로그램 시작 시 breakpointhook, displayhook, excepthook 및 unraisablehook의 원래 값을 포함합니다. breakpointhook, displayhook 및 excepthook, unraisablehook 이 파손되거나 대체 객체로 교체될 때 복원 할 수 있도록 저장됩니다.

Added in version 3.7: __breakpointhook__

Added in version 3.8: __unraisablehook__

sys.**exception**()

This function, when called while an exception handler is executing (such as an except or except* clause), returns the exception instance that was caught by this handler. When exception handlers are nested within one another, only the exception handled by the innermost handler is accessible.

If no exception handler is executing, this function returns None.

Added in version 3.11.

sys.**exc_info**()

This function returns the old-style representation of the handled exception. If an exception e is currently handled (so *exception()* would return e), *exc_info()* returns the tuple (type(e), e, e.__traceback__). That is, a tuple containing the type of the exception (a subclass of *BaseException*), the exception itself, and a traceback object which typically encapsulates the call stack at the point where the exception last occurred.

If no exception is being handled anywhere on the stack, this function return a tuple containing three None values.

버전 3.11에서 변경: The type and traceback fields are now derived from the value (the exception instance), so when an exception is modified while it is being handled, the changes are reflected in the results of subsequent calls to *exc_info()*.

sys.**exec_prefix**

플랫폼 특정 파이썬 파일이 설치되는 사이트 특정 디렉터리 접두사를 제공하는 문자열; 기본적으로, 이것은 '/usr/local'이기도 합니다. **configure** 스크립트에 --exec-prefix 인자를 사용하여 빌드 시 설정할 수 있습니다. 구체적으로, 모든 구성 파일(예를 들어 pyconfig.h 헤더 파일)은 디렉터리 *exec_prefix*/lib/python*X.Y*/config에 설치되고, 공유 라이브러리 모듈은 *exec_prefix*/lib/python*X.Y*/lib-dynload에 설치됩니다. 여기서 *X.Y*는 파이썬의 버전 번호입니다, 예를 들어 3.2.

> **참고**
>
> If a *virtual environment* is in effect, this *exec_prefix* will point to the virtual environment. The value for the Python installation will still be available, via *base_exec_prefix*. Refer to *Virtual Environments* for more information.

버전 3.14에서 변경: When running under a *virtual environment*, *prefix* and *exec_prefix* are now set to the virtual environment prefix by the *path initialization*, instead of *site*. This means that *prefix* and *exec_prefix* always point to the virtual environment, even when *site* is disabled (-S).

sys.**executable**

의미가 있는 시스템에서, 파이썬 인터프리터를 위한 실행 바이너리의 절대 경로를 제공하는 문자열. 파이썬이 실행 파일의 실제 경로를 검색할 수 없으면, *sys.executable*은 빈 문자열이거나 None 입니다.

sys.**exit**([*arg*])

Raise a *SystemExit* exception, signaling an intention to exit the interpreter.

선택적 인자 *arg*는 종료 상태(기본값은 0)를 나타내는 정수나 다른 형의 객체일 수 있습니다. 정수면, 0은 "성공적인 종료"로 간주하고, 0이 아닌 값은 셸 등이 "비정상 종료"로 간주합니다. 대부분 시스템은 0–127 범위를 요구하고, 그렇지 않으면 정의되지 않은 결과를 생성합니다. 일부 시스템에는 특정 종료 코드에 특정 의미를 지정하는 규칙이 있지만, 일반적으로 잘 개발되지 않은 상태입니다; 유닉스 프로그램은 일반적으로 명령 줄 구문 에러에 2를, 다른 모든 종류의 에러에 1을 사용합니다. 다른 형의 객체가 전달되면, None은 0을 전달하는 것과 동등하며, 다른 모든 객체는 *stderr*로 인쇄되고 종료 코드 1을 만듭니다. 특히, sys.exit("some error message")는 에러가 발생할 때 프로그램을 종료하는 빠른 방법입니다.

*exit()*는 궁극적으로 "단지" 예외를 발생시키기만 하므로, 메인 스레드에서 호출되고 예외를 가로채지 않을 때만 프로세스를 종료합니다. try 문에서 finally 절로 지정된 정리 작업은 수행되며, 바깥 수준에서의 종료 시도를 가로챌 수 있습니다.

버전 3.6에서 변경: 파이썬 인터프리터가 *SystemExit*를 잡은 후 정리할 때 에러가 발생하면 (가령 표준 스트림에 버퍼링 된 데이터를 플러시할 때의 에러), 종료 상태가 120으로 변경됩니다.

sys.**flags**

The *named tuple flags* exposes the status of command line flags. Flags should only be accessed only by name and not by index. The attributes are read only.

flags.**debug**	-d
flags.**inspect**	-i
flags.**interactive**	-i
flags.**isolated**	-I
flags.**optimize**	-O 또는 -OO
flags.**dont_write_bytecode**	-B
flags.**no_user_site**	-s
flags.**no_site**	-S
flags.**ignore_environment**	-E
flags.**verbose**	-v
flags.**bytes_warning**	-b
flags.**quiet**	-q
flags.**hash_randomization**	-R
flags.**dev_mode**	-X dev (파이썬 개발 모드)
flags.**utf8_mode**	-X utf8
flags.**safe_path**	-P
flags.**int_max_str_digits**	-X int_max_str_digits (*integer string conversion length limitation*)
flags.**warn_default_encoding**	-X warn_default_encoding
flags.**gil**	-X gil and PYTHON_GIL
flags.**thread_inherit_context**	-X thread_inherit_context and PYTHON_THREAD_INHERIT_CONTEXT
flags.**context_aware_warnings**	-X context_aware_warnings and PYTHON_CONTEXT_AWARE_WARNINGS

버전 3.2에서 변경: 새로운 `-q` 플래그에 대한 `quiet` 어트리뷰트가 추가되었습니다.

Added in version 3.2.3: `hash_randomization` 어트리뷰트.

버전 3.3에서 변경: 사용되지 않는 `division_warning` 어트리뷰트를 제거했습니다.

버전 3.4에서 변경: `-I` `isolated` 플래그에 대한 `isolated` 어트리뷰트가 추가되었습니다.

버전 3.7에서 변경: 새로운 파이썬 개발자 모드에 대한 `dev_mode` 어트리뷰트와 새로운 `-X` `utf8` 플래그에 대한 `utf8_mode` 어트리뷰트가 추가되었습니다.

버전 3.10에서 변경: `-X` `warn_default_encoding` 플래그에 대한 `warn_default_encoding` 어트리뷰트가 추가되었습니다.

버전 3.11에서 변경: `-P` 옵션 대한 `safe_path` 어트리뷰트가 추가되었습니다.

버전 3.11에서 변경: `int_max_str_digits` 어트리뷰트가 추가되었습니다.

버전 3.13에서 변경: Added the `gil` attribute.

버전 3.14에서 변경: Added the `thread_inherit_context` attribute.

버전 3.14에서 변경: Added the `context_aware_warnings` attribute.

`sys.`**`float_info`**

float 형에 대한 정보를 담은 네임드 튜플. 정밀도와 내부 표현에 대한 저수준 정보를 포함합니다. 값은 'C' 프로그래밍 언어의 표준 헤더 파일 `float.h`에 정의된 다양한 부동 소수점 상수에 해당합니다; 자세한 내용은 1999 ISO/IEC C 표준 [C99] 의 섹션 5.2.4.2.2 'Characteristics of floating types' 를 참조하십시오.

표 1: Attributes of the `float_info` named tuple

어트리뷰트	float.h 매크로	설명
`float_info.`**`epsilon`**	`DBL_EPSILON`	1.0과 부동 소수점으로 표현할 수 있는 1.0보다 큰 최솟값의 차이. *`math.ulp()`*도 참조하십시오.
`float_info.`**`dig`**	`DBL_DIG`	부동 소수점으로 충실하게 표현할 수 있는 최대 십진 숫자의 개수; 아래를 참조하십시오.
`float_info.`**`mant_dig`**	`DBL_MANT_DIG`	float 정밀도: float의 유효 숫자에서 기본-`radix` 자릿수.
`float_info.`**`max`**	`DBL_MAX`	최대 표현 가능한 양의 유한 float.
`float_info.`**`max_exp`**	`DBL_MAX_EXP`	`radix**(e-1)` 이 표현 가능한 유한 float가 되도록 하는 최대 정수 *e*.
`float_info.`**`max_10_exp`**	`DBL_MAX_10_EXP`	`10**e`가 표현 가능한 유한 float의 범위에 있도록 하는 최대 정수 *e*.
`float_info.`**`min`**	`DBL_MIN`	최소 표현 가능한 양의 (positive) 정규화된 float. *`math.ulp(0.0)`*를 사용하여 가장 작은 양의 정규화되지 않은 (*denormalized*) 표현 가능한 float를 얻습니다.
`float_info.`**`min_exp`**	`DBL_MIN_EXP`	`radix**(e-1)` 이 정규화된 float가 되도록 하는 최소 정수 *e*.
`float_info.`**`min_10_exp`**	`DBL_MIN_10_EXP`	`10**e`가 정규화된 float가 되도록 하는 최소 정수 *e*.
`float_info.`**`radix`**	`FLT_RADIX`	지수 표현의 기수.
`float_info.`**`rounds`**	`FLT_ROUNDS`	부동 소수점 산술을 위한 자리 올림 모드를 나타내는 정수. 이는 인터프리터 시작 시 시스템 `FLT_ROUNDS` 매크로의 값을 반영합니다: • `-1`: indeterminable • `0`: toward zero • `1`: to nearest • `2`: toward positive infinity • `3`: toward negative infinity All other values for `FLT_ROUNDS` characterize implementation-defined rounding behavior.

`sys.float_info.dig` 어트리뷰트는 추가 설명이 필요합니다. `s`가 최대 `sys.float_info.dig` 유효 숫자를 가진 십진수를 나타내는 문자열이면, `s`를 float로 변환한 후 다시 역변환하면 같은 십진수 값을 나타내는 문자열이 복구됩니다:

```
>>> import sys
>>> sys.float_info.dig
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
15
>>> s = '3.14159265358979'    # 15개의 유효 숫자가 있는 십진수 문자열
>>> format(float(s), '.15g')  # float로 변환 후 되돌립니다 -> 같은 값
'3.14159265358979'
```

그러나 유효 숫자가 *sys.float_info.dig*보다 많은 문자열의 경우, 항상 그렇지는 않습니다:

```
>>> s = '9876543211234567'    # 16개의 유효 숫자는 너무 많습니다!
>>> format(float(s), '.16g')  # 변환은 값을 바꿉니다
'9876543211234568'
```

sys.**float_repr_style**

repr() 함수가 float에 대해 작동하는 방식을 나타내는 문자열. 문자열이 `'short'` 값을 가지면, 유한 float `x`의 경우, `repr(x)` 는 `float(repr(x)) == x` 속성을 유지하는 짧은 문자열을 생성하는 것을 목표로 합니다. 이것은 파이썬 3.1 이상에서 일반적인 동작입니다. 그렇지 않으면, `float_repr_style` 은 `'legacy'` 값을 가지며 `repr(x)` 는 3.1 이전의 파이썬 버전에서와 같은 방식으로 작동합니다.

Added in version 3.1.

sys.**getallocatedblocks**()

크기에 상관없이, 인터프리터가 현재 할당한 메모리 블록 수를 반환합니다. 이 함수는 주로 메모리 누수를 추적하고 디버깅하는 데 유용합니다. 인터프리터의 내부 캐시로 인해, 결과는 호출마다 다를 수 있습니다; 보다 예측 가능한 결과를 얻으려면 *_clear_internal_caches()*와 *gc.collect()*를 호출해야 할 수도 있습니다.

파이썬 빌드나 구현이 이 정보를 합리적으로 계산할 수 없으면, *getallocatedblocks()*는 대신 0을 반환할 수 있습니다.

Added in version 3.4.

sys.**getunicodeinternedsize**()

Return the number of unicode objects that have been interned.

Added in version 3.12.

sys.**getandroidapilevel**()

Return the build-time API level of Android as an integer. This represents the minimum version of Android this build of Python can run on. For runtime version information, see *platform.android_ver()*.

가용성: Android.

Added in version 3.7.

sys.**getdefaultencoding**()

`'utf-8'`을 반환합니다. *str.encode()* 같은 메서드에서 사용되는, 기본 문자열 인코딩의 이름입니다.

sys.**getdlopenflags**()

`dlopen()` 호출에 사용되는 플래그의 현재 값을 반환합니다. 플래그 값의 기호 이름은 *os* 모듈에서 찾을 수 있습니다 (`RTLD_xxx` 상수, 예를 들어 *os.RTLD_LAZY*).

가용성: Unix.

sys.**getfilesystemencoding**()

Get the *filesystem encoding*: the encoding used with the *filesystem error handler* to convert between Unicode filenames and bytes filenames. The filesystem error handler is returned from *getfilesystemencodeerrors()*.

최상의 호환성을 위해, 파일명을 바이트열로 나타내는 것도 지원되지만, 모든 경우에 파일명에 str을 사용해야 합니다. 파일명을 받아들이거나 반환하는 함수는 str이나 bytes를 지원하고 내부적으로 시스템의 선호되는 표현으로 변환해야 합니다.

올바른 인코딩과 에러 모드를 사용하려면 *os.fsencode()*와 *os.fsdecode()*를 사용해야 합니다.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

버전 3.2에서 변경: `getfilesystemencoding()` 결과는 더는 `None`일 수 없습니다.

버전 3.6에서 변경: 윈도우는 더는 `'mbcs'`를 반환한다고 보장하지 않습니다. 자세한 정보는 **PEP 529**와 `_enablelegacywindowsfsencoding()`를 참조하십시오.

버전 3.7에서 변경: Return `'utf-8'` if the *Python UTF-8 Mode* is enabled.

`sys.`**`getfilesystemencodeerrors`**`()`

파일시스템 에러 처리기를 얻습니다: 유니코드 파일명과 바이트열 파일명 사이를 변환하는 파일시스템 인코딩과 함께 사용되는 에러 처리기. 파일시스템 인코딩은 `getfilesystemencoding()` 에서 반환됩니다.

올바른 인코딩과 에러 모드를 사용하려면 `os.fsencode()`와 `os.fsdecode()`를 사용해야 합니다.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

Added in version 3.6.

`sys.`**`get_int_max_str_digits`**`()`

Returns the current value for the *integer string conversion length limitation*. See also `set_int_max_str_digits()`.

Added in version 3.11.

`sys.`**`getrefcount`**`(`*object*`)`

*object*의 참조 횟수를 반환합니다. 반환된 수는 일반적으로 예상보다 1이 높습니다. `getrefcount()`에 대한 인자로서의 (임시) 참조를 포함하기 때문입니다.

Note that the returned value may not actually reflect how many references to the object are actually held. For example, some objects are *immortal* and have a very high refcount that does not reflect the actual number of references. Consequently, do not rely on the returned value to be accurate, other than a value of 0 or 1.

CPython 구현 상세: *Immortal* objects with a large reference count can be identified via `_is_immortal()`.

버전 3.12에서 변경: Immortal objects have very large refcounts that do not match the actual number of references to the object.

`sys.`**`getrecursionlimit`**`()`

파이썬 인터프리터 스택의 최대 깊이인, 재귀 한계의 현재 값을 반환합니다. 이 제한은 무한 재귀로 인해 C 스택의 오버플로가 발생하고 파이썬이 충돌하는 것을 방지합니다. `setrecursionlimit()`로 설정할 수 있습니다.

`sys.`**`getsizeof`**`(`*object*`[, `*default*`])`

객체의 크기를 바이트 단위로 반환합니다. 객체는 모든 형의 객체일 수 있습니다. 모든 내장 객체는 올바른 결과를 반환하지만, 구현 특정이기 때문에 제삼자 확장에서도 그렇다고 보장할 수는 없습니다.

객체에 직접 기여한 메모리 소비만 포함하며, 이 객체가 참조하는 객체의 메모리 소비는 따지지 않습니다.

주어진 경우, 객체가 크기를 조회하는 수단을 제공하지 않으면 *default*가 반환됩니다. 그렇지 않으면 `TypeError`가 발생합니다.

`getsizeof()`는 객체의 `__sizeof__` 메서드를 호출하고 객체가 가비지 수거기에 의해 관리되면 추가 가비지 수거기 오버헤드를 추가합니다.

컨테이너와 모든 내용물의 크기를 찾기 위해 `getsizeof()`를 재귀적으로 사용하는 예는 recursive sizeof recipe를 참조하십시오.

`sys.`**`getswitchinterval`**`()`

인터프리터의 "스레드 스위치 간격"을 초 단위로 반환합니다; `setswitchinterval()`을 참조하십시오.

Added in version 3.2.

sys.**_getframe**([*depth*])

호출 스택에서 프레임 객체를 반환합니다. 선택적 정수 *depth*가 제공되면, 스택 맨 위에서 지정한 수만큼 아래에 있는 호출의 프레임 객체를 반환합니다. 호출 스택보다 깊으면, *ValueError*가 발생합니다. *depth*의 기본값은 0이며, 호출 스택의 맨 위에 있는 프레임을 반환합니다.

인자 frame으로 감사 이벤트 sys._getframe을 발생시킵니다.

이 함수는 내부와 특수 목적으로만 사용해야 합니다. 모든 파이썬 구현에 존재한다고 보장되는 것은 아닙니다.

sys.**_getframemodulename**([*depth*])

호출 스택에서 모듈의 이름을 반환합니다. 선택적 정수 *depth*가 제공되면, 스택 맨 위에서 지정한 수만큼 아래에 있는 호출의 모듈을 반환합니다. 호출 스택보다 깊거나, 모듈을 식별할 수 없으면, None이 반환됩니다. *depth*의 기본값은 0이며, 호출 스택의 맨 위에 있는 모듈을 반환합니다.

인자 depth로 감사 이벤트 sys._getframemodulename을 발생시킵니다.

이 함수는 내부와 특수 목적으로만 사용해야 합니다. 모든 파이썬 구현에 존재한다고 보장되는 것은 아닙니다.

Added in version 3.12.

sys.**getobjects**(*limit*[, *type*])

This function only exists if CPython was built using the specialized configure option --with-trace-refs. It is intended only for debugging garbage-collection issues.

Return a list of up to *limit* dynamically allocated Python objects. If *type* is given, only objects of that exact type (not subtypes) are included.

Objects from the list are not safe to use. Specifically, the result will include objects from all interpreters that share their object allocator state (that is, ones created with PyInterpreterConfig.use_main_obmalloc set to 1 or using Py_NewInterpreter(), and the main interpreter). Mixing objects from different interpreters may lead to crashes or other unexpected behavior.

이 함수는 특수 목적으로만 사용해야 합니다. 모든 파이썬 구현에 존재한다고 보장되는 것은 아닙니다.

버전 3.14에서 변경: The result may include objects from other interpreters.

sys.**getprofile**()

*setprofile()*에 의해 설정된 프로파일러 함수를 얻습니다.

sys.**gettrace**()

*settrace()*에 의해 설정된 추적 함수를 얻습니다.

gettrace() 함수는 디버거, 프로파일러, 커버리지 도구 등을 구현하기 위한 것입니다. 그 동작은 언어 정의의 일부라기보다는 구현 플랫폼의 일부이기 때문에, 모든 파이썬 구현에서 사용 가능한 것은 아닙니다.

sys.**getwindowsversion**()

현재 실행 중인 윈도우 버전을 설명하는 네임드 튜플을 반환합니다. 명명된 요소는 *major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, *product_type* 및 *platform_version*입니다. *service_pack*은 문자열을 포함하고 *platform_version*은 3-튜플이며 다른 모든 값은 정수입니다. 구성 요소는 이름으로도 액세스할 수 있어서, sys.getwindowsversion()[0] 은 sys.getwindowsversion().major와 동등합니다. 이전 버전과의 호환성을 위해, 처음 5개 요소 만 인덱싱을 통해 꺼낼 수 있습니다.

*platform*은 2 (VER_PLATFORM_WIN32_NT) 입니다.

*product_type*은 다음 값 중 하나일 수 있습니다:

상수	의미
1 (VER_NT_WORKSTATION)	시스템은 워크 스테이션입니다.
2 (VER_NT_DOMAIN_CONTROLLER)	시스템은 도메인 컨트롤러입니다.
3 (VER_NT_SERVER)	시스템은 서버이지만, 도메인 컨트롤러는 아닙니다.

이 함수는 Win32 `GetVersionEx()` 함수를 감쌉니다; 이러한 필드에 대한 자세한 내용은 `OSVERSIONINFOEX()` 에 대한 마이크로소프트 설명서를 참조하십시오.

platform_version 은 프로세스에 대해 에뮬레이션 되는 버전이 아니라, 주 버전, 부 버전 및 현재 운영 체제의 빌드 번호를 반환합니다. 기능 감지가 아닌 로깅에 사용하기 위한 것입니다.

> **참고**
>
> *platform_version* derives the version from kernel32.dll which can be of a different version than the OS version. Please use `platform` module for achieving accurate OS version.

가용성: Windows.

버전 3.2에서 변경: 네임드 튜플로 변경되어 *service_pack_minor*, *service_pack_major*, *suite_mask* 및 *product_type* 이 추가되었습니다.

버전 3.6에서 변경: *platform_version* 을 추가했습니다

`sys.`**`get_asyncgen_hooks`**`()`

`(firstiter, finalizer)` 형식의 `namedtuple`과 유사한 *asyncgen_hooks* 객체를 반환합니다. 여기서 *firstiter* 와 *finalizer* 는 `None` 이나 비동기 제너레이터 이터레이터를 인자로 취하는 함수로 기대되며, 이벤트 루프에 의해 비동기 제너레이터의 파이널리제이션을 스케줄 하는 데 사용됩니다.

Added in version 3.6: 자세한 내용은 **PEP 525**를 참조하십시오.

> **참고**
>
> 이 함수는 잠정적(provisional)으로 추가되었습니다 (자세한 내용은 **PEP 411**을 참조하십시오).

`sys.`**`get_coroutine_origin_tracking_depth`**`()`

`set_coroutine_origin_tracking_depth()` 로 설정된 현재 코루틴 원점 추적 깊이를 가져옵니다.

Added in version 3.7.

> **참고**
>
> 이 함수는 잠정적(provisional)으로 추가되었습니다 (자세한 내용은 **PEP 411**을 참조하십시오). 디버깅 목적으로만 사용하십시오.

`sys.`**`hash_info`**

숫자 해시 구현의 매개 변수를 제공하는 네임드 튜플. 숫자 형의 해싱에 대한 자세한 내용은 숫자 형의 해싱을 참조하십시오.

`hash_info.`**`width`**

해시값에 사용되는 비트 폭

`hash_info.`**`modulus`**

수치 해시 체계에 사용되는 소수 모듈러스 P

`hash_info.`**`inf`**

양의 무한대에 대해 반환된 해시값

hash_info.**nan**

(This attribute is no longer used)

hash_info.**imag**

복소수의 허수부에 사용되는 승수(multiplier)

hash_info.**algorithm**

str, bytes 및 memoryview의 해싱 알고리즘 이름

hash_info.**hash_bits**

해시 알고리즘의 내부 출력 크기

hash_info.**seed_bits**

해시 알고리즘의 시드 키 크기

Added in version 3.2.

버전 3.4에서 변경: *algorithm*, *hash_bits* 및 *seed_bits*를 추가했습니다

sys.**hexversion**

단일 정수로 인코딩된 버전 번호. 이것은 비 프로덕션 릴리스에 대한 적절한 지원을 포함하여, 버전마다 증가함이 보장됩니다. 예를 들어, 파이썬 인터프리터가 버전 1.5.2 이상인지 검사하려면, 다음을 사용하십시오:

```
if sys.hexversion >= 0x010502F0:
    # 고급 기능을 사용합니다
    ...
else:
    # 대안 구현을 사용하거나 사용자에게 경고합니다
    ...
```

내장 *hex()* 함수에 전달한 결과로 볼 때만 실제로 의미가 있기 때문에 이것을 hexversion이라고 합니다. 네임드 튜플 *sys.version_info*는 같은 정보의 더 인간 친화적인 인코딩으로 사용될 수 있습니다.

hexversion에 대한 자세한 내용은 apiabiversion 에서 찾을 수 있습니다.

sys.**implementation**

현재 실행 중인 파이썬 인터프리터의 구현에 대한 정보가 포함된 객체. 모든 파이썬 구현에서 다음과 같은 어트리뷰트가 있어야 합니다.

*name*은 구현 식별자입니다, 예를 들어 'cpython'. 실제 문자열은 파이썬 구현에 의해 정의되지만, 소문자임이 보장됩니다.

*version*은 *sys.version_info*와 같은 형식의 네임드 튜플입니다. 파이썬 구현의 버전을 나타냅니다. 이것은 현재 실행 중인 인터프리터가 준수하는, sys.version_info가 나타내는, 특정 버전의 파이썬 언어와는 다른 의미입니다. 예를 들어, PyPy 1.8의 경우 sys.implementation.version은 sys.version_info(1, 8, 0, 'final', 0)일 수 있지만, sys.version_info는 sys.version_info(2, 7, 2, 'final', 0)입니다. CPython의 경우 참조 구현이기 때문에 같은 값입니다.

*hexversion*은 *sys.hexversion*과 같은 16진수 형식의 구현 버전입니다.

*cache_tag*는 임포트 절차에서 캐시 된 모듈의 파일명에 사용되는 태그입니다. 관습상, 'cpython-33'과 같이 구현 이름과 버전을 합성한 것입니다. 그러나, 파이썬 구현은 적절하다면 다른 값을 사용할 수 있습니다. cache_tag가 None으로 설정되면, 모듈 캐싱을 사용하지 않아야 함을 나타냅니다.

supports_isolated_interpreters is a boolean value, whether this implementation supports multiple isolated interpreters. It is True for CPython on most platforms. Platforms with this support implement the low-level _interpreters module.

> **더 보기**
>
> **PEP 684**, **PEP 734**, and *concurrent.interpreters*.

*sys.implementation*은 파이썬 구현에 고유한 추가 어트리뷰트를 포함할 수 있습니다. 이러한 비표준 어트리뷰트는 밑줄로 시작해야 하며, 여기에서는 설명하지 않습니다. 내용과 관계없이, *sys.implementation*은 인터프리터 실행 중이나 구현 버전 간에 변경되지 않습니다. (그러나, 파이썬 언어 버전 간에는 변경될 수 있습니다.) 자세한 정보는 **PEP 421**을 참조하십시오.

Added in version 3.3.

버전 3.14에서 변경: Added `supports_isolated_interpreters` field.

> **참고**
>
> 새로운 필수 어트리뷰트를 추가하려면 일반 PEP 프로세스를 거쳐야 합니다. 자세한 정보는 **PEP 421**을 참조하십시오.

`sys.`**`int_info`**

파이썬의 정수 내부 표현에 대한 정보를 담고 있는 네임드 튜플. 어트리뷰트는 읽기 전용입니다.

`int_info.`**`bits_per_digit`**

각 자릿수에 담긴 비트 수. 파이썬 정수는 내부적으로 기수(base) `2**int_info.bits_per_digit`로 저장됩니다.

`int_info.`**`sizeof_digit`**

자릿수를 나타내는 데 사용되는 C형의 바이트 단위 크기.

`int_info.`**`default_max_str_digits`**

The default value for *sys.get_int_max_str_digits()* when it is not otherwise explicitly configured.

`int_info.`**`str_digits_check_threshold`**

The minimum non-zero value for *sys.set_int_max_str_digits()*, `PYTHONINTMAXSTRDIGITS`, or `-X int_max_str_digits`.

Added in version 3.1.

버전 3.11에서 변경: Added *default_max_str_digits* and *str_digits_check_threshold*.

`sys.`**`__interactivehook__`**

이 어트리뷰트가 존재하면, 대화형 모드로 인터프리터가 시작될 때 해당 값이 (인자 없이) 자동으로 호출됩니다. 이것은 `PYTHONSTARTUP` 파일을 읽은 후에 수행되므로, 그곳에서 이 훅을 설정할 수 있습니다. *site* 모듈은 이것을 설정합니다.

시작 시 훅이 호출될 때 훅 객체를 인자로 감사 이벤트 `cpython.run_interactivehook`을 발생시킵니다.

Added in version 3.4.

`sys.`**`intern`**(*string*)

"인턴 된(interned)" 문자열 테이블에 *string*을 넣고, *string* 자신이거나 사본인 인턴 된 문자열을 반환합니다. 문자열을 인턴 하는 것은 딕셔너리 조회에서 약간의 성능 개선을 얻는 데 유용합니다 – 딕셔너리의 키가 인턴 되고. 조회 키가 인턴 되면, (해싱 후의) 키 비교는 문자열 비교 대신 포인터 비교를 수행 할 수 있습니다. 일반적으로, 파이썬 프로그램에서 사용되는 이름은 자동으로 인턴 되며, 모듈, 클래스 또는 인스턴스 어트리뷰트를 담는 데 사용되는 딕셔너리는 인턴 된 키를 갖습니다.

인턴 된 문자열은 불멸이 아닙니다; 이점을 얻으려면 *intern()*의 반환 값에 대한 참조를 유지해야 합니다.

sys.**_is_gil_enabled**()

Return *True* if the *GIL* is enabled and *False* if it is disabled.

Added in version 3.13.

모든 파이썬 구현에 존재한다고 보장되는 것은 아닙니다.

sys.**is_finalizing**()

메인 파이썬 인터프리터가 종료 중이면 *True*를 반환합니다. 그렇지 않으면 *False*를 반환합니다.

See also the *PythonFinalizationError* exception.

Added in version 3.5.

sys.**_jit**

Utilities for observing just-in-time compilation.

CPython 구현 상세: JIT compilation is an *experimental implementation detail* of CPython. `sys._jit` is not guaranteed to exist or behave the same way in all Python implementations, versions, or build configurations.

Added in version 3.14.

_jit.**is_available**()

Return `True` if the current Python executable supports JIT compilation, and `False` otherwise. This can be controlled by building CPython with the `--experimental-jit` option on Windows, and the `--enable-experimental-jit` option on all other platforms.

_jit.**is_enabled**()

Return `True` if JIT compilation is enabled for the current Python process (implies *sys._jit.is_available()*), and `False` otherwise. If JIT compilation is available, this can be controlled by setting the `PYTHON_JIT` environment variable to `0` (disabled) or `1` (enabled) at interpreter startup.

_jit.**is_active**()

Return `True` if the topmost Python frame is currently executing JIT code (implies *sys._jit.is_enabled()*), and `False` otherwise.

> **참고**
>
> This function is intended for testing and debugging the JIT itself. It should be avoided for any other purpose.

> **참고**
>
> Due to the nature of tracing JIT compilers, repeated calls to this function may give surprising results. For example, branching on its return value will likely lead to unexpected behavior (if doing so causes JIT code to be entered or exited):

```
>>> for warmup in range(BIG_NUMBER):
...     # This line is "hot", and is eventually JIT-compiled:
...     if sys._jit.is_active():
...         # This line is "cold", and is run in the interpreter:
...         assert sys._jit.is_active()
...
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
    assert sys._jit.is_active()
           ~~~~~~~~~~~~~~~~~~^^
AssertionError
```

sys.**last_exc**

이 변수가 항상 정의되는 것은 아닙니다; 예외가 처리되지 않고 인터프리터가 에러 메시지와 스택 트레이스백을 인쇄할 때 예외 인스턴스로 설정됩니다. 의도된 용도는 대화형 사용자가 디버거 모듈을 임포트하고 에러를 일으킨 명령을 다시 실행하지 않고도 사후 디버깅에 참여할 수 있도록 하는 것입니다. (사후 디버거에 들어가기 위해 일반적으로 `import pdb; pdb.pm()`을 사용합니다; 자세한 내용은 *pdb* 모듈을 참조하십시오.)

Added in version 3.12.

sys.**_is_immortal**(*op*)

Return `True` if the given object is *immortal*, `False` otherwise.

> **참고**
>
> Objects that are immortal (and thus return `True` upon being passed to this function) are not guaranteed to be immortal in future versions, and vice versa for mortal objects.

Added in version 3.14.

이 함수는 특수 목적으로만 사용해야 합니다. 모든 파이썬 구현에 존재한다고 보장되는 것은 아닙니다.

sys.**_is_interned**(*string*)

주어진 문자열이 "인턴" 되었으면 `True`를, 그렇지 않으면 `False`를 반환합니다.

Added in version 3.13.

모든 파이썬 구현에 존재한다고 보장되는 것은 아닙니다.

sys.**last_type**

sys.**last_value**

sys.**last_traceback**

These three variables are deprecated; use `sys.last_exc` instead. They hold the legacy representation of `sys.last_exc`, as returned from `exc_info()` above.

sys.**maxsize**

`Py_ssize_t` 형의 변수가 취할 수 있는 최댓값을 제공하는 정수. 일반적으로 32비트 플랫폼에서는 `2**31 - 1`이고 64비트 플랫폼에서는 `2**63 - 1`입니다.

sys.**maxunicode**

가장 큰 유니코드 코드 포인트의 값을 제공하는 정수, 즉 `1114111` (16진수로 `0x10FFFF`).

버전 3.3에서 변경: **PEP 393** 이전에는, 유니코드 문자가 UCS-2와 UCS-4 중 어는 것으로 저장되었는지를 지정하는 구성 옵션에 따라, `sys.maxunicode`는 `0xFFFF`나 `0x10FFFF`이었습니다.

sys.**meta_path**

메타 경로 파인더 객체의 리스트. 이 객체의 `find_spec()` 메서드를 호출해서 임포트 할 모듈을 찾을 수 있는지 확인할 수 있습니다. 기본적으로, 파이썬의 기본 임포트 의미 체계를 구현하는 항목들을 포함합니다. 최소한 임포트 할 모듈의 절대 이름으로 `find_spec()` 메서드가 호출됩니다. 임포트 할 모듈이 패키지에 포함되어 있으면, 부모 패키지의 `__path__` 어트리뷰트가 두 번째 인자로 전달됩니다. 이 메서드는 모듈 스펙이나 모듈을 찾을 수 없으면 `None`을 반환합니다.

> **더 보기**
>
> ***importlib.abc.MetaPathFinder***
> `meta_path`에 있는 파인더 객체의 인터페이스를 정의하는 추상 베이스 클래스.
>
> ***importlib.machinery.ModuleSpec***
> `find_spec()`이 이 구상 클래스의 인스턴스를 반환해야 합니다.

버전 3.4에서 변경: 모듈 스펙은 파이썬 3.4에서 **PEP 451**에 의해 도입되었습니다.

버전 3.12에서 변경: *meta_path* 항목에 *find_spec()* 메서드가 없으면 `find_module()` 메서드를 찾는 폴백을 제거했습니다.

`sys.`**`modules`**

이것은 모듈 이름을 이미 로드된 모듈로 매핑하는 딕셔너리입니다. 모듈의 재로딩과 기타 트릭을 강제하기 위해 조작할 수 있습니다. 그러나 딕셔너리를 교체하는 것은 예상대로 작동하지는 않으며 딕셔너리에서 필수 항목을 삭제하면 파이썬이 실패할 수 있습니다. 이 전역 딕셔너리를 이터레이트하려면 항상 `sys.modules.copy()` 또는 `tuple(sys.modules)` 를 사용하여, 다른 스레드의 코드나 활동의 부작용으로 인해 이터레이션 중 딕셔너리 크기가 변경되어 발생하는 예외를 피하십시오.

`sys.`**`orig_argv`**

The list of the original command line arguments passed to the Python executable.

The elements of *sys.orig_argv* are the arguments to the Python interpreter, while the elements of *sys.argv* are the arguments to the user' s program. Arguments consumed by the interpreter itself will be present in *sys.orig_argv* and missing from *sys.argv*.

Added in version 3.10.

`sys.`**`path`**

모듈의 검색 경로를 지정하는 문자열 리스트. 환경 변수 `PYTHONPATH`와 설치 종속 기본값으로 초기화되었습니다.

By default, as initialized upon program startup, a potentially unsafe path is prepended to *sys.path* (*before* the entries inserted as a result of `PYTHONPATH`):

- `python -m module` 명령 줄: 현재 작업 디렉터리를 앞에 붙입니다.
- `python script.py` command line: prepend the script' s directory. If it' s a symbolic link, resolve symbolic links.
- `python -c code` and `python` (REPL) command lines: prepend an empty string, which means the current working directory.

To not prepend this potentially unsafe path, use the `-P` command line option or the `PYTHONSAFEPATH` environment variable.

프로그램은 자체 목적으로 이 리스트를 자유롭게 수정할 수 있습니다. 문자열만 *sys.path*에 추가해야 합니다; 임포트 하는 동안 다른 모든 데이터형은 무시됩니다.

더 보기

- 모듈 *site*. 이것은 .pth 파일을 사용하여 *sys.path*를 확장하는 방법에 관해 설명합니다.

`sys.`**`path_hooks`**

경로 인자를 취해서 경로를 위한 파인더를 만들려고 시도하는 콜러블의 리스트. 파인더를 만들 수 있으면, 콜러블이 반환하고, 그렇지 않으면 *ImportError*를 발생시킵니다.

원래 **PEP 302**에서 지정되었습니다.

`sys.`**`path_importer_cache`**

파인더 객체의 캐시 역할을 하는 딕셔너리. 키는 *sys.path_hooks*에 전달된 경로이며 값은 찾은 파인더입니다. 경로가 유효한 파일 시스템 경로이지만 *sys.path_hooks*에 파인더가 없으면 `None`이 저장됩니다.

원래 **PEP 302**에서 지정되었습니다.

`sys.`**`platform`**

A string containing a platform identifier. Known values are:

시스템	platform 값
AIX	'aix'
안드로이드	'android'
Emscripten	'emscripten'
FreeBSD	'freebsd'
iOS	'ios'
리눅스	'linux'
맥 OS	'darwin'
윈도우	'win32'
윈도우/Cygwin	'cygwin'
WASI	'wasi'

On Unix systems not listed in the table, the value is the lowercased OS name as returned by `uname -s`, with the first part of the version as returned by `uname -r` appended, e.g. `'sunos5'`, *at the time when Python was built*. Unless you want to test for a specific system version, it is therefore recommended to use the following idiom:

```
if sys.platform.startswith('sunos'):
    # SunOS-specific code here...
```

버전 3.3에서 변경: 리눅스에서, *sys.platform*은 더는 주 버전을 포함하지 않습니다. `'linux2'`나 `'linux3'` 대신, 항상 `'linux'`입니다.

버전 3.8에서 변경: AIX에서, *sys.platform*은 더는 주 버전을 포함하지 않습니다. `'aix5'`나 `'aix7'` 대신, 항상 `'aix'`입니다.

버전 3.13에서 변경: On Android, *sys.platform* now returns `'android'` rather than `'linux'`.

버전 3.14에서 변경: On FreeBSD, *sys.platform* doesn' t contain the major version anymore. It is always `'freebsd'`, instead of `'freebsd13'` or `'freebsd14'`.

더 보기

*os.name*은 덜 세분되어 있습니다. *os.uname()*은 시스템 종속 버전 정보를 제공합니다.

platform 모듈은 시스템 식별자에 대한 상세한 검사를 제공합니다.

`sys.`**`platlibdir`**

플랫폼별 라이브러리 디렉터리의 이름. 표준 라이브러리의 경로와 설치된 확장 모듈들의 경로를 빌드하는 데 사용됩니다.

대부분의 플랫폼에서 `"lib"`와 같습니다. Fedora와 SuSE에서는, 64비트 플랫폼에서 `"lib64"`와 같으며 다음과 같은 `sys.path` 경로를 제공합니다 (`X.Y`는 파이썬 `major.minor` 버전):

- `/usr/lib64/pythonX.Y/`: 표준 라이브러리 (*os* 모듈의 `os.py`와 같은)
- `/usr/lib64/pythonX.Y/lib-dynload/`: 표준 라이브러리의 C 확장 모듈 (*errno* 모듈과 같은, 정확한 파일 이름은 플랫폼에 따라 다릅니다)
- `/usr/lib/pythonX.Y/site-packages/` (항상 `lib`를 사용합니다, *sys.platlibdir*이 아닙니다): 제삼자 모듈
- `/usr/lib64/pythonX.Y/site-packages/`: 제삼자 패키지의 C 확장 모듈

Added in version 3.9.

`sys.`**`prefix`**

플랫폼 독립적인 파이썬 파일이 설치되는 사이트별 디렉터리 접두사를 제공하는 문자열; 유닉스에서, 기본 값은 `/usr/local`입니다. **`configure`** 스크립트에 대한 `--prefix` 인자를 사용하여 빌드 시 설정할 수 있습니다. 파생 경로는 설치 경로를 참조하십시오.

> **참고**
>
> If a *virtual environment* is in effect, this `prefix` will point to the virtual environment. The value for the Python installation will still be available, via `base_prefix`. Refer to *Virtual Environments* for more information.

버전 3.14에서 변경: When running under a *virtual environment*, `prefix` and `exec_prefix` are now set to the virtual environment prefix by the *path initialization*, instead of `site`. This means that `prefix` and `exec_prefix` always point to the virtual environment, even when `site` is disabled (`-S`).

`sys.`**`ps1`**

`sys.`**`ps2`**

인터프리터의 기본과 보조 프롬프트를 지정하는 문자열. 인터프리터가 대화형 모드일 때만 정의됩니다. 이 경우 초깃값은 `'>>> '`과 `'... '`입니다. 문자열이 아닌 객체가 어느 변수에라도 지정되면, 인터프리터가 새 대화식 명령을 읽을 준비를 할 때마다 그 객체의 `str()`이 재평가됩니다; 동적 프롬프트를 구현하는 데 사용할 수 있습니다.

`sys.`**`setdlopenflags`**(*n*)

인터프리터가 확장 모듈을 로드할 때와 같이 `dlopen()` 호출에 인터프리터가 사용하는 플래그를 설정합니다. 무엇보다도, 이것은 `sys.setdlopenflags(0)`라고 호출하는 경우, 모듈을 임포트 할 때 심볼의 지연된 결정(lazy resolving)을 활성화합니다. 확장 모듈 간에 심볼을 공유하려면, `sys.setdlopenflags(os.RTLD_GLOBAL)`로 호출하십시오. 플래그 값의 기호 이름은 `os` 모듈에서 찾을 수 있습니다 (`RTLD_xxx` 상수, 예를 들어 `os.RTLD_LAZY`).

가용성: Unix.

`sys.`**`set_int_max_str_digits`**(*maxdigits*)

Set the *integer string conversion length limitation* used by this interpreter. See also `get_int_max_str_digits()`.

Added in version 3.11.

`sys.`**`setprofile`**(*profilefunc*)

시스템의 프로파일 함수를 설정합니다. 파이썬에서 파이썬 소스 코드 프로파일러를 구현할 수 있도록 합니다. 파이썬 프로파일러에 대한 자세한 내용은 파이썬 프로파일러 장을 참조하십시오. 시스템의 프로파일 함수는 시스템의 추적 함수(`settrace()`를 참조하십시오)와 유사하게 호출되지만, 다른 이벤트로 호출됩니다, 예를 들어 이 함수는 실행되는 줄마다 호출되지 않습니다 (오직 호출과 반환에서만 호출됩니다만, 예외가 설정되었을 때도 반환 이벤트가 보고됩니다). 이 함수는 스레드로 한정되지만, 프로파일러가 스레드 간의 컨텍스트 전환에 대해 알 방법이 없어서, 여러 스레드가 있을 때 이를 사용하는 것은 의미가 없습니다. 또한, 반환 값이 사용되지 않아서, 단순히 `None`을 반환할 수 있습니다. 프로파일 함수에서 에러가 발생하면 설정이 해제됩니다.

> **참고**
>
> The same tracing mechanism is used for `setprofile()` as `settrace()`. To trace calls with `setprofile()` inside a tracing function (e.g. in a debugger breakpoint), see `call_tracing()`.

프로파일 함수에는 세 가지 인자가 있습니다: *frame*, *event* 및 *arg*. *frame*은 현재 스택 프레임입니다. *event*는 문자열입니다: `'call'`, `'return'`, `'c_call'`, `'c_return'` 또는 `'c_exception'`. *arg*는 이벤트 유형에 따라 다릅니다.

이벤트의 의미는 다음과 같습니다:

`'call'`
함수가 호출되었습니다 (또는 다른 코드 블록에 진입했습니다). 프로파일 함수가 호출됩니다; *arg*는 `None`입니다.

`'return'`
함수(또는 다른 코드 블록)가 반환하려고 합니다. 프로파일 함수가 호출됩니다; *arg*는 반환될 값이거나, 예외가 발생하여 이벤트가 발생한 경우는 `None`입니다.

'c_call'
C 함수를 호출하려고 합니다. 확장 함수나 내장일 수 있습니다. *arg*는 C 함수 객체입니다.

'c_return'
C 함수가 반환했습니다. *arg*는 C 함수 객체입니다.

'c_exception'
C 함수에서 예외가 발생했습니다. *arg*는 C 함수 객체입니다.

인자 없이 감사 이벤트 `sys.setprofile`을 발생시킵니다.

`sys.`**`setrecursionlimit`**(*limit*)
파이썬 인터프리터 스택의 최대 깊이를 *limit*로 설정합니다. 이 제한은 무한 재귀로 인해 C 스택의 오버플로가 발생하고 파이썬이 충돌하는 것을 방지합니다.

가능한 최대 제한은 플랫폼에 따라 다릅니다. 사용자는 깊은 재귀가 필요한 프로그램과 더 높은 제한을 지원하는 플랫폼이 있을 때 제한을 더 높게 설정해야 할 수 있습니다. 제한이 너무 높으면 충돌이 발생할 수 있기 때문에, 주의해서 사용해야 합니다.

현재 재귀 깊이에서 새 제한이 너무 낮으면 `RecursionError` 예외가 발생합니다.

버전 3.5.1에서 변경: 현재 재귀 깊이에서 새 한계가 너무 낮으면 이제 `RecursionError` 예외가 발생합니다.

`sys.`**`setswitchinterval`**(*interval*)
인터프리터의 스레드 전환 간격을 (초 단위로) 설정합니다. 이 부동 소수점 값은 동시에 실행 중인 파이썬 스레드에 할당된 "시 분할(timeslices)"의 이상적인 지속 시간을 결정합니다. 특히 오래 실행되는 내부 함수나 메서드가 사용된다면, 실제 값은 더 클 수 있음에 유의하십시오. 또한, 간격이 끝날 때 어떤 스레드가 예약되는지는 운영 체제의 결정입니다. 인터프리터에는 자체 스케줄러가 없습니다.

Added in version 3.2.

`sys.`**`settrace`**(*tracefunc*)
시스템의 추적 함수를 설정합니다. 파이썬에서 파이썬 소스 코드 디버거를 구현할 수 있도록 합니다. 이 함수는 스레드로 한정됩니다; 디버거가 여러 스레드를 지원하려면, 디버깅 중인 각 스레드에 대해 `settrace()`를 사용하여 추적 함수를 등록하거나, `threading.settrace()`를 사용해야 합니다.

Trace functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'line'`, `'return'`, `'exception'` or `'opcode'`. *arg* depends on the event type.

추적 함수는 새로운 로컬 스코프에 진입할 때마다 호출됩니다 (*event*가 `'call'`로 설정됩니다); 새 스코프에서 사용될 지역 추적 함수(local trace function)에 대한 참조를 반환하거나, 스코프를 추적하지 않아야 하면 `None`을 반환해야 합니다.

지역 추적 함수는 자기 자신, 또는 해당 스코프의 지역 추적 함수로 사용될 다른 함수에 대한 참조를 반환해야 합니다.

추적 함수에서 에러가 발생하면, `settrace(None)` 이 호출되는 것처럼 설정이 해제됩니다.

> **참고**
>
> Tracing is disabled while calling the trace function (e.g. a function set by `settrace()`). For recursive tracing see `call_tracing()`.

이벤트의 의미는 다음과 같습니다:

'call'
함수가 호출되었습니다 (또는 다른 코드 블록에 진입했습니다). 전역 추적 함수가 호출됩니다; *arg*는 `None` 입니다; 반환 값은 지역 추적 함수를 지정합니다.

'line'
인터프리터가 새로운 코드 줄을 실행하거나 루프의 조건을 다시 실행하려고 합니다. 지역 추적 함수가 호출됩니다; *arg*는 `None` 입니다; 반환 값은 새로운 지역 추적 함수를 지정합니다. 작동 방식에 대한 자세한 설명은 `Objects/lnotab_notes.txt`를 참조하십시오. 해당 프레임에서 `f_trace_lines`를 `False`로 설정하여 줄별 이벤트를 비활성화 할 수 있습니다.

'return'
함수(또는 다른 코드 블록)가 반환하려고 합니다. 지역 추적 함수가 호출됩니다; *arg*는 반환될 값이거나, 예외가 발생하여 이벤트가 발생한 경우는 `None`입니다. 추적 함수의 반환 값은 무시됩니다.

'exception'
예외가 발생했습니다. 지역 추적 함수가 호출됩니다; *arg*는 튜플 `(exception, value, traceback)`입니다; 반환 값은 새로운 지역 추적 함수를 지정합니다.

'opcode'
인터프리터가 새 옵코드(opcode)를 실행하려고 합니다 (opcode 세부 사항은 `dis`를 참조하십시오). 지역 추적 함수가 호출됩니다; *arg*는 `None`입니다; 반환 값은 새로운 지역 추적 함수를 지정합니다. 옵코드 별 이벤트는 기본적으로 발생하지 않습니다; 해당 프레임에서 `f_trace_opcodes`를 `True`로 설정하여 명시적으로 요청해야 합니다.

호출자 체인을 따라 예외가 전파됨에 따라, 각 수준에서 `'exception'` 이벤트가 생성됨에 유의하십시오.

더 세분된 사용을 위해, 이미 설치된 추적 함수의 반환 값을 통해 간접적으로 설정되는 것에 의존하는 대신, `frame.f_trace = tracefunc`를 명시적으로 대입하여 추적 함수를 설정할 수 있습니다. 이것은 현재 프레임에서 추적 함수를 활성화하는 데에도 필요한데, `settrace()`가 하지 않는 일입니다. 이것이 작동하려면 실행 시간 추적 장치를 활성화하기 위해 전역 추적 함수가 `settrace()`로 설치되어 있어야 하지만, 같은 추적 함수 일 필요는 없음에 유의하십시오 (예를 들어, 각 프레임에서 즉시 비활성화되도록 단순히 `None`을 반환하는 오버헤드가 낮은 추적 함수일 수 있습니다).

코드와 프레임 객체에 대한 자세한 내용은 types를 참조하십시오.

인자 없이 감사 이벤트 `sys.settrace`를 발생시킵니다.

`settrace()` 함수는 오직 디버거, 프로파일러, 커버리지(coverage) 도구 등을 구현하기 위한 것입니다. 그 동작은 언어 정의의 일부라기보다는 구현 플랫폼의 일부라서, 모든 파이썬 구현에서 사용 가능한 것은 아닙니다.

버전 3.7에서 변경: `'opcode'` 이벤트 유형이 추가되었습니다; `f_trace_lines`와 `f_trace_opcodes` 어트리뷰트가 프레임에 추가되었습니다

`sys.`**`set_asyncgen_hooks`**(*[firstiter] [, finalizer]*)
두 개의 선택적 키워드 인자를 받아들이는데, 모두 비동기 제너레이터 이터레이터를 인자로 받아들이는 콜러블입니다. 비동기 제너레이터가 처음으로 이터레이트 될 때 *firstiter* 콜러블이 호출됩니다. 비동기 제너레이터가 가비지 수거될 때 *finalizer*가 호출됩니다.

인자 없이 감사 이벤트 `sys.set_asyncgen_hooks_firstiter`를 발생시킵니다.

인자 없이 감사 이벤트 `sys.set_asyncgen_hooks_finalizer`를 발생시킵니다.

하부 API는 두 개의 호출로 구성되기 때문에, 두 개의 감사 이벤트가 발생합니다, 각각은 자체 이벤트를 발생시켜야 합니다.

Added in version 3.6: 자세한 내용은 **PEP 525**를 참조하고, *finalizer* 메서드의 참조 예제는 Lib/asyncio/base_events.py의 `asyncio.Loop.shutdown_asyncgens` 구현을 참조하십시오.

> **참고**
> 이 함수는 잠정적(provisional)으로 추가되었습니다 (자세한 내용은 **PEP 411**을 참조하십시오).

`sys.`**`set_coroutine_origin_tracking_depth`**(*depth*)
코루틴 원점 추적을 활성화하거나 비활성화하도록 합니다. 활성화하면, 코루틴 객체의 `cr_origin` 어트리뷰트에 코루틴 객체가 만들어진 트레이스백을 설명하는 (파일명, 줄 번호, 함수 이름) 튜플이 포함됩니다. 가장 최근의 호출이 먼저 옵니다. 비활성화하면 `cr_origin`은 `None`입니다.

To enable, pass a *depth* value greater than zero; this sets the number of frames whose information will be captured. To disable, set *depth* to zero.

이 설정은 스레드에 한정됩니다.

Added in version 3.7.

> **참고**
>
> 이 함수는 잠정적(provisional)으로 추가되었습니다 (자세한 내용은 **PEP 411**을 참조하십시오). 디버깅 목적으로만 사용하십시오.

`sys.activate_stack_trampoline`(*backend, /*)

Activate the stack profiler trampoline *backend*. The only supported backend is `"perf"`.

Stack trampolines cannot be activated if the JIT is active.

가용성: Linux.

Added in version 3.12.

> **더 보기**
>
> - perf_profiling
> - https://perf.wiki.kernel.org

`sys.deactivate_stack_trampoline`()

Deactivate the current stack profiler trampoline backend.

If no stack profiler is activated, this function has no effect.

가용성: Linux.

Added in version 3.12.

`sys.is_stack_trampoline_active`()

Return `True` if a stack profiler trampoline is active.

가용성: Linux.

Added in version 3.12.

`sys.remote_exec`(*pid, script*)

Executes *script*, a file containing Python code in the remote process with the given *pid*.

This function returns immediately, and the code will be executed by the target process' s main thread at the next available opportunity, similarly to how signals are handled. There is no interface to determine when the code has been executed. The caller is responsible for making sure that the file still exists whenever the remote process tries to read it and that it hasn' t been overwritten.

The remote process must be running a CPython interpreter of the same major and minor version as the local process. If either the local or remote interpreter is pre-release (alpha, beta, or release candidate) then the local and remote interpreters must be the same exact version.

When the code is executed in the remote process, an *auditing event* `sys.remote_exec` is raised with the *pid* and the path to the script file. This event is raised in the process that called `sys.remote_exec()`.

When the script is executed in the remote process, an *auditing event* `cpython.remote_debugger_script` is raised with the path in the remote process. This event is raised in the remote process, not the one that called `sys.remote_exec()`.

가용성: Unix, Windows.

Added in version 3.14.

sys.**_enablelegacywindowsfsencoding**()

3.6 이전의 파이썬 버전과 일관성을 유지하기 위해, 파일 시스템 인코딩과 에러 처리기를 각각 'mbcs' 와 'replace' 로 변경합니다.

이것은 파이썬을 시작하기 전에 PYTHONLEGACYWINDOWSFSENCODING 환경 변수를 정의하는 것과 동등합니다.

See also *sys.getfilesystemencoding()* and *sys.getfilesystemencodeerrors()*.

가용성: Windows.

> **참고**
>
> Changing the filesystem encoding after Python startup is risky because the old fsencoding or paths encoded by the old fsencoding may be cached somewhere. Use PYTHONLEGACYWINDOWSFSENCODING instead.

Added in version 3.6: 자세한 내용은 **PEP 529**를 참조하십시오.

Deprecated since version 3.13, will be removed in version 3.16: Use PYTHONLEGACYWINDOWSFSENCODING instead.

sys.**stdin**

sys.**stdout**

sys.**stderr**

인터프리터가 표준 입력, 출력 및 에러에 사용하는 파일 객체:

- stdin는 모든 대화식 입력에 사용됩니다 (*input()* 호출을 포함합니다);
- stdout은 *print()*와 표현식 문장의 출력과 *input()*의 프롬프트에 사용됩니다;
- 인터프리터 자신의 프롬프트와 에러 메시지는 stderr로 갑니다.

이 스트림은 *open()* 함수에 의해 반환되는 것과 같은 일반적인 텍스트 파일입니다. 매개 변수는 다음과 같이 선택됩니다:

- The encoding and error handling are is initialized from PyConfig.stdio_encoding and PyConfig.stdio_errors.

  윈도우에서는, 콘솔 장치에 UTF-8이 사용됩니다. 디스크 파일과 파이프와 같은 비문자 장치는 시스템 로케일 인코딩(즉, ANSI 코드 페이지)을 사용합니다. NUL(즉, isatty() 가 True를 반환하는)과 같은 비 콘솔 문자 장치는 시작 시에 콘솔 입력과 출력 코드 페이지의 값을 각각 stdin과 stdout/stderr에 사용합니다. 프로세스가 초기에 콘솔에 연결되지 않았으면 시스템 로케일 인코딩이 기본값입니다.

  파이썬을 시작하기 전에 환경 변수 PYTHONLEGACYWINDOWSSTDIO를 설정하여 콘솔의 특수 동작을 재정의할 수 있습니다. 이 경우, 콘솔 코드 페이지는 다른 모든 문자 장치에서처럼 사용됩니다.

  모든 플랫폼에서, 파이썬을 시작하기 전에 PYTHONIOENCODING 환경 변수를 설정하거나 새로운 -X utf8 명령 줄 옵션과 PYTHONUTF8 환경 변수를 사용하여 문자 인코딩을 재정의할 수 있습니다. 그러나, 윈도우 콘솔의 경우, PYTHONLEGACYWINDOWSSTDIO 도 설정했을 때만 적용됩니다.

- 대화형일 때, stdout 스트림은 줄 버퍼링 됩니다. 그렇지 않으면, 일반 텍스트 파일처럼 블록 버퍼링 됩니다. stderr 스트림은 두 경우 모두 줄 버퍼링 됩니다. -u 명령 줄 옵션을 전달하거나 PYTHONUNBUFFERED 환경 변수를 설정하여 두 스트림을 모두 버퍼링하지 않을 수 있습니다.

버전 3.9에서 변경: 비 대화형 stderr은 이제 완전히 버퍼링 되는 대신 줄 버퍼링 됩니다.

> **참고**
>
> 표준 스트림에서 바이너리 데이터를 읽거나 표준 스트림으로 바이너리 데이터를 쓰려면, 하부 바이너리 *buffer* 객체를 사용하십시오. 예를 들어, 바이트열을 *stdout*에 쓰려면, sys.stdout.buffer.write(b'abc') 를 사용하십시오.

> 그러나, 라이브러리를 작성하고 있다면 (그리고 코드가 실행될 문맥을 제어하지 않으면), 표준 스트림은 `buffer` 어트리뷰트를 지원하지 않는 *io.StringIO*와 같은 파일류 객체로 대체 될 수 있음을 유의하십시오.

`sys.__stdin__`

`sys.__stdout__`

`sys.__stderr__`

이 객체는 프로그램 시작 시 `stdin`, `stderr` 및 `stdout`의 원래 값을 포함합니다. 이들은 파이널리제이션 중에 사용되며, `sys.std*` 객체가 리디렉션 되었는지에 관계없이 실제 표준 스트림으로 인쇄하는 데 유용할 수 있습니다.

또한 잘못된 객체로 덮어쓴 경우 실제 파일을 알려진 작동하는 파일 객체로 복원하는 데 사용할 수 있습니다. 그러나, 이를 수행하기 위해 선호되는 방법은 이전 스트림을 교체하기 전에 명시적으로 저장하고, 저장된 객체를 복원하는 것입니다.

> **참고**
>
> 일부 조건에서, `stdin`, `stdout` 및 `stderr` 뿐만 아니라 원래 값 `__stdin__`, `__stdout__` 및 `__stderr__`은 `None`일 수 있습니다. 보통 콘솔에 연결되지 않은 윈도우 GUI 앱과 **pythonw**로 시작된 파이썬 앱이 이런 경우입니다.

`sys.stdlib_module_names`

A frozenset of strings containing the names of standard library modules.

It is the same on all platforms. Modules which are not available on some platforms and modules disabled at Python build are also listed. All module kinds are listed: pure Python, built-in, frozen and extension modules. Test modules are excluded.

For packages, only the main package is listed: sub-packages and sub-modules are not listed. For example, the `email` package is listed, but the `email.mime` sub-package and the `email.message` sub-module are not listed.

See also the *sys.builtin_module_names* list.

Added in version 3.10.

`sys.thread_info`

스레드 구현에 대한 정보를 담은 네임드 튜플.

`thread_info.name`

스레드 구현 이름:

- `"nt"`: 윈도우 스레드
- `"pthread"`: POSIX 스레드
- `"pthread-stubs"`: stub POSIX threads (on WebAssembly platforms without threading support)
- `"solaris"`: 솔라리스 스레드

`thread_info.lock`

록 구현 이름:

- `"semaphore"`: 록은 세마포어를 사용합니다
- `"mutex+cond"`: 록은 뮤텍스(mutex)와 조건 변수(condition variable)를 사용합니다
- 이 정보를 알 수 없으면 `None`

`thread_info.version`

스레드 라이브러리의 이름과 버전. 문자열이거나, 이 정보를 알 수 없으면 `None`입니다.

Added in version 3.3.

sys.**tracebacklimit**

이 변수가 정숫값으로 설정되면, 처리되지 않은 예외가 발생할 때 인쇄되는 트레이스백 정보의 최대 수준 수를 결정합니다. 기본값은 1000 입니다. 0 이하로 설정하면, 모든 트레이스백 정보가 억제되고 예외 형과 값만 인쇄됩니다.

sys.**unraisablehook**(*unraisable*, /)

발생시킬 수 없는 예외(unraisable exception)를 처리합니다.

예외가 발생했지만, 파이썬이 예외를 처리할 방법이 없을 때 호출됩니다. 예를 들어, 파괴자가 예외를 발생시키거나 가비지 수거 (*gc.collect()*) 중에.

unraisable 인자에는 다음과 같은 어트리뷰트가 있습니다:

- exc_type: 예외 형.
- exc_value: 예외 값. None 일 수 있습니다.
- exc_traceback: 예외 트레이스백, None 일 수 있습니다.
- err_msg: 에러 메시지, None 일 수 있습니다.
- object: 예외를 발생시킨 객체, None 일 수 있습니다.

기본 훅은 err_msg 와 object 를 다음과 같이 포맷합니다: f'{err_msg}: {object!r}'; err_msg 가 None 이면 "Exception ignored in" 에러 메시지를 사용합니다.

sys.unraisablehook() 은 발생시킬 수 없는 예외 처리 방법을 제어하기 위해 재정의될 수 있습니다.

더 보기

잡히지 않은 예외를 처리하는 *excepthook()*.

경고

사용자 정의 훅을 사용하여 exc_value 를 저장하면 참조 순환이 만들어질 수 있습니다. 예외가 더는 필요하지 않을 때 참조 순환을 끊기 위해 명시적으로 지워야 합니다.

사용자 정의 훅을 사용하여 object 를 저장하면 파이널라이즈 중인 객체로 설정될 때 그것을 되살릴 수 있습니다. 객체 되살림을 방지하려면 사용자 정의 훅이 완료된 후 object 를 저장하지 마십시오.

처리할 수 없는 예외가 발생할 때 인자 *hook*, *unraisable* 로 감사 이벤트 sys.unraisablehook 을 발생시킵니다. *unraisable* 객체는 훅에 전달되는 것과 같습니다. 아무런 훅도 설정되지 않았으면, *hook* 은 None 일 수 있습니다.

Added in version 3.8.

sys.**version**

파이썬 인터프리터의 버전 번호와 빌드 번호 및 사용된 컴파일러에 대한 추가 정보가 포함된 문자열. 이 문자열은 대화식 인터프리터가 시작될 때 표시됩니다. 여기서 버전 정보를 추출하지 말고, *version_info* 와 *platform* 모듈이 제공하는 함수를 사용하십시오.

sys.**api_version**

The C API version, equivalent to the C macro PYTHON_API_VERSION. Defined for backwards compatibility.

Currently, this constant is not updated in new Python versions, and is not useful for versioning. This may change in the future.

sys.**version_info**

버전 번호의 5가지 구성 요소를 포함하는 튜플: *major*, *minor*, *micro*, *releaselevel* 및 *serial*. *releaselevel* 을 제외한 모든 값은 정수입니다; 릴리스 수준은 'alpha', 'beta', 'candidate' 또는 'final' 입니다.

파이썬 버전 2.0에 해당하는 `version_info` 값은 `(2, 0, 0, 'final', 0)` 입니다. 구성 요소는 이름으로도 액세스 할 수 있어서, `sys.version_info[0]`는 `sys.version_info.major`와 동등합니다.

버전 3.1에서 변경: 이름있는 구성 요소 어트리뷰트를 추가했습니다.

`sys.`**`warnoptions`**

이것은 경고 프레임워크의 구현 세부 사항입니다; 이 값을 수정하지 마십시오. 경고 프레임워크에 대한 자세한 정보는 *warnings* 모듈을 참조하십시오.

`sys.`**`winver`**

윈도우 플랫폼에서 레지스트리 키를 형성하는 데 사용되는 버전 번호. 이것은 파이썬 DLL에서 문자열 리소스 1000으로 저장됩니다. 값은 일반적으로 파이썬 인터프리터의 주와 부 버전입니다. 정보용으로 *sys* 모듈에서 제공됩니다; 이 값을 수정해도 파이썬에서 사용하는 레지스트리 키에는 영향을 미치지 않습니다.

가용성: Windows.

`sys.`**`monitoring`**

Namespace containing functions and constants for register callbacks and controlling monitoring events. See *sys.monitoring* for details.

`sys.`**`_xoptions`**

`-X` 명령 줄 옵션을 통해 전달된 다양한 구현 특정 플래그의 딕셔너리. 옵션 이름은 명시적으로 지정되면 그들의 값으로, 그렇지 않으면 *True*로 매핑됩니다. 예:

```
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
```

이는 `-X`를 통해 전달된 옵션에 액세스하는 CPython 특정 방법입니다. 다른 구현은 다른 수단을 통해, 또는 전혀 노출하지 않을 수 있습니다.

Added in version 3.2.

인용

29.2 `sys.monitoring` — Execution event monitoring

Added in version 3.12.

> **참고**
>
> *sys.monitoring* is a namespace within the *sys* module, not an independent module, so there is no need to `import sys.monitoring`, simply `import sys` and then use `sys.monitoring`.

This namespace provides access to the functions and constants necessary to activate and control event monitoring.

As programs execute, events occur that might be of interest to tools that monitor execution. The *sys.monitoring* namespace provides means to receive callbacks when events of interest occur.

The monitoring API consists of three components:

- *Tool identifiers*
- *Events*
- *Callbacks*

29.2.1 Tool identifiers

A tool identifier is an integer and the associated name. Tool identifiers are used to discourage tools from interfering with each other and to allow multiple tools to operate at the same time. Currently tools are completely independent and cannot be used to monitor each other. This restriction may be lifted in the future.

Before registering or activating events, a tool should choose an identifier. Identifiers are integers in the range 0 to 5 inclusive.

Registering and using tools

`sys.monitoring.`**`use_tool_id`**(*tool_id:* int, *name:* str, /) → *None*

Must be called before *tool_id* can be used. *tool_id* must be in the range 0 to 5 inclusive. Raises a `ValueError` if *tool_id* is in use.

`sys.monitoring.`**`clear_tool_id`**(*tool_id:* int, /) → *None*

Unregister all events and callback functions associated with *tool_id*.

`sys.monitoring.`**`free_tool_id`**(*tool_id:* int, /) → *None*

Should be called once a tool no longer requires *tool_id*. Will call `clear_tool_id()` before releasing *tool_id*.

`sys.monitoring.`**`get_tool`**(*tool_id:* int, /) → *str* | *None*

Returns the name of the tool if *tool_id* is in use, otherwise it returns `None`. *tool_id* must be in the range 0 to 5 inclusive.

All IDs are treated the same by the VM with regard to events, but the following IDs are pre-defined to make co-operation of tools easier:

```
sys.monitoring.DEBUGGER_ID = 0
sys.monitoring.COVERAGE_ID = 1
sys.monitoring.PROFILER_ID = 2
sys.monitoring.OPTIMIZER_ID = 5
```

29.2.2 Events

The following events are supported:

`sys.monitoring.events.`**`BRANCH_LEFT`**

A conditional branch goes left.

It is up to the tool to determine how to present "left" and "right" branches. There is no guarantee which branch is "left" and which is "right", except that it will be consistent for the duration of the program.

`sys.monitoring.events.`**`BRANCH_RIGHT`**

A conditional branch goes right.

`sys.monitoring.events.`**`CALL`**

A call in Python code (event occurs before the call).

`sys.monitoring.events.`**`C_RAISE`**

An exception raised from any callable, except for Python functions (event occurs after the exit).

`sys.monitoring.events.`**`C_RETURN`**

Return from any callable, except for Python functions (event occurs after the return).

`sys.monitoring.events.`**`EXCEPTION_HANDLED`**

An exception is handled.

`sys.monitoring.events.`**`INSTRUCTION`**

A VM instruction is about to be executed.

sys.monitoring.events.**JUMP**

An unconditional jump in the control flow graph is made.

sys.monitoring.events.**LINE**

An instruction is about to be executed that has a different line number from the preceding instruction.

sys.monitoring.events.**PY_RESUME**

Resumption of a Python function (for generator and coroutine functions), except for `throw()` calls.

sys.monitoring.events.**PY_RETURN**

Return from a Python function (occurs immediately before the return, the callee' s frame will be on the stack).

sys.monitoring.events.**PY_START**

Start of a Python function (occurs immediately after the call, the callee' s frame will be on the stack)

sys.monitoring.events.**PY_THROW**

A Python function is resumed by a `throw()` call.

sys.monitoring.events.**PY_UNWIND**

Exit from a Python function during exception unwinding. This includes exceptions raised directly within the function and that are allowed to continue to propagate.

sys.monitoring.events.**PY_YIELD**

Yield from a Python function (occurs immediately before the yield, the callee' s frame will be on the stack).

sys.monitoring.events.**RAISE**

An exception is raised, except those that cause a *STOP_ITERATION* event.

sys.monitoring.events.**RERAISE**

An exception is re-raised, for example at the end of a `finally` block.

sys.monitoring.events.**STOP_ITERATION**

An artificial *StopIteration* is raised; see *the STOP_ITERATION event*.

More events may be added in the future.

These events are attributes of the `sys.monitoring.events` namespace. Each event is represented as a power-of-2 integer constant. To define a set of events, simply bitwise OR the individual events together. For example, to specify both *PY_RETURN* and *PY_START* events, use the expression `PY_RETURN | PY_START`.

sys.monitoring.events.**NO_EVENTS**

An alias for `0` so users can do explicit comparisons like:

```
if get_events(DEBUGGER_ID) == NO_EVENTS:
    ...
```

Setting this event deactivates all events.

Local events

Local events are associated with normal execution of the program and happen at clearly defined locations. All local events can be disabled. The local events are:

- *PY_START*
- *PY_RESUME*
- *PY_RETURN*
- *PY_YIELD*
- *CALL*
- *LINE*
- *INSTRUCTION*

- *JUMP*
- *BRANCH_LEFT*
- *BRANCH_RIGHT*
- *STOP_ITERATION*

Deprecated event

- BRANCH

The BRANCH event is deprecated in 3.14. Using *BRANCH_LEFT* and *BRANCH_RIGHT* events will give much better performance as they can be disabled independently.

Ancillary events

Ancillary events can be monitored like other events, but are controlled by another event:

- *C_RAISE*
- *C_RETURN*

The *C_RETURN* and *C_RAISE* events are controlled by the *CALL* event. *C_RETURN* and *C_RAISE* events will only be seen if the corresponding *CALL* event is being monitored.

Other events

Other events are not necessarily tied to a specific location in the program and cannot be individually disabled.

The other events that can be monitored are:

- *PY_THROW*
- *PY_UNWIND*
- *RAISE*
- *EXCEPTION_HANDLED*

The STOP_ITERATION event

PEP 380 specifies that a *StopIteration* exception is raised when returning a value from a generator or coroutine. However, this is a very inefficient way to return a value, so some Python implementations, notably CPython 3.12+, do not raise an exception unless it would be visible to other code.

To allow tools to monitor for real exceptions without slowing down generators and coroutines, the *STOP_ITERATION* event is provided. *STOP_ITERATION* can be locally disabled, unlike *RAISE*.

Note that the *STOP_ITERATION* event and the *RAISE* event for a *StopIteration* exception are equivalent, and are treated as interchangeable when generating events. Implementations will favor *STOP_ITERATION* for performance reasons, but may generate a *RAISE* event with a *StopIteration*.

29.2.3 Turning events on and off

In order to monitor an event, it must be turned on and a corresponding callback must be registered. Events can be turned on or off by setting the events either globally and/or for a particular code object. An event will trigger only once, even if it is turned on both globally and locally.

Setting events globally

Events can be controlled globally by modifying the set of events being monitored.

sys.monitoring.**get_events**(*tool_id:* int, /) → *int*

Returns the int representing all the active events.

sys.monitoring.**set_events**(*tool_id:* int, *event_set:* int, /) → *None*

Activates all events which are set in *event_set*. Raises a `ValueError` if *tool_id* is not in use.

No events are active by default.

Per code object events

Events can also be controlled on a per code object basis. The functions defined below which accept a `types.CodeType` should be prepared to accept a look-alike object from functions which are not defined in Python (see c-api-monitoring).

sys.monitoring.**get_local_events**(*tool_id:* int, *code:* CodeType, /) → *int*

Returns all the local events for *code*

sys.monitoring.**set_local_events**(*tool_id:* int, *code:* CodeType, *event_set:* int, /) → *None*

Activates all the local events for *code* which are set in *event_set*. Raises a `ValueError` if *tool_id* is not in use.

Disabling events

sys.monitoring.**DISABLE**

A special value that can be returned from a callback function to disable events for the current code location.

Local events can be disabled for a specific code location by returning `sys.monitoring.DISABLE` from a callback function. This does not change which events are set, or any other code locations for the same event.

Disabling events for specific locations is very important for high performance monitoring. For example, a program can be run under a debugger with no overhead if the debugger disables all monitoring except for a few breakpoints.

sys.monitoring.**restart_events**() → *None*

Enable all the events that were disabled by `sys.monitoring.DISABLE` for all tools.

29.2.4 Registering callback functions

sys.monitoring.**register_callback**(*tool_id:* int, *event:* int, *func:* Callable | None, /) → *Callable* | *None*

Registers the callable *func* for the *event* with the given *tool_id*

If another callback was registered for the given *tool_id* and *event*, it is unregistered and returned. Otherwise `register_callback()` returns `None`.

Raises an *auditing event* `sys.monitoring.register_callback` with argument `func`.

Functions can be unregistered by calling `sys.monitoring.register_callback(tool_id, event, None)`.

Callback functions can be registered and unregistered at any time.

Callbacks are called only once regardless if the event is turned on both globally and locally. As such, if an event could be turned on for both global and local events by your code then the callback needs to be written to handle either trigger.

Callback function arguments

sys.monitoring.**MISSING**

A special value that is passed to a callback function to indicate that there are no arguments to the call.

When an active event occurs, the registered callback function is called. Callback functions returning an object other than `DISABLE` will have no effect. Different events will provide the callback function with different arguments, as follows:

- `PY_START` and `PY_RESUME`:

```
func(code: CodeType, instruction_offset: int) -> object
```

- `PY_RETURN` and `PY_YIELD`:

```
func(code: CodeType, instruction_offset: int, retval: object) -> object
```

- *CALL*, *C_RAISE* and *C_RETURN* (*arg0* can be *MISSING* specifically):

```
func(code: CodeType, instruction_offset: int, callable: object, arg0: object) -
↪> object
```

  code represents the code object where the call is being made, while *callable* is the object that is about to be called (and thus triggered the event). If there are no arguments, *arg0* is set to *sys.monitoring.MISSING*.

  For instance methods, *callable* will be the function object as found on the class with *arg0* set to the instance (i.e. the `self` argument to the method).

- *RAISE*, *RERAISE*, *EXCEPTION_HANDLED*, *PY_UNWIND*, *PY_THROW* and *STOP_ITERATION*:

```
func(code: CodeType, instruction_offset: int, exception: BaseException) ->␣
↪object
```

- *LINE*:

```
func(code: CodeType, line_number: int) -> object
```

- *BRANCH_LEFT*, *BRANCH_RIGHT* and *JUMP*:

```
func(code: CodeType, instruction_offset: int, destination_offset: int) ->␣
↪object
```

  Note that the *destination_offset* is where the code will next execute.

- *INSTRUCTION*:

```
func(code: CodeType, instruction_offset: int) -> object
```

29.3 sysconfig — 파이썬의 구성 정보에 접근하기

Added in version 3.2.

소스 코드: Lib/sysconfig

---

sysconfig 모듈은 설치 경로 목록과 현재 플랫폼과 관련된 구성 변수와 같은 파이썬 구성 정보에 대한 액세스를 제공합니다.

29.3.1 구성 변수

Python 배포판에는 `Makefile` 과 `pyconfig.h` 헤더 파일이 들어 있습니다. 이 파일은 파이썬 바이너리 자체와 `setuptools` 를 사용하여 컴파일된 타사 C 확장을 빌드하는 데 필요합니다.

sysconfig 는 *get_config_vars()* 또는 *get_config_var()* 를 사용하여 액세스 할 수 있는 딕셔너리에 이들 파일에 있는 모든 변수를 넣습니다.

윈도우에서는 훨씬 작은 세트입니다.

`sysconfig.`**`get_config_vars`**(**args*)

인자가 없으면, 현재 플랫폼과 관련된 모든 구성 변수의 딕셔너리를 반환합니다.

인자가 있으면, 인자를 사용하여 구성 변수 딕셔너리에서 각 인자를 조회한 결괏값 리스트를 돌려줍니다.

인자별로, 값이 없으면 `None` 을 반환합니다.

sysconfig.**get_config_var**(*name*)

하나의 변수 *name* 의 값을 반환합니다. `get_config_vars().get(name)` 과 같습니다.

name 을 찾지 못하면 `None` 을 반환합니다.

사용 예:

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

29.3.2 설치 경로

파이썬은 플랫폼과 설치 옵션에 따라 다른 설치 스킴을 사용합니다. 이 스킴은 `os.name` 에 의해 반환된 값을 기반으로 하는 고유한 식별자로 `sysconfig` 에 저장됩니다. 이 스킴은 패키지 설치기가 파일을 복사할 위치를 결정하는 데 사용됩니다.

파이썬은 현재 9가지 스킴을 지원합니다:

- *posix_prefix*: 리눅스나 맥 OS 같은 포직스(POSIX) 플랫폼용 스킴. 이것은 파이썬이나 컴포넌트가 설치될 때 사용되는 기본 스킴입니다.
- *posix_home*: *home* 옵션이 사용될 때, 포직스 플랫폼용 스킴. 이 스킴은 특정 홈 접두어 아래에 있는 경로를 정의합니다.
- *posix_user*: *user* 옵션이 사용될 때, 포직스 플랫폼용 스킴. 이 스킴은 사용자 홈 디렉터리 아래에 있는 경로를 정의합니다 (`site.USER_BASE`).
- *posix_venv*: scheme for `Python virtual environments` on POSIX platforms; by default it is the same as *posix_prefix*.
- *nt*: 윈도우용 스킴. 이것은 파이썬이나 컴포넌트가 설치될 때 사용되는 기본 스킴입니다.
- *nt_user*: *user* 옵션이 사용될 때, 윈도우용 스킴.
- *nt_venv*: scheme for `Python virtual environments` on Windows; by default it is the same as *nt*.
- *venv*: a scheme with values from either *posix_venv* or *nt_venv* depending on the platform Python runs on.
- *osx_framework_user*: *user* 옵션이 사용될, 맥 OS용 스킴.

각 스킴은 일련의 경로로 구성되며 각 경로는 고유한 식별자를 가집니다. 파이썬은 현재 8개의 경로를 사용합니다:

- *stdlib*: 플랫폼마다 다르지 않은 표준 파이썬 라이브러리 파일이 들어있는 디렉터리.
- *platstdlib*: 플랫폼마다 다른 표준 파이썬 라이브러리 파일이 들어있는 디렉터리.
- *platlib*: 사이트마다, 플랫폼마다 다른 파일용 디렉터리.
- *purelib*: 사이트마다 다르지만, 플랫폼마다 다르지 않은 파일이 들어있는 디렉터리 ('순수한' 파이썬').
- *include*: 파이썬 C-API 를 위한 플랫폼마다 다르지 않은 헤더 파일용 디렉터리.
- *platinclude*: 파이썬 C-API 를 위한 플랫폼마다 다른 헤더 파일용 디렉터리.
- *scripts*: 스크립트 파일용 디렉터리.
- *data*: 데이터 파일용 디렉터리.

29.3.3 User scheme

This scheme is designed to be the most convenient solution for users that don' t have write permission to the global site-packages directory or don' t want to install into it.

Files will be installed into subdirectories of `site.USER_BASE` (written as `userbase` hereafter). This scheme installs pure Python modules and extension modules in the same location (also known as `site.USER_SITE`).

posix_user

경로	설치 디렉터리
stdlib	`userbase/lib/pythonX.Y`
platstdlib	`userbase/lib/pythonX.Y`
platlib	`userbase/lib/pythonX.Y/site-packages`
purelib	`userbase/lib/pythonX.Y/site-packages`
include	`userbase/include/pythonX.Y`
scripts	`userbase/bin`
data	`userbase`

nt_user

경로	설치 디렉터리
stdlib	`userbase\PythonXY`
platstdlib	`userbase\PythonXY`
platlib	`userbase\PythonXY\site-packages`
purelib	`userbase\PythonXY\site-packages`
include	`userbase\PythonXY\Include`
scripts	`userbase\PythonXY\Scripts`
data	`userbase`

osx_framework_user

경로	설치 디렉터리
stdlib	`userbase/lib/python`
platstdlib	`userbase/lib/python`
platlib	`userbase/lib/python/site-packages`
purelib	`userbase/lib/python/site-packages`
include	`userbase/include/pythonX.Y`
scripts	`userbase/bin`
data	`userbase`

29.3.4 Home scheme

The idea behind the "home scheme" is that you build and maintain a personal stash of Python modules. This scheme' s name is derived from the idea of a "home" directory on Unix, since it' s not unusual for a Unix user to make their home directory have a layout similar to `/usr/` or `/usr/local/`. This scheme can be used by anyone, regardless of the operating system they are installing for.

posix_home

경로	설치 디렉터리
stdlib	*home*/lib/python
platstdlib	*home*/lib/python
platlib	*home*/lib/python
purelib	*home*/lib/python
include	*home*/include/python
platinclude	*home*/include/python
scripts	*home*/bin
data	*home*

29.3.5 Prefix scheme

The "prefix scheme" is useful when you wish to use one Python installation to perform the build/install (i.e., to run the setup script), but install modules into the third-party module directory of a different Python installation (or something that looks like a different Python installation). If this sounds a trifle unusual, it is—that' s why the user and home schemes come before. However, there are at least two known cases where the prefix scheme will be useful.

First, consider that many Linux distributions put Python in `/usr`, rather than the more traditional `/usr/local`. This is entirely appropriate, since in those cases Python is part of "the system" rather than a local add-on. However, if you are installing Python modules from source, you probably want them to go in `/usr/local/lib/python2.X` rather than `/usr/lib/python2.X`.

Another possibility is a network filesystem where the name used to write to a remote directory is different from the name used to read it: for example, the Python interpreter accessed as `/usr/local/bin/python` might search for modules in `/usr/local/lib/python2.X`, but those modules would have to be installed to, say, `/mnt/@server/export/lib/python2.X`.

posix_prefix

경로	설치 디렉터리
stdlib	*prefix*/lib/python*X.Y*
platstdlib	*prefix*/lib/python*X.Y*
platlib	*prefix*/lib/python*X.Y*/site-packages
purelib	*prefix*/lib/python*X.Y*/site-packages
include	*prefix*/include/python*X.Y*
platinclude	*prefix*/include/python*X.Y*
scripts	*prefix*/bin
data	*prefix*

nt

경로	설치 디렉터리
stdlib	*prefix*\Lib
platstdlib	*prefix*\Lib
platlib	*prefix*\Lib\site-packages
purelib	*prefix*\Lib\site-packages
include	*prefix*\Include
platinclude	*prefix*\Include
scripts	*prefix*\Scripts
data	*prefix*

29.3.6 설치 경로 함수

sysconfig 는 이러한 설치 경로를 결정하는 몇 가지 함수를 제공합니다.

sysconfig.**get_scheme_names**()

현재 *sysconfig* 에서 지원되는 모든 스킴을 포함하는 튜플을 돌려줍니다.

sysconfig.**get_default_scheme**()

현재 플랫폼에 대한 기본 스킴 이름을 반환합니다.

Added in version 3.10: This function was previously named `_get_default_scheme()` and considered an implementation detail.

버전 3.11에서 변경: When Python runs from a virtual environment, the *venv* scheme is returned.

sysconfig.**get_preferred_scheme**(*key*)

Return a preferred scheme name for an installation layout specified by *key*.

key must be either `"prefix"`, `"home"`, or `"user"`.

The return value is a scheme name listed in *get_scheme_names()*. It can be passed to *sysconfig* functions that take a *scheme* argument, such as *get_paths()*.

Added in version 3.10.

버전 3.11에서 변경: When Python runs from a virtual environment and `key="prefix"`, the *venv* scheme is returned.

sysconfig.**_get_preferred_schemes**()

Return a dict containing preferred scheme names on the current platform. Python implementers and redistributors may add their preferred schemes to the `_INSTALL_SCHEMES` module-level global value, and modify this function to return those scheme names, to e.g. provide different schemes for system and language package managers to use, so packages installed by either do not mix with those by the other.

End users should not use this function, but *get_default_scheme()* and *get_preferred_scheme()* instead.

Added in version 3.10.

sysconfig.**get_path_names**()

현재 *sysconfig* 에서 지원되는 모든 경로명을 포함하는 튜플을 돌려줍니다.

sysconfig.**get_path**(*name*[, *scheme*[, *vars*[, *expand*]]])

scheme 이라는 설치 스킴에서, 경로 *name* 에 해당하는 설치 경로를 돌려줍니다.

name 은 *get_path_names()* 가 돌려주는 리스트에 있는 값이어야 합니다.

sysconfig 는 각 경로명에 해당하는 설치 경로를 플랫폼별로 확장할 변수와 함께 저장합니다. 예를 들어, *nt* 스킴의 *stdlib* 경로는 `{base}/Lib` 입니다.

get_path() 는 *get_config_vars()* 에 의해 반환된 변수를 사용하여 경로를 확장합니다. 모든 변수는 각 플랫폼에 대한 기본값을 가지므로 이 함수를 호출하고 기본값을 가져올 수 있습니다.

scheme 이 제공되면, 그것은 *get_scheme_names()* 에 의해 반환된 리스트에 있는 값이어야 합니다. 그렇지 않으면, 현재 플랫폼의 기본 스킴이 사용됩니다.

vars 가 제공되면, *get_config_vars()* 에 의해 반환된 딕셔너리를 갱신할 변수의 딕셔너리여야 합니다.

expand 가 `False` 로 설정되면, 경로는 변수를 사용하여 확장되지 않습니다.

name 을 찾지 못하면, *KeyError*를 발생시킵니다.

sysconfig.**get_paths**([*scheme*[, *vars*[, *expand*]]])

설치 스킴에 해당하는 모든 설치 경로를 포함하는 딕셔너리를 돌려줍니다. 자세한 정보는 *get_path()* 를 보십시오.

scheme 을 제공하지 않으면, 현재 플랫폼에 대한 기본 스킴을 사용합니다.

vars 가 제공되면, 경로를 확장하는 데 사용되는 딕셔너리를 갱신하는 변수의 딕셔너리여야 합니다.

expand 를 거짓으로 설정하면 경로가 확장되지 않습니다.

scheme 이 존재하는 스킴이 아니면, `get_paths()` 는 `KeyError` 를 발생시킵니다.

29.3.7 기타 함수

`sysconfig.`**`get_python_version`**`()`

`MAJOR.MINOR` 파이썬 버전 번호를 문자열로 반환합니다. `'%d.%d' % sys.version_info[:2]` 와 유사합니다.

`sysconfig.`**`get_platform`**`()`

현재의 플랫폼을 식별하는 문자열을 돌려줍니다.

이는 주로 플랫폼별 빌드 디렉터리와 플랫폼별로 빌드 된 배포판을 구별하기 위해 사용됩니다. 포함된 정확한 정보는 OS에 따라 다르지만, 일반적으로 OS 이름과 버전 및 아키텍처를 포함합니다 (`os.uname()` 에서 제공됩니다); 예를 들어, 리눅스에서 커널 버전은 특별히 중요하지 않습니다.

반환 값의 예:

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u

윈도우는 다음 중 하나를 반환합니다:

- win-amd64 (AMD64의 64비트 윈도우, 일명 x86_64, Intel64, EM64T 등)
- win-arm64 (ARM64의 64비트 윈도우, 일명 AArch64)
- win32 (기타 모든 것 - 구체적으로, sys.platform이 반환됩니다)

맥 OS는 다음을 반환 할 수 있습니다:

- macosx-10.6-ppc
- macosx-10.4-ppc64
- macosx-10.3-i386
- macosx-10.4-fat

다른 포직스 이외의 플랫폼의 경우, 현재는 `sys.platform` 만 반환합니다.

`sysconfig.`**`is_python_build`**`()`

실행 중인 파이썬 인터프리터가 소스에서 빌드되어 빌드 된 위치에서 실행되고, `make install` 을 실행하거나 바이너리 설치 프로그램을 통해 설치 한 결과가 아닌 위치에서 실행되는 것이 아니라면 `True` 를 반환합니다.

`sysconfig.`**`parse_config_h`**`(`*fp*[, *vars*]`)`

`config.h`-스타일 파일을 해석합니다.

fp 는 `config.h`-류 파일을 가리키는 파일류 객체입니다.

이름/값 쌍을 포함하는 딕셔너리가 반환됩니다. 선택적 딕셔너리가 두 번째 인자로 전달되면, 새 사전 대신 사용되며 파일에서 읽은 값으로 갱신됩니다.

`sysconfig.`**`get_config_h_filename`**`()`

`pyconfig.h` 의 경로를 반환합니다.

`sysconfig.`**`get_makefile_filename`**`()`

`Makefile` 의 경로를 반환합니다.

29.3.8 Command-line usage

sysconfig 를 파이썬의 *-m* 옵션으로 스크립트로 사용할 수 있습니다:

```
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"

Paths:
        data = "/usr/local"
        include = "/Users/tarek/Dev/svn.python.org/py3k/Include"
        platinclude = "."
        platlib = "/usr/local/lib/python3.2/site-packages"
        platstdlib = "/usr/local/lib/python3.2"
        purelib = "/usr/local/lib/python3.2/site-packages"
        scripts = "/usr/local/bin"
        stdlib = "/usr/local/lib/python3.2"

Variables:
        AC_APPLE_UNIVERSAL_BUILD = "0"
        AIX_GENUINE_CPLUSPLUS = "0"
        AR = "ar"
        ARFLAGS = "rc"
        ...
```

이 호출은 *get_platform()*, *get_python_version()*, *get_path()* 및 *get_config_vars()* 에 의해 반환된 정보를 표준 출력에 인쇄합니다.

29.4 builtins — 내장 객체

이 모듈은 파이썬의 모든 '내장' 식별자에 대한 직접 액세스를 제공합니다. 예를 들어, `builtins.open` 은 내장 함수 *open()* 의 완전한 이름입니다.

이 모듈은 일반적으로 대부분의 응용 프로그램에서 명시적으로 액세스하지 않지만, 내장된 값과 이름이 같은 객체를 제공하면서도 그 이름의 내장 객체가 필요한 모듈에서 유용 할 수 있습니다. 예를 들어, 내장 *open()* 을 감싸는 *open()* 함수를 구현하고자 하는 모듈에서 이 모듈을 직접 사용할 수 있습니다:

```
import builtins

def open(path):
    f = builtins.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''출력을 대문자로 변환하도록 파일을 감싸는 래퍼.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

구현 세부 사항으로, 대부분 모듈은 전역 변수로 `__builtins__` 라는 이름을 가지고 있습니다. `__builtins__` 의 값은, 보통 이 모듈이거나 모듈의 `__dict__` 어트리뷰트의 값입니다. 이것은 구현 세

부 사항이므로, 파이썬의 대안 구현에서는 사용되지 않을 수 있습니다.

더 보기

- 내장 상수
- 내장 예외
- 내장 함수
- 내장형

29.5 __main__ — 최상위 코드 환경

In Python, the special name `__main__` is used for two important constructs:

1. the name of the top-level environment of the program, which can be checked using the `__name__ == '__main__'` expression; and
2. the `__main__.py` file in Python packages.

Both of these mechanisms are related to Python modules; how users interact with them and how they interact with each other. They are explained in detail below. If you' re new to Python modules, see the tutorial section tut-modules for an introduction.

29.5.1 __name__ == '__main__'

When a Python module or package is imported, `__name__` is set to the module' s name. Usually, this is the name of the Python file itself without the `.py` extension:

```
>>> import configparser
>>> configparser.__name__
'configparser'
```

If the file is part of a package, `__name__` will also include the parent package' s path:

```
>>> from concurrent.futures import process
>>> process.__name__
'concurrent.futures.process'
```

However, if the module is executed in the top-level code environment, its `__name__` is set to the string `'__main__'`.

"최상위 코드 환경" 이란 무엇인가요?

`__main__` is the name of the environment where top-level code is run. "Top-level code" is the first user-specified Python module that starts running. It' s "top-level" because it imports all other modules that the program needs. Sometimes "top-level code" is called an *entry point* to the application.

The top-level code environment can be:

- the scope of an interactive prompt:

```
>>> __name__
'__main__'
```

- the Python module passed to the Python interpreter as a file argument:

```
$ python helloworld.py
Hello, world!
```

- the Python module or package passed to the Python interpreter with the `-m` argument:

```
$ python -m tarfile
usage: tarfile.py [-h] [-v] (...)
```

- Python code read by the Python interpreter from standard input:

```
$ echo "import this" | python
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
...
```

- Python code passed to the Python interpreter with the `-c` argument:

```
$ python -c "import this"
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
...
```

In each of these situations, the top-level module' s `__name__` is set to `'__main__'`.

결과적으로, 모듈은 자신의 `__name__` 을 검사하여 최상위 환경에서 실행 중인지를 확인할 수 있습니다. 이 때문에 모듈이 임포트 문에서 초기화되지 않을 때 조건부로 코드를 실행하는 공통 관용구를 사용할 수 있습니다:

```
if __name__ == '__main__':
    # 모듈이 임포트 문에서 초기화되지 않을 때 실행합니다.
    ...
```

더 보기

For a more detailed look at how `__name__` is set in all situations, see the tutorial section tut-modules.

Idiomatic Usage

Some modules contain code that is intended for script use only, like parsing command-line arguments or fetching data from standard input. If a module like this was imported from a different module, for example to unit test it, the script code would unintentionally execute as well.

This is where using the `if __name__ == '__main__'` code block comes in handy. Code within this block won' t run unless the module is executed in the top-level environment.

Putting as few statements as possible in the block below `if __name__ == '__main__'` can improve code clarity and correctness. Most often, a function named `main` encapsulates the program' s primary behavior:

```
# echo.py

import shlex
import sys

def echo(phrase: str) -> None:
    """print 를 둘러싼 더미 래퍼."""
    # 시연의 목적으로, 이 함수 안에 쓸모있고 재사용할 수 있는
    #  로직이 있다고 상상할 수 있습니다
    print(phrase)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def main() -> int:
    """입력 인자를 표준 출력으로 에코합니다"""
    phrase = shlex.join(sys.argv)
    echo(phrase)
    return 0

if __name__ == '__main__':
    sys.exit(main())  # 다음 절에서 sys.exit 의 사용에 대해 설명합니다
```

Note that if the module didn' t encapsulate code inside the `main` function but instead put it directly within the `if __name__ == '__main__'` block, the `phrase` variable would be global to the entire module. This is error-prone as other functions within the module could be unintentionally using the global variable instead of a local name. A `main` function solves this problem.

Using a `main` function has the added benefit of the `echo` function itself being isolated and importable elsewhere. When `echo.py` is imported, the `echo` and `main` functions will be defined, but neither of them will be called, because `__name__ != '__main__'`.

Packaging Considerations

`main` functions are often used to create command-line tools by specifying them as entry points for console scripts. When this is done, pip inserts the function call into a template script, where the return value of `main` is passed into *`sys.exit()`*. For example:

```
sys.exit(main())
```

Since the call to `main` is wrapped in *`sys.exit()`*, the expectation is that your function will return some value acceptable as an input to *`sys.exit()`*; typically, an integer or `None` (which is implicitly returned if your function does not have a return statement).

By proactively following this convention ourselves, our module will have the same behavior when run directly (i.e. `python echo.py`) as it will have if we later package it as a console script entry-point in a pip-installable package.

In particular, be careful about returning strings from your `main` function. *`sys.exit()`* will interpret a string argument as a failure message, so your program will have an exit code of `1`, indicating failure, and the string will be written to *`sys.stderr`*. The `echo.py` example from earlier exemplifies using the `sys.exit(main())` convention.

> **더 보기**
>
> Python Packaging User Guide contains a collection of tutorials and references on how to distribute and install Python packages with modern tools.

29.5.2 `__main__.py` in Python Packages

If you are not familiar with Python packages, see section tut-packages of the tutorial. Most commonly, the `__main__.py` file is used to provide a command-line interface for a package. Consider the following hypothetical package, "bandclass":

```
bandclass
  □□□ __init__.py
  □□□ __main__.py
  □□□ student.py
```

`__main__.py` will be executed when the package itself is invoked directly from the command line using the `-m` flag. For example:

```
$ python -m bandclass
```

This command will cause `__main__.py` to run. How you utilize this mechanism will depend on the nature of the package you are writing, but in this hypothetical case, it might make sense to allow the teacher to search for students:

```
# bandclass/__main__.py

import sys
from .student import search_students

student_name = sys.argv[1] if len(sys.argv) >= 2 else ''
print(f'Found student: {search_students(student_name)}')
```

Note that `from .student import search_students` is an example of a relative import. This import style can be used when referencing modules within a package. For more details, see intra-package-references in the tut-modules section of the tutorial.

Idiomatic Usage

The content of `__main__.py` typically isn' t fenced with an `if __name__ == '__main__'` block. Instead, those files are kept short and import functions to execute from other modules. Those other modules can then be easily unit-tested and are properly reusable.

If used, an `if __name__ == '__main__'` block will still work as expected for a `__main__.py` file within a package, because its `__name__` attribute will include the package' s path if imported:

```
>>> import asyncio.__main__
>>> asyncio.__main__.__name__
'asyncio.__main__'
```

This won' t work for `__main__.py` files in the root directory of a `.zip` file though. Hence, for consistency, a minimal `__main__.py` without a `__name__` check is preferred.

> **더 보기**
>
> See *venv* for an example of a package with a minimal `__main__.py` in the standard library. It doesn' t contain a `if __name__ == '__main__'` block. You can invoke it with `python -m venv [directory]`.
>
> See *runpy* for more details on the `-m` flag to the interpreter executable.
>
> See *zipapp* for how to run applications packaged as *.zip* files. In this case Python looks for a `__main__.py` file in the root directory of the archive.

29.5.3 import __main__

Regardless of which module a Python program was started with, other modules running within that same program can import the top-level environment' s scope (*namespace*) by importing the `__main__` module. This doesn' t import a `__main__.py` file but rather whichever module that received the special name `'__main__'`.

Here is an example module that consumes the `__main__` namespace:

```
# namely.py

import __main__

def did_user_define_their_name():
    return 'my_name' in dir(__main__)

def print_user_name():
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    if not did_user_define_their_name():
        raise ValueError('Define the variable `my_name`!')

    print(__main__.my_name)
```

Example usage of this module could be as follows:

```
# start.py

import sys

from namely import print_user_name

# my_name = "Dinsdale"

def main():
    try:
        print_user_name()
    except ValueError as ve:
        return str(ve)

if __name__ == "__main__":
    sys.exit(main())
```

Now, if we started our program, the result would look like this:

```
$ python start.py
Define the variable `my_name`!
```

The exit code of the program would be 1, indicating an error. Uncommenting the line with `my_name = "Dinsdale"` fixes the program and now it exits with status code 0, indicating success:

```
$ python start.py
Dinsdale
```

Note that importing `__main__` doesn' t cause any issues with unintentionally running top-level code meant for script use which is put in the `if __name__ == "__main__"` block of the `start` module. Why does this work?

Python inserts an empty `__main__` module in *`sys.modules`* at interpreter startup, and populates it by running top-level code. In our example this is the `start` module which runs line by line and imports `namely`. In turn, `namely` imports `__main__` (which is really `start`). That' s an import cycle! Fortunately, since the partially populated `__main__` module is present in *`sys.modules`*, Python passes that to `namely`. See Special considerations for __main__ in the import system' s reference for details on how this works.

The Python REPL is another example of a "top-level environment", so anything defined in the REPL becomes part of the `__main__` scope:

```
>>> import namely
>>> namely.did_user_define_their_name()
False
>>> namely.print_user_name()
Traceback (most recent call last):
...
ValueError: Define the variable `my_name`!
>>> my_name = 'Jabberwocky'
>>> namely.did_user_define_their_name()
True
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> namely.print_user_name()
Jabberwocky
```

The __main__ scope is used in the implementation of *pdb* and *rlcompleter*.

29.6 warnings — 경고 제어

소스 코드: Lib/warnings.py

경고 메시지는 일반적으로 프로그램에서 사용자에게 (일반적으로) 예외를 발생시키거나 프로그램을 종료하는 것을 보증하지 않는 특정 조건에 대해 경고하는 것이 유용한 상황 상황에서 발행됩니다. 예를 들어, 프로그램이 더는 사용되지 않는 모듈을 사용할 때 경고를 발행하려고 할 수 있습니다.

파이썬 프로그래머는 이 모듈에 정의된 *warn()* 함수를 호출하여 경고를 발행합니다. (C 프로그래머는 PyErr_WarnEx() 를 사용합니다; 자세한 내용은 exceptionhandling를 참조하십시오).

경고 메시지는 일반적으로 *sys.stderr*에 기록되지만, 모든 경고를 무시하는 것에서 예외로 변경하는 것에 이르기까지 배치를 유연하게 변경할 수 있습니다. 경고의 처리는 경고 범주, 경고 메시지의 텍스트 및 발행된 소스 위치에 따라 달라질 수 있습니다. 같은 소스 위치에 대한 특정 경고의 반복은 일반적으로 억제됩니다.

경고 제어에는 두 가지 단계가 있습니다; 첫째, 경고가 발행될 때마다, 메시지를 발행할지를 결정합니다; 다음으로, 메시지가 발행된다면, 사용자 설정 가능한 훅을 사용하여 포맷되고 인쇄됩니다.

경고 메시지를 발행할지는 경고 필터에 의해 제어되며, 이는 일치 규칙과 조치의 시퀀스입니다. *filterwarnings()*를 호출하여 규칙을 필터에 추가하고 *resetwarnings()*를 호출하여 기본 상태로 재설정할 수 있습니다.

경고 메시지의 인쇄는 *showwarning()*을 호출하여 수행되며, 이는 재정의될 수 있습니다; 이 함수의 기본 구현은 *formatwarning()*을 호출하여 메시지를 포맷하며, 사용자 정의 구현에서도 사용할 수 있습니다.

더 보기

*logging.captureWarnings()*를 사용하면 표준 로깅 인프라로 모든 경고를 처리할 수 있습니다.

29.6.1 경고 범주

경고 범주를 나타내는 여러 가지 내장 예외가 있습니다. 이 범주화는 경고 그룹을 필터링하는 데 유용합니다.

이들은 기술적으로 내장 예외이지만, 개념적으로 경고 메커니즘에 속하기 때문에 여기에서 설명합니다.

사용자 코드는 표준 경고 범주 중 하나를 서브 클래싱 하여 추가 경고 범주를 정의할 수 있습니다. 경고 범주는 항상 *Warning* 클래스의 서브 클래스여야 합니다.

다음과 같은 경고 범주 클래스가 현재 정의되어 있습니다:

클래스	설명
`Warning`	이것은 모든 경고 범주 클래스의 베이스 클래스입니다. `Exception`의 서브 클래스입니다.
`UserWarning`	`warn()`의 기본 범주.
`DeprecationWarning`	폐지된 기능에 대한 경고의 베이스 범주, 경고가 다른 파이썬 개발자를 대상으로 할 때 (`__main__`에 있는 코드로 트리거 되지 않는 한 기본적으로 무시됩니다).
`SyntaxWarning`	Base category for warnings about dubious syntactic features (typically emitted when compiling Python source code, and hence may not be suppressed by runtime filters)
`RuntimeWarning`	모호한 런타임 기능에 대한 경고의 베이스 범주.
`FutureWarning`	폐지된 기능에 대한 경고의 베이스 범주, 경고가 파이썬으로 작성된 응용 프로그램의 최종 사용자를 대상으로 할 때.
`PendingDeprecationWarning`	향후 폐지될 기능에 대한 경고의 베이스 범주 (기본적으로 무시됩니다).
`ImportWarning`	모듈을 임포트 하는 과정에서 트리거 되는 경고의 베이스 범주 (기본적으로 무시됩니다).
`UnicodeWarning`	유니코드와 관련된 경고의 베이스 범주.
`BytesWarning`	`bytes`와 `bytearray`와 관련된 경고의 베이스 범주.
`ResourceWarning`	자원 사용과 관련된 경고의 베이스 범주 (기본적으로 무시됩니다).

버전 3.7에서 변경: 이전에는 `DeprecationWarning`과 `FutureWarning`은 기능이 완전히 제거되었는지 또는 동작을 변경하는지에 따라 구별되었습니다. 이제 의도한 대상과 기본 경고 필터에서 처리하는 방식에 따라 구별됩니다.

29.6.2 경고 필터

경고 필터는 경고를 무시, 표시 또는 에러로 전환(예외 발생)할지를 제어합니다.

개념적으로, 경고 필터는 필터 명세의 순서 있는 목록을 유지합니다; 일치가 발견될 때까지 목록의 각 필터 명세에 대해 특정 경고를 일치시킵니다; 필터는 일치의 처리를 결정합니다. 각 항목은 (*action*, *message*, *category*, *module*, *lineno*) 형식의 튜플입니다, 여기서:

- *action*은 다음 문자열 중 하나입니다:

값	처리
`"default"`	경고가 발행된 각 위치(모듈 + 줄 번호)에 대해 일치하는 경고의 첫 번째 발생을 인쇄합니다
`"error"`	일치하는 경고를 예외로 바꿉니다
`"ignore"`	일치하는 경고를 인쇄하지 않습니다
`"always"`	일치하는 경고를 항상 인쇄합니다
`"all"`	alias to "always"
`"module"`	경고가 발행된 모듈마다 (줄 번호와 관계없이) 일치하는 경고의 첫 번째 발생을 인쇄합니다
`"once"`	위치와 관계없이 일치하는 경고의 첫 번째 발생만 인쇄합니다

- *message*는 경고 메시지의 시작 부분이 대소문자를 구분 없이 일치해야 하는 정규식을 포함하는 문자열입니다. `-W`와 `PYTHONWARNINGS`에서, *message*는 경고 메시지의 시작 부분에 (대소문자 구분 없이) 포함되어야 하는 리터럴 문자열로, *message*의 시작이나 끝에 있는 공백은 무시합니다.
- *category*는 클래스(`Warning`의 서브 클래스)이며, 일치하는 경고 범주는 이것의 서브 클래스여야 합니다.
- *module*은 완전히 정규화된 모듈 이름의 시작 부분이 대소문자를 구분하여 일치해야 하는 정규식을 포함하는 문자열입니다. `-W`와 `PYTHONWARNINGS`에서, *module*은 완전히 정규화된 모듈 이름과 (대소문자 구분하여) 같아야 하는 리터럴 문자열로, *module*의 시작이나 끝에 있는 공백은 무시합니다.

- *lineno*는 경고가 발생한 줄 번호가 일치해야 하는 정수이거나, 모든 줄 번호와 일치하려면 `0` 입니다.

Warning 클래스는 내장 *Exception* 클래스에서 파생되므로, 경고를 에러로 바꾸려면 단순히 `category(message)` 를 raise 합니다.

경고가 보고되고 등록된 필터와 일치하지 않으면 "default" 조치가 적용됩니다 (그래서 그런 이름을 갖고 있습니다).

Repeated Warning Suppression Criteria

The filters that suppress repeated warnings apply the following criteria to determine if a warning is considered a repeat:

- `"default"`: A warning is considered a repeat only if the (*message*, *category*, *module*, *lineno*) are all the same.
- `"module"`: A warning is considered a repeat if the (*message*, *category*, *module*) are the same, ignoring the line number.
- `"once"`: A warning is considered a repeat if the (*message*, *category*) are the same, ignoring the module and line number.

경고 필터 설명

경고 필터는 파이썬 인터프리터 명령 줄로 전달된 `-W` 옵션과 `PYTHONWARNINGS` 환경 변수로 초기화됩니다. 인터프리터는 *sys.warnoptions*에서 제공된 모든 항목에 대한 인자를 해석하지 않고 저장합니다; *warnings* 모듈은 처음 임포트 될 때 이를 구문 분석합니다 (유효하지 않은 옵션은 메시지를 *sys.stderr*에 인쇄한 후 무시됩니다).

개별 경고 필터는 콜론으로 구분된 필드의 시퀀스로 지정됩니다:

```
action:message:category:module:line
```

이러한 각 필드의 의미는 경고 필터에 설명된 대로입니다. 한 줄에 여러 필터를 나열할 때 (`PYTHONWARNINGS` 와 같이), 개별 필터는 쉼표로 구분되고 나중에 나열된 필터가 그 앞에 나열된 필터보다 우선합니다 (왼쪽에서 오른쪽으로 적용되고, 가장 최근에 적용된 필터가 앞서 나온 필터에 우선하기 때문입니다).

일반적으로 사용되는 경고 필터는 모든 경고, 특정 범주의 경고 또는 특정 모듈이나 패키지에서 발생하는 경고에 적용됩니다. 몇 가지 예:

```
default                      # 모든 경고를 표시합니다 (설사 기본적으로 무시되더라도)
ignore                       # 모든 경고를 무시합니다
error                        # 모든 경고를 에러로 변환합니다
error::ResourceWarning       # ResourceWarning 메시지를 에러로 취급합니다
default::DeprecationWarning  # DeprecationWarning 메시지를 표시합니다
ignore,default:::mymodule    # "mymodule"이 트리거 한 경고만 보고합니다
error:::mymodule             # "mymodule"에서 경고를 에러로 변환합니다
```

기본 경고 필터

기본적으로, 파이썬은 `-W` 명령 줄 옵션, `PYTHONWARNINGS` 환경 변수 및 *filterwarnings()* 호출로 재정의할 수 있는 몇 가지 경고 필터를 설치합니다.

정규 릴리스 빌드에서, 기본 경고 필터에는 다음과 같은 항목이 있습니다 (우선순위 순서로):

```
default::DeprecationWarning:__main__
ignore::DeprecationWarning
ignore::PendingDeprecationWarning
ignore::ImportWarning
ignore::ResourceWarning
```

디버그 빌드에서, 기본 경고 필터 목록은 비어 있습니다.

버전 3.2에서 변경: *DeprecationWarning*은 이제 *PendingDeprecationWarning* 에 더해 기본적으로 무시됩니다.

버전 3.7에서 변경: *DeprecationWarning*은 __main__의 코드에 의해 직접 트리거 될 때 기본적으로 다시 한번 표시됩니다.

버전 3.7에서 변경: *BytesWarning*은 더는 기본 필터 목록에 나타나지 않으며 대신 -b가 두 번 지정되면 *sys.warnoptions*를 통해 구성됩니다.

기본 필터 재정의

파이썬으로 작성된 응용 프로그램 개발자는 기본적으로 사용자에게 모든 파이썬 수준 경고를 숨기고, 테스트를 실행하거나 달리 응용 프로그램에 대해 작업할 때만 표시하고 싶을 수 있습니다. 필터 구성을 인터프리터에 전달하는 데 사용되는 *sys.warnoptions* 어트리뷰트는 경고를 비활성화해야 하는지를 나타내는 마커로 사용할 수 있습니다:

```
import sys

if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

파이썬 코드용 테스트 실행기 개발자는 대신 다음과 같은 코드를 사용하여 테스트 대상 코드에 대해 기본적으로 모든 경고가 표시되도록 하는 것이 좋습니다:

```
import sys

if not sys.warnoptions:
    import os, warnings
    warnings.simplefilter("default") # 이 프로세스의 필터를 변경합니다
    os.environ["PYTHONWARNINGS"] = "default" # 서브 프로세스에도 영향을 줍니다
```

마지막으로, __main__ 이외의 이름 공간에서 사용자 코드를 실행하는 대화식 셸 개발자는 다음과 같은 코드를 사용하여 *DeprecationWarning* 메시지가 기본적으로 표시되도록 하는 것이 좋습니다 (여기서 user_ns는 대화식으로 입력된 코드를 실행하는 데 사용되는 모듈입니다):

```
import warnings
warnings.filterwarnings("default", category=DeprecationWarning,
                                   module=user_ns.get("__name__"))
```

29.6.3 일시적인 경고 억제

폐지된 함수처럼, 경고를 발생시킬 것을 알고 있는 코드를 사용하고 있지만, 경고를 보고 싶지 않으면 (명령줄을 통해 경고가 명시적으로 구성된 경우조차), *catch_warnings* 컨텍스트 관리자를 사용하여 경고를 억제할 수 있습니다:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

While within the context manager all warnings will simply be ignored. This allows you to use known-deprecated code without having to see the warning while not suppressing the warning for other code that might not be aware of its use of deprecated code.

참고

> See *Concurrent safety of Context Managers* for details on the concurrency-safety of the `catch_warnings` context manager when used in programs using multiple threads or async functions.

29.6.4 경고 테스트

코드가 발생시키는 경고를 테스트하려면, `catch_warnings` 컨텍스트 관리자를 사용하십시오. 이를 통해 쉽게 테스트할 수 있도록 경고 필터를 일시적으로 변경할 수 있습니다. 예를 들어, 검사할 모든 경고를 캡처하려면 다음을 수행하십시오:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

`always` 대신 `error`를 사용하여 모든 경고를 예외로 만들 수도 있습니다. 한 가지 알아야 할 것은 `once` / `default` 규칙으로 인해 경고가 이미 발생했으면, 어떤 필터가 설정되어 있더라도 경고와 관련된 경고 레지스트리가 지워지지 않으면 경고가 다시 표시되지 않는다는 것입니다.

Once the context manager exits, the warnings filter is restored to its state when the context was entered. This prevents tests from changing the warnings filter in unexpected ways between tests and leading to indeterminate test results.

> **참고**
>
> See *Concurrent safety of Context Managers* for details on the concurrency-safety of the `catch_warnings` context manager when used in programs using multiple threads or async functions.

같은 종류의 경고를 발생시키는 여러 작업을 테스트할 때, 각 작업이 새로운 경고를 발생시키는지 확인하는 방식으로 테스트하는 것이 중요합니다 (예를 들어 경고가 예외를 발생시키도록 설정하고 작업이 예외를 일으키는지 확인합니다, 각 작업 후에 경고 목록의 길이가 계속 증가하는지 확인합니다, 또는 각 새 작업 전에 경고 목록에서 이전 항목을 삭제합니다).

29.6.5 새 버전의 종속성에 대한 코드 갱신

(파이썬으로 작성된 응용 프로그램의 최종 사용자가 아닌) 파이썬 개발자가 주로 관심을 두는 경고 범주는 기본적으로 무시됩니다.

특히, 이 "기본적으로 무시됨" 목록에는 `DeprecationWarning`(`__main__`을 제외한 모든 모듈에서)가 포함되어 있습니다. 이는 개발자가 (표준 라이브러리와 제삼자 패키지 모두에서) 호환성을 깨는 향후 API 변경에 대한 시기적절한 알림을 받기 위해 일반적으로 무시되는 경고를 가시화해서 코드를 테스트해야 한다는 것을 의미합니다.

이상적인 경우, 코드에 적절한 테스트 스위트가 있고, 테스트 실행기는 테스트를 실행할 때 모든 경고를 묵시적으로 활성화합니다 (`unittest` 모듈에서 제공하는 테스트 실행기가 이렇게 합니다).

덜 이상적인 경우, `-Wd` 를 파이썬 인터프리터에 전달하거나 (`-W default`의 줄임 표현입니다), 환경에 `PYTHONWARNINGS=default`를 설정하여 응용프로그램이 폐지된 인터페이스를 사용하는지를 확인할 수

있습니다. 이를 통해 기본적으로 무시되는 경고를 포함한 모든 경고에 대한 default 처리가 가능합니다. 발생한 경고에 대해 수행할 조치를 변경하려면 `-W`로 전달되는 인자를 변경할 수 있습니다 (예를 들어 `-W error`). 어떤 것이 가능한지에 대한 자세한 내용은 `-W` 플래그를 참조하십시오.

29.6.6 사용 가능한 함수

`warnings.`**`warn`**(*message*, *category=None*, *stacklevel=1*, *source=None*, **, skip_file_prefixes=()*)

경고를 발행하거나, 무시하거나 예외를 발생시킵니다. 주어지면 *category* 인자는 경고 범주 클래스여야 합니다; 기본값은 `UserWarning`입니다. 또는, *message*가 `Warning` 인스턴스일 수 있으며, 이 경우 *category*는 무시되고 `message.__class__`가 사용됩니다. 이 경우, 메시지 텍스트는 `str(message)` 입니다. 이 함수는 발행된 특정 경고가 경고 필터에 의해 에러로 변경되면 예외를 발생시킵니다. *stacklevel* 인자는 다음과 같이 파이썬으로 작성된 래퍼 함수에서 사용할 수 있습니다:

```
def deprecated_api(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

이것은 경고가 `deprecated_api` 자체의 소스가 아닌 `deprecated_api`의 호출자를 참조하게 합니다 (전자는 경고 메시지의 목적을 무효로 하기 때문입니다).

The *skip_file_prefixes* keyword argument can be used to indicate which stack frames are ignored when counting stack levels. This can be useful when you want the warning to always appear at call sites outside of a package when a constant *stacklevel* does not fit all call paths or is otherwise challenging to maintain. If supplied, it must be a tuple of strings. When prefixes are supplied, stacklevel is implicitly overridden to be `max(2, stacklevel)`. To cause a warning to be attributed to the caller from outside of the current package you might write:

```
# example/lower.py
_warn_skips = (os.path.dirname(__file__),)

def one_way(r_luxury_yacht=None, t_wobbler_mangrove=None):
    if r_luxury_yacht:
        warnings.warn("Please migrate to t_wobbler_mangrove=.",
                      skip_file_prefixes=_warn_skips)

# example/higher.py
from . import lower

def another_way(**kw):
    lower.one_way(**kw)
```

This makes the warning refer to both the `example.lower.one_way()` and `example.higher.another_way()` call sites only from calling code living outside of `example` package.

제공되면, *source*는 `ResourceWarning`을 방출한 파괴된 객체입니다.

버전 3.6에서 변경: *source* 매개 변수를 추가했습니다.

버전 3.12에서 변경: Added *skip_file_prefixes*.

`warnings.`**`warn_explicit`**(*message*, *category*, *filename*, *lineno*, *module=None*, *registry=None*, *module_globals=None*, *source=None*)

이것은 `warn()`의 기능에 대한 저수준 인터페이스로서, 메시지, 범주, 파일명 및 줄 번호, 그리고 선택적으로 모듈 이름과 레지스트리(모듈의 `__warningregistry__` 딕셔너리이어야 합니다)를 명시적으로 전달합니다. 모듈 이름의 기본값은 `.py`가 제거된 파일명입니다; 레지스트리가 전달되지 않으면, 경고는 억제되지 않습니다. *message*는 문자열이고 *category*는 `Warning`의 서브 클래스여야 하고, 또는 *message*가 `Warning` 인스턴스일 수 있는데, 이 경우 *category*는 무시됩니다.

제공되면, *module_globals*는 경고가 발행되는 코드에서 사용 중인 전역 이름 공간이어야 합니다. (이 인자는 zip 파일이나 다른 파일 시스템이 아닌 임포트 소스에서 찾은 모듈의 소스 표시를 지원하는 데 사용됩니다).

제공되면, *source*는 `ResourceWarning`을 방출한 파괴된 객체입니다.

버전 3.6에서 변경: *source* 매개 변수를 추가합니다.

`warnings.showwarning`(*message*, *category*, *filename*, *lineno*, *file=None*, *line=None*)

파일에 경고를 기록합니다. 기본 구현은 `formatwarning(message, category, filename, lineno, line)`를 호출하고 결과 문자열을 *file*에 씁니다, *file*의 기본값은 `sys.stderr`입니다. `warnings.showwarning`에 대입하여 이 함수를 임의의 콜러블로 대체할 수 있습니다. *line*은 경고 메시지에 포함될 소스 코드 줄입니다; *line*이 제공되지 않으면, `showwarning()`은 *filename*과 *lineno*로 지정된 줄을 읽으려고 시도합니다.

`warnings.formatwarning`(*message*, *category*, *filename*, *lineno*, *line=None*)

표준 방식으로 경고를 포맷합니다. 내장된 개행 문자를 포함하고 개행 문자로 끝날 수 있는 문자열을 반환합니다. *line*은 경고 메시지에 포함될 소스 코드 줄입니다; *line*이 제공되지 않으면, `formatwarning()`은 *filename*과 *lineno*로 지정된 줄을 읽으려고 시도합니다.

`warnings.filterwarnings`(*action*, *message=''*, *category=Warning*, *module=''*, *lineno=0*, *append=False*)

경고 필터 명세 목록에 항목을 삽입합니다. 항목은 기본적으로 앞에 삽입됩니다; *append*가 참이면, 끝에 삽입됩니다. 인자의 형을 확인하고, *message*와 *module* 정규식을 컴파일한 후 경고 필터 목록에 튜플로 삽입합니다. 둘 다 특정 경고와 일치하면, 목록 앞쪽에 더 가까운 항목이 목록의 뒷부분에 있는 항목보다 우선합니다. 생략된 인자의 기본값은 모든 것과 일치하는 값입니다.

`warnings.simplefilter`(*action*, *category=Warning*, *lineno=0*, *append=False*)

경고 필터 명세 목록에 간단한 항목을 삽입합니다. 함수 매개 변수의 의미는 `filterwarnings()`와 같지만, 범주와 줄 번호가 일치하는 한 삽입 된 필터가 항상 모든 모듈의 메시지와 일치하기 때문에 정규식이 필요하지 않습니다.

`warnings.resetwarnings`()

경고 필터를 재설정합니다. 이는 `-W` 명령 줄 옵션과 `simplefilter()`에 대한 호출을 포함하여 `filterwarnings()`에 대한 모든 이전 호출의 영향을 되돌립니다.

`@warnings.deprecated`(*msg*, **, category=DeprecationWarning*, *stacklevel=1*)

Decorator to indicate that a class, function or overload is deprecated.

When this decorator is applied to an object, deprecation warnings may be emitted at runtime when the object is used. *static type checkers* will also generate a diagnostic on usage of the deprecated object.

사용법:

```
from warnings import deprecated
from typing import overload

@deprecated("Use B instead")
class A:
    pass

@deprecated("Use g instead")
def f():
    pass

@overload
@deprecated("int support is deprecated")
def g(x: int) -> int: ...
@overload
def g(x: str) -> int: ...
```

The warning specified by *category* will be emitted at runtime on use of deprecated objects. For functions, that happens on calls; for classes, on instantiation and on creation of subclasses. If the *category* is `None`, no warning is emitted at runtime. The *stacklevel* determines where the warning is emitted. If it is `1` (the default), the warning is emitted at the direct caller of the deprecated object; if it is higher, it is emitted further up the stack. Static type checker behavior is not affected by the *category* and *stacklevel* arguments.

The deprecation message passed to the decorator is saved in the `__deprecated__` attribute on the decorated object. If applied to an overload, the decorator must be after the `@overload` decorator for the attribute to exist on the overload as returned by `typing.get_overloads()`.

Added in version 3.13: See **PEP 702**.

29.6.7 사용 가능한 컨텍스트 관리자

class warnings.**catch_warnings**(*, *record=False*, *module=None*, *action=None*, *category=Warning*, *lineno=0*, *append=False*)

경고 필터와 `showwarning()` 함수를 복사하고 종료 시 복원하는 컨텍스트 관리자. *record* 인자가 `False`(기본값)이면 컨텍스트 관리자는 진입할 때 `None`을 반환합니다. *record*가 `True`이면, 재정의된 `showwarning()` 함수에 보이는 객체로 점진적으로 채워지는 리스트가 반환됩니다 (`sys.stdout`으로의 출력도 억제합니다). 리스트의 각 객체에는 `showwarning()`에 대한 인자와 이름이 같은 어트리뷰트가 있습니다.

module 인자는 필터가 보호되는 `warnings`를 임포트 할 때 반환되는 모듈 대신 사용되는 모듈을 취합니다. 이 인자는 주로 `warnings` 모듈 자체를 테스트하기 위해 존재합니다.

If the *action* argument is not `None`, the remaining arguments are passed to `simplefilter()` as if it were called immediately on entering the context.

See 경고 필터 for the meaning of the *category* and *lineno* parameters.

> **참고**
>
> See *Concurrent safety of Context Managers* for details on the concurrency-safety of the `catch_warnings` context manager when used in programs using multiple threads or async functions.

버전 3.11에서 변경: Added the *action*, *category*, *lineno*, and *append* parameters.

29.6.8 Concurrent safety of Context Managers

The behavior of `catch_warnings` context manager depends on the `sys.flags.context_aware_warnings` flag. If the flag is true, the context manager behaves in a concurrent-safe fashion and otherwise not. Concurrent-safe means that it is both thread-safe and safe to use within *asyncio coroutines* and tasks. Being thread-safe means that behavior is predictable in a multi-threaded program. The flag defaults to true for free-threaded builds and false otherwise.

If the `context_aware_warnings` flag is false, then `catch_warnings` will modify the global attributes of the `warnings` module. This is not safe if used within a concurrent program (using multiple threads or using asyncio coroutines). For example, if two or more threads use the `catch_warnings` class at the same time, the behavior is undefined.

If the flag is true, `catch_warnings` will not modify global attributes and will instead use a `ContextVar` to store the newly established warning filtering state. A context variable provides thread-local storage and it makes the use of `catch_warnings` thread-safe.

The *record* parameter of the context handler also behaves differently depending on the value of the flag. When *record* is true and the flag is false, the context manager works by replacing and then later restoring the module' s `showwarning()` function. That is not concurrent-safe.

When *record* is true and the flag is true, the `showwarning()` function is not replaced. Instead, the recording status is indicated by an internal property in the context variable. In this case, the `showwarning()` function will not be restored when exiting the context handler.

The `context_aware_warnings` flag can be set the `-X context_aware_warnings` command-line option or by the `PYTHON_CONTEXT_AWARE_WARNINGS` environment variable.

> **참고**
>
> It is likely that most programs that desire thread-safe behaviour of the warnings module will also want to set the `thread_inherit_context` flag to true. That flag causes threads created by `threading.Thread` to start with a copy of the context variables from the thread starting it. When true, the context established by `catch_warnings` in one thread will also apply to new threads started by it. If false, new threads will start with an empty warnings context variable, meaning that any filtering that was established by a `catch_warnings` context manager will no longer be active.

버전 3.14에서 변경: Added the `sys.flags.context_aware_warnings` flag and the use of a context variable for `catch_warnings` if the flag is true. Previous versions of Python acted as if the flag was always set to false.

29.7 dataclasses — 데이터 클래스

소스 코드: Lib/dataclasses.py

이 모듈은 `__init__()` 나 `__repr__()` 과 같은 생성된 특수 메서드 를 사용자 정의 클래스에 자동으로 추가하는 데코레이터와 함수를 제공합니다. 원래 **PEP 557** 에 설명되어 있습니다.

생성된 메서드에서 사용할 멤버 변수는 **PEP 526** 형 어노테이션을 사용하여 정의됩니다. 예를 들어, 이 코드는:

```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """인벤토리에 있는 항목을 추적하는 클래스."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

다른 것 중에서도, 다음과 같은 `__init__()` 를 추가합니다:

```
def __init__(self, name: str, unit_price: float, quantity_on_hand: int = 0):
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand
```

이 메서드는 클래스에 자동으로 추가됩니다: 위의 `InventoryItem` 정의에서 직접 지정되지는 않았습니다.

Added in version 3.7.

29.7.1 Module contents

`@dataclasses.`**`dataclass`**(*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*, *match_args=True*, *kw_only=False*, *slots=False*, *weakref_slot=False*)

이 함수는 (아래에서 설명하는) 생성된 특수 메서드를 클래스에 추가하는데 사용되는 데코레이터입니다.

`@dataclass` 데코레이터는 클래스를 검사하여 필드를 찾습니다. 필드는 형 어노테이션을 가진 클래스 변수로 정의됩니다. 아래에 설명된 두 가지 예외를 제외하고는, `@dataclass` 는 변수 어노테이션에 지정된 형을 검사하지 않습니다.

생성된 모든 메서드의 필드 순서는 클래스 정의에 나타나는 순서입니다.

@dataclass 데코레이터는 여러 "던더(dunder)" 메서드들을 클래스에 추가하는데, 아래에서 설명합니다. 추가할 메서드가 클래스에 이미 존재하면, 동작은 아래에 설명된 대로 매개변수에 따라 다릅니다. 데코레이터는 호출된 클래스와 같은 클래스를 반환합니다; 새 클래스가 만들어지지 않습니다.

@dataclass 가 매개변수 없는 단순한 데코레이터로 사용되면, 이 서명에 문서화 된 기본값들이 제공된 것처럼 행동합니다. 즉, 다음 @dataclass 의 세 가지 용법은 동등합니다:

```
@dataclass
class C:
    ...

@dataclass()
class C:
    ...

@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False,␣
↪frozen=False,
           match_args=True, kw_only=False, slots=False, weakref_slot=False)
class C:
    ...
```

@dataclass 의 매개변수는 다음과 같습니다:

- *init*: 참(기본값)이면, __init__() 메서드가 생성됩니다.

  클래스가 이미 __init__() 를 정의했으면, 이 매개변수는 무시됩니다.

- *repr*: 참(기본값)이면, __repr__() 메서드가 생성됩니다. 생성된 repr 문자열은 클래스 이름과 각 필드의 이름과 repr 을 갖습니다. 각 필드는 클래스에 정의된 순서대로 표시됩니다. repr 에서 제외하도록 표시된 필드는 포함되지 않습니다. 예를 들어: 예 :InventoryItem(name='widget', unit_price=3.0, quantity_on_hand=10).

  클래스가 이미 __repr__() 을 정의했으면, 이 매개변수는 무시됩니다.

- *eq*: 참(기본값)이면, __eq__() 메서드가 생성됩니다. 이 메서드는 클래스를 필드의 튜플인 것처럼 순서대로 비교합니다. 비교되는 두 인스턴스는 같은 형이어야 합니다.

  클래스가 이미 __eq__() 를 정의했으면, 이 매개변수는 무시됩니다.

- *order*: 참이면 (기본값은 False), __lt__(), __le__(), __gt__(), __ge__() 메서드가 생성됩니다. 이것들은 클래스를 필드의 튜플인 것처럼 순서대로 비교합니다. 비교되는 두 인스턴스는 같은 형이어야 합니다. *order* 가 참이고 *eq* 가 거짓이면 *ValueError* 가 발생합니다.

  클래스가 이미 __lt__(), __le__(), __gt__(), __ge__() 중 하나를 정의하고 있다면 *TypeError* 가 발생합니다.

- *unsafe_hash*: If true, force dataclasses to create a __hash__() method, even though it may not be safe to do so. Otherwise, generate a __hash__() method according to how *eq* and *frozen* are set. The default value is False.

  __hash__() 는 내장 *hash()* 에 의해 사용되며, 딕셔너리와 집합 같은 해시 컬렉션에 객체가 추가될 때 사용됩니다. __hash__() 를 갖는다는 것은 클래스의 인스턴스가 불변이라는 것을 의미합니다. 가변성은 프로그래머의 의도, __eq__() 의 존재와 행동, @dataclass 데코레이터의 *eq* 와 *frozen* 플래그의 값에 의존하는 복잡한 성질입니다.

  기본적으로, @dataclass 는 안전하지 않다면 __hash__() 메서드를 묵시적으로 추가하지 않습니다. 기존에 명시적으로 정의된 __hash__() 메서드를 추가하거나 변경하지도 않습니다. __hash__() 문서에서 설명된 대로, 클래스 어트리뷰트를 __hash__ = None 로 설정하는 것은 파이썬에 특별한 의미가 있습니다.

  __hash__() 가 명시적으로 정의되어 있지 않거나 None 으로 설정된 경우, @dataclass 는 묵시적 __hash__() 메서드를 추가할 수 있습니다. 권장하지는 않지만, unsafe_hash=True 로 @dataclass 가 __hash__() 메서드를 만들도록 강제할 수 있습니다. 이것은 당신의 클래스가 논리적으로 불변이지만, 여전히 변경될 수 있는 경우 일 수 있습니다. 이는 특수한 사용 사례이므로 신중하게 고려해야 합니다.

다음은 `__hash__()` 메서드의 묵시적 생성을 관장하는 규칙입니다. 데이터 클래스에 명시적 `__hash__()` 메서드를 가지면서 `unsafe_hash=True` 를 설정할 수는 없습니다; 그러면 `TypeError` 가 발생합니다.

eq 와 *frozen* 이 모두 참이면, 기본적으로 `@dataclass` 는 `__hash__()` 메서드를 만듭니다. *eq* 가 참이고 *frozen* 이 거짓이면, `__hash__()` 가 `None` 으로 설정되어 해시 불가능하다고 표시됩니다 (가변이기 때문입니다). 만약 *eq* 가 거짓이면, `__hash__()` 를 건드리지 않는데, 슈퍼 클래스의 `__hash__()` 가 사용된다는 뜻이 됩니다 (슈퍼 클래스가 `object` 라면, id 기반 해싱으로 돌아간다는 뜻입니다).

- *frozen*: If true (the default is `False`), assigning to fields will generate an exception. This emulates read-only frozen instances. See the *discussion* below.

  If `__setattr__()` or `__delattr__()` is defined in the class and *frozen* is true, then `TypeError` is raised.

- *match_args*: If true (the default is `True`), the `__match_args__` tuple will be created from the list of non keyword-only parameters to the generated `__init__()` method (even if `__init__()` is not generated, see above). If false, or if `__match_args__` is already defined in the class, then `__match_args__` will not be generated.

  Added in version 3.10.

- *kw_only*: If true (the default value is `False`), then all fields will be marked as keyword-only. If a field is marked as keyword-only, then the only effect is that the `__init__()` parameter generated from a keyword-only field must be specified with a keyword when `__init__()` is called. See the *parameter* glossary entry for details. Also see the `KW_ONLY` section.

  Keyword-only fields are not included in `__match_args__`.

  Added in version 3.10.

- *slots*: 참이면 (기본값은 `False`), `__slots__` 어트리뷰트가 만들어지고 원래 클래스 대신 새 클래스가 반환됩니다. `__slots__` 이 클래스에 이미 정의되어 있다면 `TypeError` 가 발생합니다.

  경고

  Passing parameters to a base class `__init_subclass__()` when using `slots=True` will result in a `TypeError`. Either use `__init_subclass__` with no parameters or use default values as a workaround. See gh-91126 for full details.

  Added in version 3.10.

  버전 3.11 에서 변경: If a field name is already included in the `__slots__` of a base class, it will not be included in the generated `__slots__` to prevent overriding them. Therefore, do not use `__slots__` to retrieve the field names of a dataclass. Use `fields()` instead. To be able to determine inherited slots, base class `__slots__` may be any iterable, but *not* an iterator.

- *weakref_slot*: If true (the default is `False`), add a slot named "__weakref__", which is required to make an instance `weakref-able`. It is an error to specify `weakref_slot=True` without also specifying `slots=True`.

  Added in version 3.11.

필드는 선택적으로 일반적인 파이썬 문법을 사용하여 기본값을 지정할 수 있습니다:

```
@dataclass
class C:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    a: int        # 'a' 에는 기본값이 없습니다
    b: int = 0    # 'b' 에 기본값을 대입합니다
```

이 예제에서, a 와 b 모두 추가된 __init__() 메서드에 포함되는데, 이런 식으로 정의됩니다:

```
def __init__(self, a: int, b: int = 0):
```

기본값이 없는 필드가 기본값이 있는 필드 뒤에 오는 경우 *TypeError* 가 발생합니다. 이것은 단일 클래스 내에서 발생하든, 클래스 상속의 결과로 발생하든 마찬가지입니다.

dataclasses.**field**(*, *default=MISSING*, *default_factory=MISSING*, *init=True*, *repr=True*, *hash=None*, *compare=True*, *metadata=None*, *kw_only=MISSING*, *doc=None*)

일반적이고 간단한 사용 사례의 경우 다른 기능은 필요하지 않습니다. 그러나 필드별로 추가 정보가 필요한 일부 데이터 클래스 기능이 있습니다. 추가 정보에 대한 필요성을 충족시키기 위해, 기본 필드 값을 제공된 field() 함수 호출로 바꿀 수 있습니다. 예를 들면:

```
@dataclass
class C:
    mylist: list[int] = field(default_factory=list)

c = C()
c.mylist += [1, 2, 3]
```

위에서 보인 것처럼, *MISSING* 값은 사용자가 일부 매개변수를 제공했는지를 탐지하는데 사용되는 표지 객체입니다. 일부 매개변수에 대해서는 None 이 별개의 의미를 갖는 유효한 값이기 때문에 이 표지가 사용됩니다. 어떤 코드도 *MISSING* 값을 직접 사용해서는 안 됩니다.

field() 의 매개변수는 다음과 같습니다:

- *default*: 제공되면, 이 필드의 기본값이 됩니다. 이것은 field() 호출 자체가 기본값의 정상 위치를 대체하기 때문에 필요합니다.
- *default_factory*: 제공되면, 이 필드의 기본값이 필요할 때 호출되는 인자가 없는 콜러블이어야 합니다. 여러 용도 중에서도, 이것은 아래에서 논의되는 것처럼 가변 기본값을 가진 필드를 지정하는 데 사용될 수 있습니다. *default* 와 *default_factory* 를 모두 지정하는 것은 에러입니다.
- *init*: 참(기본값)이면, 이 필드는 생성된 __init__() 메서드의 매개변수로 포함됩니다.
- *repr*: 참(기본값)이면, 이 필드는 생성된 __repr__() 메서드가 돌려주는 문자열에 포함됩니다.
- *hash*: 이것은 bool 또는 None 일 수 있습니다. 참이면, 이 필드는 생성된 __hash__() 메서드에 포함됩니다. 거짓이면, 이 필드는 생성된 __hash__() 에서 제외됩니다. None (기본값) 이면, *compare* 의 값을 사용합니다: 필드가 비교에 사용되면 해시에 포함되어야 하기 때문에, 이것은 일반적으로 기대되는 행동입니다. 이 값을 None 이외의 값으로 설정하는 것은 권장하지 않습니다.

  hash=False 이지만 compare=True 로 설정하는 한 가지 가능한 이유는, 동등 비교에 포함되는 필드가 해시값을 계산하는 데 비용이 많이 들고, 형의 해시값에 이바지하는 다른 필드가 있는 경우입니다. 필드가 해시에서 제외된 경우에도 비교에는 계속 사용됩니다.
- *compare*: 참(기본값)이면, 이 필드는 생성된 같음 및 비교 메서드(__eq__(), __gt__() 등)에 포함됩니다.
- *metadata*: 매핑이나 None 이 될 수 있습니다. None 은 빈 딕셔너리로 취급됩니다. 이 값은 *MappingProxyType()* 로 감싸져서 읽기 전용으로 만들어지고, *Field* 객체에 노출됩니다. 데이터 클래스에서는 전혀 사용되지 않으며, 제삼자 확장 메커니즘으로 제공됩니다. 여러 제삼자는 이름 공간으로 사용할 자신만의 키를 가질 수 있습니다.
- *kw_only*: If true, this field will be marked as keyword-only. This is used when the generated __init__() method' s parameters are computed.

  Keyword-only fields are also not included in __match_args__.

Added in version 3.10.

- *doc*: optional docstring for this field.

  Added in version 3.14.

필드의 기본값이 `field()` 호출로 지정되면, 이 필드의 클래스 어트리뷰트는 지정한 *default* 값으로 대체됩니다. *default* 가 제공되지 않으면 클래스 어트리뷰트는 삭제됩니다. 그 의도는, *@dataclass* 데코레이터 실행 후에, 기본값 자체가 지정된 것처럼 클래스 어트리뷰트가 모드 필드의 기본값을 갖도록 만드는 것입니다. 예를 들어, 이렇게 한 후에는:

```
@dataclass
class C:
    x: int
    y: int = field(repr=False)
    z: int = field(repr=False, default=10)
    t: int = 20
```

클래스 어트리뷰트 `C.z` 는 `10` 이 되고, 클래스 어트리뷰트 `C.t` 는 `20` 이 되고, 클래스 어트리뷰트 `C.x` 와 `C.y` 는 설정되지 않게 됩니다.

class dataclasses.**Field**

`Field` 객체는 정의된 각 필드를 설명합니다. 이 객체는 내부적으로 생성되며 *fields()* 모듈 수준 메서드(아래 참조)가 돌려줍니다. 사용자는 직접 `Field` 인스턴스 객체를 만들어서는 안 됩니다. 문서화 된 어트리뷰트는 다음과 같습니다:

- `name`: 필드의 이름.
- `type`: 필드의 형.
- `default`, `default_factory`, `init`, `repr`, `hash`, `compare`, `metadata` 및 `kw_only` 는 *field()* 함수에서와 같은 의미와 값을 가지고 있습니다.

다른 어트리뷰트도 있을 수 있지만, 내부적인 것이므로 검사하거나 의존해서는 안 됩니다.

class dataclasses.**InitVar**

`InitVar[T]` type annotations describe variables that are *init-only*. Fields annotated with `InitVar` are considered pseudo-fields, and thus are neither returned by the *fields()* function nor used in any way except adding them as parameters to `__init__()` and an optional *__post_init__()*.

dataclasses.**fields**(*class_or_instance*)

데이터 클래스의 필드들을 정의하는 *Field* 객체들의 튜플을 돌려줍니다. 데이터 클래스나 데이터 클래스의 인스턴스를 받아들입니다. 데이터 클래스 나 데이터 클래스의 인스턴스를 전달하지 않으면 *TypeError* 를 돌려줍니다. `ClassVar` 또는 `InitVar` 인 의사 필드는 반환하지 않습니다.

dataclasses.**asdict**(*obj*, *, *dict_factory=dict*)

데이터 클래스 *obj* 를 딕셔너리로 변환합니다 (팩토리 함수 *dict_factory* 를 사용합니다). 각 데이터 클래스는 각 필드를 `name: value` 쌍으로 갖는 딕셔너리로 변환됩니다. 데이터 클래스, 딕셔너리, 리스트 및 튜플은 재귀적으로 변환됩니다. 다른 객체들은 *copy.deepcopy()* 로 복사됩니다.

Example of using `asdict()` on nested dataclasses:

```
@dataclass
class Point:
     x: int
     y: int

@dataclass
class C:
     mylist: list[Point]
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
p = Point(10, 20)
assert asdict(p) == {'x': 10, 'y': 20}

c = C([Point(0, 0), Point(10, 4)])
assert asdict(c) == {'mylist': [{'x': 0, 'y': 0}, {'x': 10, 'y': 4}]}
```

To create a shallow copy, the following workaround may be used:

```
{field.name: getattr(obj, field.name) for field in fields(obj)}
```

`asdict()` 는 *obj* 가 데이터 클래스 인스턴스가 아닌 경우 `TypeError` 를 일으킵니다.

`dataclasses.`**`astuple`**(*obj*, **, tuple_factory=tuple*)

데이터 클래스 *obj* 를 튜플로 변환합니다 (팩토리 함수 *tuple_factory* 를 사용합니다). 각 데이터 클래스는 각 필드 값들의 튜플로 변환됩니다. 데이터 클래스, 딕셔너리, 리스트 및 튜플은 재귀적으로 변환됩니다. 다른 객체들은 `copy.deepcopy()` 로 복사됩니다.

이전 예에서 계속하면:

```
assert astuple(p) == (10, 20)
assert astuple(c) == ([(0, 0), (10, 4)],)
```

To create a shallow copy, the following workaround may be used:

```
tuple(getattr(obj, field.name) for field in dataclasses.fields(obj))
```

`astuple()` 은 *obj* 가 데이터 클래스 인스턴스가 아닌 경우 `TypeError` 를 일으킵니다.

`dataclasses.`**`make_dataclass`**(*cls_name*, *fields*, **, bases=()*, *namespace=None*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*, *match_args=True*, *kw_only=False*, *slots=False*, *weakref_slot=False*, *module=None*, *decorator=dataclass*)

새로운 데이터 클래스를 만드는데, 이름은 *cls_name* 이고, *fields* 에 정의된 필드들을 갖고, *bases* 에 주어진 베이스 클래스들을 갖고, *namespace* 로 주어진 이름 공간으로 초기화됩니다. *fields* 는 요소가 `name`, `(name, type)` 또는 `(name, type, Field)` 인 이터러블입니다. `name` 만 제공되면 `typing.Any` 가 `type` 으로 사용됩니다. *init*, *repr*, *eq*, *order*, *unsafe_hash*, *frozen*, *match_args*, *kw_only*, *slots* 및 *weakref_slot* 의 값은 `@dataclass` 에서와 같은 의미가 있습니다.

If *module* is defined, the `__module__` attribute of the dataclass is set to that value. By default, it is set to the module name of the caller.

The *decorator* parameter is a callable that will be used to create the dataclass. It should take the class object as a first argument and the same keyword arguments as `@dataclass`. By default, the `@dataclass` function is used.

This function is not strictly required, because any Python mechanism for creating a new class with `__annotations__` can then apply the `@dataclass` function to convert that class to a dataclass. This function is provided as a convenience. For example:

```
C = make_dataclass('C',
                   [('x', int),
                     'y',
                    ('z', int, field(default=5))],
                   namespace={'add_one': lambda self: self.x + 1})
```

는 다음과 동등합니다:

```
@dataclass
class C:
    x: int
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    y: 'typing.Any'
    z: int = 5

    def add_one(self):
        return self.x + 1
```

Added in version 3.14: Added the *decorator* parameter.

dataclasses.**replace**(*obj*, /, ***changes*)

obj 와 같은 형의 새 객체를 만드는데, 필드를 *changes* 의 값들로 대체합니다. *obj* 가 데이터 클래스가 아니라면 *TypeError* 를 발생시킵니다. *changes* 의 키가 주어진 데이터 클래스의 필드 이름이 아니면, *TypeError* 를 발생시킵니다.

새로 반환된 객체는 데이터 클래스의 `__init__()` 메서드를 호출하여 생성됩니다. 이렇게 함으로써 (있는 경우) *__post_init__()* 의 호출을 보장합니다.

기본값을 가지지 않는 초기화 전용 변수가 존재한다면, `replace()` 호출에 반드시 지정해서 `__init__()` 와 *__post_init__()* 에 전달 될 수 있도록 해야 합니다.

changes 가 `init=False` 를 갖는 것으로 정의된 필드를 포함하는 것은 에러입니다. 이 경우 *ValueError* 가 발생합니다.

`replace()` 를 호출하는 동안 `init=False` 필드가 어떻게 작동하는지 미리 경고합니다. 그것들은 소스 객체로부터 복사되는 것이 아니라, (초기화되기는 한다면) *__post_init__()* 에서 초기화됩니다. `init=False` 필드는 거의 사용되지 않으리라고 예상합니다. 사용된다면, 대체 클래스 생성자를 사용하거나, 인스턴스 복사를 처리하는 사용자 정의 `replace()` (또는 비슷하게 이름 지어진) 메서드를 사용하는 것이 좋을 것입니다.

Dataclass instances are also supported by generic function *copy.replace()*.

dataclasses.**is_dataclass**(*obj*)

매개변수가 데이터 클래스(데이터 클래스의 서브클래스 포함)나 그의 인스턴스면 `True` 를 반환하고, 그렇지 않으면 `False` 를 반환합니다.

(데이터 클래스 자체가 아니라) 데이터 클래스의 인스턴스인지 알아야 한다면 `not isinstance(obj, type)` 검사를 추가하십시오:

```
def is_dataclass_instance(obj):
    return is_dataclass(obj) and not isinstance(obj, type)
```

dataclasses.**MISSING**

A sentinel value signifying a missing default or default_factory.

dataclasses.**KW_ONLY**

A sentinel value used as a type annotation. Any fields after a pseudo-field with the type of `KW_ONLY` are marked as keyword-only fields. Note that a pseudo-field of type `KW_ONLY` is otherwise completely ignored. This includes the name of such a field. By convention, a name of _ is used for a `KW_ONLY` field. Keyword-only fields signify `__init__()` parameters that must be specified as keywords when the class is instantiated.

In this example, the fields `y` and `z` will be marked as keyword-only fields:

```
@dataclass
class Point:
    x: float
    _: KW_ONLY
    y: float
    z: float

p = Point(0, y=1.5, z=2.0)
```

In a single dataclass, it is an error to specify more than one field whose type is `KW_ONLY`.

Added in version 3.10.

exception dataclasses.**FrozenInstanceError**

frozen=True 로 정의된 데이터 클래스에서 묵시적으로 정의된 __setattr__() 또는 __delattr__() 이 호출 될 때 발생합니다. *AttributeError*의 서브 클래스입니다.

29.7.2 초기화 후처리

dataclasses.**__post_init__**()

클래스에 정의된 경우, 생성된 __init__() 에 의해 호출되며, 일반적으로 self.__post_init__() 형태로 호출됩니다. 그러나, InitVar 필드가 정의되어 있으면, 클래스에 정의된 순서대로 __post_init__() 로 전달됩니다. __init__() 메서드가 생성되지 않으면, __post_init__() 가 자동으로 호출되지 않습니다.

다른 용도 중에서도, 하나나 그 이상의 다른 필드에 의존하는 필드 값을 초기화하는데 사용할 수 있습니다. 예를 들면:

```
@dataclass
class C:
    a: float
    b: float
    c: float = field(init=False)

    def __post_init__(self):
        self.c = self.a + self.b
```

The __init__() method generated by *@dataclass* does not call base class __init__() methods. If the base class has an __init__() method that has to be called, it is common to call this method in a *__post_init__()* method:

```
class Rectangle:
    def __init__(self, height, width):
        self.height = height
        self.width = width

@dataclass
class Square(Rectangle):
    side: float

    def __post_init__(self):
        super().__init__(self.side, self.side)
```

Note, however, that in general the dataclass-generated __init__() methods don' t need to be called, since the derived dataclass will take care of initializing all fields of any base class that is a dataclass itself.

매개변수를 __post_init__() 에 전달하는 방법은 초기화 전용 변수에 대한 아래 섹션을 참조하십시오. 또한 *replace()* 가 init=False 필드를 처리하는 방식에 관한 경고를 보십시오.

29.7.3 클래스 변수

@dataclass 가 실제로 필드의 형을 검사하는 몇 안 되는 곳 중 하나는 필드가 **PEP 526** 에 정의된 클래스 변수인지를 확인하는 것입니다. 필드의 형이 *typing.ClassVar* 인지 검사합니다. 필드가 ClassVar 인 경우, 필드로 취급되지 않고 데이터 클래스 메커니즘에서 무시됩니다. 이런 ClassVar 의사 필드는 모듈 수준 *fields()* 함수에 의해 반환되지 않습니다.

29.7.4 초기화 전용 변수

@dataclass 가 형 어노테이션을 검사하는 다른 곳은 필드가 초기화 전용 변수인지 확인하는 것입니다. 필드의 형이 *InitVar* 인지 검사합니다. 필드가 *InitVar* 인 경우, 초기화 전용 변수라고 불리는 의사 필드로 간주합니다. 실제 필드가 아니므로, 모듈 수준 *fields()* 함수에 의해 반환되지 않습니다. 초기화 전용 필드는 생성된 `__init__()` 메서드의 매개변수로 추가되며, 선택적인 *__post_init__()* 메서드로 전달됩니다. 이 외에 데이터 클래스에서 사용되는 곳은 없습니다.

예를 들어, 클래스를 만들 때 값이 제공되지 않으면, 필드가 데이터베이스로부터 초기화된다고 가정합시다:

```
@dataclass
class C:
    i: int
    j: int | None = None
    database: InitVar[DatabaseType | None] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

이 경우, *fields()* 는 `i` 와 `j` 를 위한 *Field* 객체를 반환하지만, `database` 는 반환하지 않습니다.

29.7.5 고정 인스턴스

정말로 불변인 파이썬 객체를 만드는 것은 불가능합니다. 그러나, `frozen=True` 를 *@dataclass* 데코레이터에 전달함으로써 불변성을 흉내 낼 수 있습니다. 이 경우, 데이터 클래스는 `__setattr__()` 과 `__delattr__()` 메서드를 클래스에 추가합니다. 이 메서드는 호출될 때 *FrozenInstanceError* 를 발생시킵니다.

`frozen=True` 를 사용할 때 약간의 성능 저하가 있습니다: `__init__()` 는 필드를 초기화하는데 간단한 대입을 사용할 수 없고, `object.__setattr__()` 을 사용해야 합니다.

29.7.6 계승

데이터 클래스가 *@dataclass* 데코레이터에 의해 생성될 때, 클래스의 모든 베이스 클래스들을 MRO 역순 (즉, *object* 에서 시작해서)으로 조사하고, 발견되는 데이터 클래스마다 그 베이스 클래스의 필드들을 순서 있는 필드 매핑에 추가합니다. 모든 생성된 메서드들은 이 합쳐지고 계산된 순서 있는 필드 매핑을 사용합니다. 필드들이 삽입 순서이기 때문에, 파생 클래스는 베이스 클래스를 재정의합니다. 예:

```
@dataclass
class Base:
    x: Any = 15.0
    y: int = 0

@dataclass
class C(Base):
    z: int = 10
    x: int = 15
```

필드의 최종 목록은 순서대로 `x`, `y`, `z` 입니다. `x` 의 최종 형은 클래스 `C` 에서 지정된 *int* 입니다.

생성된 `C` 의 `__init__()` 메서드는 이렇게 됩니다:

```
def __init__(self, x: int = 15, y: int = 0, z: int = 10):
```

29.7.7 Re-ordering of keyword-only parameters in __init__()

After the parameters needed for __init__() are computed, any keyword-only parameters are moved to come after all regular (non-keyword-only) parameters. This is a requirement of how keyword-only parameters are implemented in Python: they must come after non-keyword-only parameters.

In this example, Base.y, Base.w, and D.t are keyword-only fields, and Base.x and D.z are regular fields:

```
@dataclass
class Base:
    x: Any = 15.0
    _: KW_ONLY
    y: int = 0
    w: int = 1

@dataclass
class D(Base):
    z: int = 10
    t: int = field(kw_only=True, default=0)
```

생성된 D 의 __init__() 메서드는 이렇게 됩니다:

```
def __init__(self, x: Any = 15.0, z: int = 10, *, y: int = 0, w: int = 1, t: int =␣
↪0):
```

Note that the parameters have been re-ordered from how they appear in the list of fields: parameters derived from regular fields are followed by parameters derived from keyword-only fields.

The relative ordering of keyword-only parameters is maintained in the re-ordered __init__() parameter list.

29.7.8 기본 팩토리 함수

field() 가 *default_factory* 를 지정하면, 필드의 기본값이 필요할 때 인자 없이 호출됩니다. 예를 들어, 리스트의 새 인스턴스를 만들려면, 이렇게 하세요:

```
mylist: list = field(default_factory=list)
```

필드가 (init=False 를 사용해서) __init__() 에서 제외되고, 그 필드가 *default_factory* 를 지정하면, 생성된 __init__() 함수는 항상 기본 팩토리 함수를 호출합니다. 이는 필드에 초기화 값을 제공할 수 있는 다른 방법이 없기 때문입니다.

29.7.9 가변 기본값

파이썬은 기본 멤버 변숫값을 클래스 어트리뷰트에 저장합니다. 데이터 클래스를 사용하지 않는 이 예제를 생각해보세요:

```
class C:
    x = []
    def add(self, element):
        self.x.append(element)

o1 = C()
o2 = C()
o1.add(1)
o2.add(2)
assert o1.x == [1, 2]
assert o1.x is o2.x
```

클래스 C 의 두 인스턴스는 예상대로 같은 클래스 변수 x 를 공유합니다.

데이터 클래스를 사용해서, 만약 이 코드가 올바르다면:

```
@dataclass
class D:
    x: list = []       # 이 코드는 ValueError 를 발생시킵니다
    def add(self, element):
        self.x.append(element)
```

비슷한 코드를 생성합니다:

```
class D:
    x = []
    def __init__(self, x=x):
        self.x = x
    def add(self, element):
        self.x.append(element)

assert D().x is D().x
```

이것은 클래스 `C` 를 사용한 원래 예제와 같은 문제를 가지고 있습니다. 즉, 클래스 인스턴스를 만들 때 `x` 에 대한 값을 지정하지 않는 클래스 `D` 의 두 인스턴스는 `x`의 같은 사본을 공유합니다. 데이터 클래스는 일반적인 파이썬 클래스 생성을 사용하기 때문에, 이 동작 역시 공유합니다. 데이터 클래스가 이 조건을 감지하는 일반적인 방법은 없습니다. 대신, *@dataclass* 데코레이터는 해시 불가능한 기본 매개변수를 탐지하면 *ValueError* 를 발생시킵니다. 값이 해시 불가능하면, 가변이라는 가정입니다. 이것은 부분적인 해결책이지만, 흔한 오류로부터 보호합니다.

기본 팩토리 함수를 사용하면 필드의 기본값으로 가변형의 새 인스턴스를 만들 수 있습니다:

```
@dataclass
class D:
    x: list = field(default_factory=list)

assert D().x is not D().x
```

버전 3.11 에서 변경: Instead of looking for and disallowing objects of type *list*, *dict*, or *set*, unhashable objects are now not allowed as default values. Unhashability is used to approximate mutability.

29.7.10 Descriptor-typed fields

Fields that are assigned descriptor objects as their default value have the following special behaviors:

- The value for the field passed to the dataclass' s `__init__()` method is passed to the descriptor' s `__set__()` method rather than overwriting the descriptor object.
- Similarly, when getting or setting the field, the descriptor' s `__get__()` or `__set__()` method is called rather than returning or overwriting the descriptor object.
- To determine whether a field contains a default value, *@dataclass* will call the descriptor' s `__get__()` method using its class access form: `descriptor.__get__(obj=None, type=cls)`. If the descriptor returns a value in this case, it will be used as the field' s default. On the other hand, if the descriptor raises *AttributeError* in this situation, no default value will be provided for the field.

```
class IntConversionDescriptor:
    def __init__(self, *, default):
        self._default = default

    def __set_name__(self, owner, name):
        self._name = "_" + name

    def __get__(self, obj, type):
        if obj is None:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
            return self._default

        return getattr(obj, self._name, self._default)

    def __set__(self, obj, value):
        setattr(obj, self._name, int(value))

@dataclass
class InventoryItem:
    quantity_on_hand: IntConversionDescriptor =␣
↪IntConversionDescriptor(default=100)

i = InventoryItem()
print(i.quantity_on_hand)    # 100
i.quantity_on_hand = 2.5     # __set__ 을 2.5 로 호출합니다
print(i.quantity_on_hand)    # 2
```

Note that if a field is annotated with a descriptor type, but is not assigned a descriptor object as its default value, the field will act like a normal field.

29.8 contextlib — with 문 컨텍스트를 위한 유틸리티

소스 코드: Lib/contextlib.py

이 모듈은 `with` 문이 수반되는 일반적인 작업을 위한 유틸리티를 제공합니다. 자세한 정보는 컨텍스트 관리자 형 과 context-managers도 참조하십시오.

29.8.1 유틸리티

제공되는 함수와 클래스:

class contextlib.**AbstractContextManager**

`object.__enter__()` 와 `object.__exit__()` 를 구현하는 클래스의 추상 베이스 클래스. `self`를 반환하는 `object.__enter__()` 의 기본 구현이 제공되지만 `object.__exit__()` 는 기본적으로 `None` 을 반환하는 추상 메서드입니다. 컨텍스트 관리자 형 의 정의도 참조하십시오.

Added in version 3.6.

class contextlib.**AbstractAsyncContextManager**

`object.__aenter__()` 와 `object.__aexit__()` 를 구현하는 클래스의 추상 베이스 클래스. `self`를 반환하는 `object.__aenter__()` 의 기본 구현이 제공되지만 `object.__aexit__()` 는 기본적으로 `None`을 반환하는 추상 메서드입니다. async-context-managers의 정의도 참조하십시오.

Added in version 3.7.

@contextlib.**contextmanager**

이 함수는 클래스나 별도의 `__enter__()` 와 `__exit__()` 메서드를 작성할 필요 없이, `with` 문 컨텍스트 관리자를 위한 팩토리 함수를 정의하는 데 사용할 수 있는 데코레이터입니다.

많은 객체가 자체적으로 with 문에서의 사용을 지원하지만, 스스로는 컨텍스트 관리자가 아니고 `contextlib.closing`과 함께 사용할 `close()` 메서드를 구현하지 않는 자원을 관리해야 하는 경우가 있습니다.

올바른 자원 관리를 보장하기 위한 추상적인 예는 다음과 같습니다:

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwds):
    # 자원을 얻는 코드, 예를 들어:
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # 자원을 해제하는 코드, 예를 들어:
        release_resource(resource)
```

The function can then be used like this:

```
>>> with managed_resource(timeout=3600) as resource:
...     # 블록의 코드에서 예외가 발생하더라도,
...     # 이 블록의 끝에서 자원이 해제됩니다
```

데코레이트 되는 함수는 호출될 때 제너레이터 이터레이터를 반환해야 합니다. 이 이터레이터는 정확히 하나의 값을 산출해야 하며, 이는 `with` 문의 `as` 절에 있는 대상에 연결됩니다 (있다면).

제너레이터가 산출하는 지점에서, `with` 문에 중첩된 블록이 실행됩니다. 그런 다음 블록이 종료된 후 제너레이터가 재개합니다. 블록에서 처리되지 않은 예외가 발생하면, 제너레이터 내부의 yield 가 등장한 지점에서 다시 발생합니다. 따라서, `try…except…finally` 문을 사용하여 에러(있다면)를 잡거나, 어떤 정리가 수행되도록 할 수 있습니다. 예외를 단지 로그 하기 위해 또는 (예외를 완전히 억제하는 대신) 어떤 작업을 수행하기 위해 예외를 잡았다면 제너레이터는 해당 예외를 다시 발생시켜야 합니다. 그렇지 않으면 제너레이터 컨텍스트 관리자가 `with` 문에 예외가 처리되었음을 표시하고, `with` 문 바로 다음에 오는 문장으로 실행이 재개됩니다.

*contextmanager()*는 *ContextDecorator*를 사용해서, 만들어진 컨텍스트 관리자를 `with` 문뿐만 아니라 데코레이터로 사용할 수 있습니다. 데코레이터로 사용될 때, 새로운 제너레이터 인스턴스는 각 함수 호출마다 묵시적으로 만들어집니다 (이는 그렇지 않을 경우 *contextmanager()*에 의해 만들어진 "일회성" 컨텍스트 관리자가 데코레이터로 사용하기 위해 컨텍스트 관리자가 여러 번의 호출을 지원해야 한다는 요구 사항을 충족시킬 수 있도록 합니다).

버전 3.2에서 변경: *ContextDecorator* 사용.

`@contextlib.`**`asynccontextmanager`**

*contextmanager()*와 유사하지만, 비동기 컨텍스트 관리자를 만듭니다.

이 함수는 클래스나 별도의 `__aenter__()` 와 `__aexit__()` 메서드를 작성할 필요 없이, `async with` 문 비동기 컨텍스트 관리자를 위한 팩토리 함수를 정의하는 데 사용할 수 있는 데코레이터입니다. 비동기 제너레이터 함수에 적용해야 합니다.

간단한 예:

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def get_connection():
    conn = await acquire_db_connection()
    try:
        yield conn
    finally:
        await release_db_connection(conn)

async def get_all_users():
    async with get_connection() as conn:
        return conn.query('SELECT ...')
```

Added in version 3.7.

Context managers defined with *asynccontextmanager()* can be used either as decorators or with `async with` statements:

```
import time
from contextlib import asynccontextmanager

@asynccontextmanager
async def timeit():
    now = time.monotonic()
    try:
        yield
    finally:
        print(f'it took {time.monotonic() - now}s to run')

@timeit()
async def main():
    # ... 비동기 코드 ...
```

데코레이터로 사용될 때, 새로운 제너레이터 인스턴스는 각 함수 호출마다 묵시적으로 만들어집니다. 이는 그렇지 않을 경우 *contextmanager()* 에 의해 만들어진 "일회성" 컨텍스트 관리자가 데코레이터로 사용하기 위해 컨텍스트 관리자가 여러 번의 호출을 지원해야 한다는 요구 사항을 충족시킬 수 있도록 합니다.

버전 3.10에서 변경: Async context managers created with *asynccontextmanager()* can be used as decorators.

`contextlib.`**`closing`**(*thing*)

블록이 완료될 때 *thing*을 닫는 컨텍스트 관리자를 반환합니다. 이것은 기본적으로 다음과 동등합니다:

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

그리고 다음과 같은 코드를 작성할 수 있도록 합니다:

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('https://www.python.org')) as page:
    for line in page:
        print(line)
```

`page`를 명시적으로 닫을 필요가 없습니다. 에러가 발생하더라도, `with` 블록이 종료될 때 `page.close()` 가 호출됩니다.

> **참고**
>
> Most types managing resources support the *context manager* protocol, which closes *thing* on leaving the `with` statement. As such, `closing()` is most useful for third party types that don' t support context managers. This example is purely for illustration purposes, as *urlopen()* would normally be used in a context manager.

contextlib.**aclosing**(*thing*)

블록이 완료될 때 *thing*의 `aclose()` 메서드를 호출하는 비동기 컨텍스트 관리자를 반환합니다. 이것은 기본적으로 다음과 동등합니다:

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def aclosing(thing):
    try:
        yield thing
    finally:
        await thing.aclose()
```

Significantly, `aclosing()` supports deterministic cleanup of async generators when they happen to exit early by `break` or an exception. For example:

```
from contextlib import aclosing

async with aclosing(my_generator()) as values:
    async for value in values:
        if value == 42:
            break
```

This pattern ensures that the generator' s async exit code is executed in the same context as its iterations (so that exceptions and context variables work as expected, and the exit code isn' t run after the lifetime of some task it depends on).

Added in version 3.10.

contextlib.**nullcontext**(*enter_result=None*)

`__enter__`에서 *enter_result*를 반환하지만, 그 외에는 아무것도 하지 않는 컨텍스트 관리자를 반환합니다. 선택적 컨텍스트 관리자를 위한 대체로 사용하기 위한 것입니다, 예를 들어:

```
def myfunction(arg, ignore_exceptions=False):
    if ignore_exceptions:
        # 모든 예외를 무시하기 위해 suppress를 사용합니다.
        cm = contextlib.suppress(Exception)
    else:
        # 아무런 예외도 무시하지 않습니다, cm은 아무 효과도 없습니다.
        cm = contextlib.nullcontext()
    with cm:
        # 뭔가 합니다
```

*enter_result*를 사용하는 예:

```
def process_file(file_or_path):
    if isinstance(file_or_path, str):
        # 문자열이면, 파일을 엽니다
        cm = open(file_or_path)
    else:
        # 파일을 닫는 책임은 호출자에게 있습니다
        cm = nullcontext(file_or_path)

    with cm as file:
        # 파일에 대한 처리를 수행합니다
```

또한 비동기 컨텍스트 관리자의 대체로 사용할 수 있습니다:

```
async def send_http(session=None):
    if not session:
        # http 세션이 없으면, aiohttp 로 만듭니다
        cm = aiohttp.ClientSession()
    else:
        # 세션을 닫는 책임은 호출자에게 있습니다
        cm = nullcontext(session)

    async with cm as session:
        # 세션으로 http 요청을 보냅니다
```

Added in version 3.7.

버전 3.10에서 변경: *asynchronous context manager* support was added.

contextlib.**suppress**(**exceptions*)

`with` 문의 본문에서 발생하는 지정된 예외를 모두 억제한 후 `with` 문의 끝 다음에 오는 첫 번째 문장에서 실행을 재개하는 컨텍스트 관리자를 반환합니다.

예외를 완전히 억제하는 다른 메커니즘과 마찬가지로, 이 컨텍스트 관리자는 프로그램 실행을 조용히 계속하는 것이 옳은 것으로 알려진 매우 특정한 에러를 다루기 위해서만 사용해야 합니다.

예를 들면:

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')

with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

이 코드는 다음과 동등합니다:

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
    pass

try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

이 컨텍스트 관리자는 재진입 가능합니다.

If the code within the `with` block raises a `BaseExceptionGroup`, suppressed exceptions are removed from the group. Any exceptions of the group which are not suppressed are re-raised in a new group which is created using the original group' s `derive()` method.

Added in version 3.4.

버전 3.12에서 변경: `suppress` now supports suppressing exceptions raised as part of a `BaseExceptionGroup`.

contextlib.**redirect_stdout**(*new_target*)

`sys.stdout`을 다른 파일이나 파일류 객체로 일시적으로 리디렉션 하기 위한 컨텍스트 관리자.

이 도구는 출력이 stdout에 배선된 기존 함수나 클래스에 유연성을 추가합니다.

예를 들어, `help()`의 출력은 일반적으로 *sys.stdout*으로 전송됩니다. 출력을 `io.StringIO` 객체로 리디렉션 하여 해당 출력을 문자열로 캡처 할 수 있습니다. 대체 스트림이 `__enter__` 메서드에서 반환되어 `with` 문의 대상으로 사용할 수 있습니다:

```
with redirect_stdout(io.StringIO()) as f:
    help(pow)
s = f.getvalue()
```

*help()*의 출력을 디스크의 파일로 보내려면, 출력을 일반 파일로 리디렉션 하십시오:

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

*help()*의 출력을 *sys.stderr*로 보내려면:

```
with redirect_stdout(sys.stderr):
    help(pow)
```

*sys.stdout*에 대한 전역 부작용은 이 컨텍스트 관리자가 라이브러리 코드와 대부분의 스레드 응용 프로그램에 사용하기에 적합하지 않음을 의미합니다. 또한 서브 프로세스의 출력에는 영향을 미치지 않습니다. 그러나, 여전히 많은 유틸리티 스크립트에 유용한 접근법입니다.

이 컨텍스트 관리자는 재진입 가능합니다.

Added in version 3.4.

contextlib.**redirect_stderr**(*new_target*)

*redirect_stdout()*과 유사하지만, *sys.stderr*를 다른 파일이나 파일류 객체로 리디렉션 합니다.

이 컨텍스트 관리자는 재진입 가능합니다.

Added in version 3.5.

contextlib.**chdir**(*path*)

Non parallel-safe context manager to change the current working directory. As this changes a global state, the working directory, it is not suitable for use in most threaded or async contexts. It is also not suitable for most non-linear code execution, like generators, where the program execution is temporarily relinquished – unless explicitly desired, you should not yield when this context manager is active.

This is a simple wrapper around *chdir()*, it changes the current working directory upon entering and restores the old one on exit.

이 컨텍스트 관리자는 재진입 가능합니다.

Added in version 3.11.

class contextlib.**ContextDecorator**

컨텍스트 관리자를 데코레이터로도 사용할 수 있도록 하는 베이스 클래스.

ContextDecorator를 상속하는 컨텍스트 관리자는 일반적인 방식으로 __enter__와 __exit__를 구현해야 합니다. __exit__는 데코레이터로 사용될 때도 선택적 예외 처리를 유지합니다.

ContextDecorator는 *contextmanager()*에서 사용되므로, 이 기능을 자동으로 얻게 됩니다.

ContextDecorator의 예:

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False
```

The class can then be used like this:

```
>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
>>> function()
Starting
The bit in the middle
Finishing

>>> with mycontext():
...     print('The bit in the middle')
...
Starting
The bit in the middle
Finishing
```

이 변경은 단지 다음 형식의 구성에 대한 편의 문법일 뿐입니다:

```
def f():
    with cm():
        # 뭔가 합니다
```

`ContextDecorator`는 대신 다음과 같이 쓸 수 있도록 합니다:

```
@cm()
def f():
    # 뭔가 합니다
```

`cm`이 단지 함수의 일부가 아니라 전체 함수에 적용된다는 것을 분명히 합니다 (그리고 들여쓰기 수준을 절약하는 것도 좋습니다).

베이스 클래스가 이미 있는 기존 컨텍스트 관리자는 `ContextDecorator`를 믹스인 클래스로 사용하여 확장할 수 있습니다:

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False
```

> **참고**
>
> 데코레이트 된 함수는 여러 번 호출될 수 있어야 해서, 하부 컨텍스트 관리자는 여러 `with` 문에서의 사용을 지원해야 합니다. 그렇지 않으면, 함수 내에 명시적인 `with` 문이 있는 원래 구문을 사용해야 합니다.

Added in version 3.2.

class contextlib.**AsyncContextDecorator**

*`ContextDecorator`*와 유사하지만, 비동기 함수 전용입니다.

`AsyncContextDecorator`의 예:

```
from asyncio import run
from contextlib import AsyncContextDecorator

class mycontext(AsyncContextDecorator):
    async def __aenter__(self):
        print('Starting')
        return self

    async def __aexit__(self, *exc):
        print('Finishing')
        return False
```

The class can then be used like this:

```
>>> @mycontext()
... async def function():
...     print('The bit in the middle')
...
>>> run(function())
Starting
The bit in the middle
Finishing

>>> async def function():
...    async with mycontext():
...         print('The bit in the middle')
...
>>> run(function())
Starting
The bit in the middle
Finishing
```

Added in version 3.10.

class contextlib.**ExitStack**

다른 컨텍스트 관리자와 정리 함수, 특히 입력 데이터에 의해 선택적이거나 다른 방식으로 구동되는 것들을 프로그래밍 방식으로 쉽게 결합할 수 있도록 설계된 컨텍스트 관리자.

예를 들어, 일련의 파일을 다음과 같이 단일 with 문으로 쉽게 처리할 수 있습니다:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # 목록의 뒤에 나오는 파일을 열려는 시도가
    # 예외를 발생시키더라도, with 문의 끝에서
    # 열린 모든 파일이 자동으로 닫힙니다.
```

The `__enter__()` method returns the *ExitStack* instance, and performs no additional operations.

각 인스턴스는 인스턴스가 닫힐 때 (명시적으로 혹은 `with` 문의 끝에서 묵시적으로) 역순으로 호출되는 등록된 콜백의 스택을 유지합니다. 컨텍스트 스택 인스턴스가 가비지 수집될 때 콜백이 묵시적으로 호출되지 않음에 유의하십시오.

이 스택 모델은 `__init__` 메서드에서 자원을 확보하는 컨텍스트 관리자(가령 파일 객체)가 올바르게 처리될 수 있도록 사용됩니다.

등록된 콜백이 등록 순서의 역순으로 호출되므로, 여러 개의 중첩된 `with` 문이 등록된 콜백 집합과 함께 사용된 것처럼 작동합니다. 이것은 예외 처리로도 확장됩니다 - 내부 콜백이 예외를 억제하거나 바꾸면, 외부 콜백에는 갱신된 상태에 따른 인자가 전달됩니다.

탈출 콜백의 스택을 올바르게 되감는 세부 사항을 다루는 비교적 저수준의 API입니다. 응용 프로그램 특정 방식으로 탈출 스택을 조작하는 고수준 컨텍스트 관리자에게 적절한 기반을 제공합니다.

Added in version 3.3.

enter_context(*cm*)

새로운 컨텍스트 관리자에 진입하고 `__exit__()` 메서드를 콜백 스택에 추가합니다. 반환 값은 컨텍스트 관리자 고유의 `__enter__()` 메서드의 결과입니다.

이러한 컨텍스트 관리자는 `with` 문의 일부로 직접 사용되었다면 일반적으로 했을 것과 마찬가지로 예외를 억제할 수 있습니다.

버전 3.11에서 변경: Raises `TypeError` instead of `AttributeError` if *cm* is not a context manager.

push(*exit*)

컨텍스트 관리자의 `__exit__()` 메서드를 콜백 스택에 추가합니다.

`__enter__`가 호출되지 않기 때문에, 이 메서드를 사용하여 컨텍스트 관리자 고유의 `__exit__()` 메서드로 `__enter__()` 구현의 일부를 보호(cover)하는 데 사용할 수 있습니다.

컨텍스트 관리자가 아닌 객체를 전달하면, 이 메서드는 해당 객체가 컨텍스트 관리자의 `__exit__()` 메서드와 같은 서명을 가진 콜백인 것으로 가정하여 콜백 스택에 직접 추가합니다.

참값을 반환함으로써, 이 콜백은 컨텍스트 관리자 `__exit__()` 메서드와 같은 방식으로 예외를 억제할 수 있습니다.

전달된 객체는 함수에서 반환되어, 이 메서드를 함수 데코레이터로 사용할 수 있도록 합니다.

callback(*callback*, /, **args*, ***kwds*)

임의의 콜백 함수와 인자를 받아서 콜백 스택에 추가합니다.

다른 메서드와 달리, 이 방법으로 추가된 콜백은 예외를 무시할 수 없습니다 (예외 세부 정보가 전달되지 않기 때문입니다).

전달된 콜백은 함수에서 반환되어, 이 메서드를 함수 데코레이터로 사용할 수 있도록 합니다.

pop_all()

콜백 스택을 새로운 `ExitStack` 인스턴스로 옮기고 그것을 반환합니다. 이 작업으로 아무런 콜백도 호출되지 않습니다 - 대신, 이제 새 스택이 닫힐 때 (명시적으로나 `with` 문의 끝에서 묵시적으로) 호출됩니다.

예를 들어, 파일 그룹을 다음과 같이 "전부 아니면 아무것도" 방식으로 열 수 있습니다:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # close 메서드를 잡고 있지만, 아직 호출하지 않습니다.
    close_files = stack.pop_all().close
    # 어떤 파일을 여는 데 실패하면, 이전에 열었던 모든 파일이
    # 자동으로 닫힙니다. 모든 파일이 성공적으로 열리면,
    # with 문이 끝난 후에도 열린 상태로 유지됩니다. 그런 다음
    # close_files()를 명시적으로 호출하여 모두 닫을 수 있습니다.
```

close()

콜백 스택을 즉시 되감고, 등록 역순으로 콜백을 호출합니다. 등록된 모든 컨텍스트 관리자와 탈출 콜백에 전달되는 인자는 예외가 발생하지 않았음을 나타냅니다.

class contextlib.**AsyncExitStack**

`ExitStack`과 유사한 비동기 컨텍스트 관리자, 코루틴 정리 로직을 가질 뿐만 아니라 동기와 비동기 컨텍스트 관리자의 결합을 지원합니다.

`close()` 메서드는 구현되지 않았습니다; 대신 `aclose()`를 사용해야 합니다.

async enter_async_context(*cm*)

`ExitStack.enter_context()`와 유사하지만, 비동기 컨텍스트 관리자를 기대합니다.

버전 3.11에서 변경: Raises `TypeError` instead of `AttributeError` if *cm* is not an asynchronous context manager.

push_async_exit(*exit*)

*ExitStack.push()*와 유사하지만, 비동기 컨텍스트 관리자나 코루틴 함수를 기대합니다.

push_async_callback(*callback*, /, **args*, ***kwds*)

*ExitStack.callback()*과 유사하지만 코루틴 함수를 기대합니다.

async aclose()

*ExitStack.close()*와 유사하지만 어웨이터블을 올바르게 처리합니다.

asynccontextmanager() 에 대한 예제를 계속합니다:

```
async with AsyncExitStack() as stack:
    connections = [await stack.enter_async_context(get_connection())
        for i in range(5)]
    # 목록의 뒤에서 연결을 열려는 시도가 예외를
    # 발생시키더라도, async with 문의 끝에서
    # 열린 모든 연결이 자동으로 해제됩니다.
```

Added in version 3.7.

29.8.2 예제와 조리법

이 섹션에서는 *contextlib*가 제공하는 도구를 효과적으로 사용하기 위한 몇 가지 예와 조리법에 관해 설명합니다.

일정하지 않은 수의 컨텍스트 관리자 지원

*ExitStack*의 주요 사용 사례는 클래스 설명서에 제공된 것입니다: 일정하지 않은 수의 컨텍스트 관리자와 기타 정리 연산을 단일 `with` 문에서 지원합니다. 가변성은 사용자 입력(가령 사용자 지정한 파일 모음을 여는 것)에 의해 구동되는 필요한 컨텍스트 관리자의 수나, 일부 선택적인 컨텍스트 관리자의 존재에서 비롯될 수 있습니다:

```
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
        special = acquire_special_resource()
        stack.callback(release_special_resource, special)
    # 획득한 자원들을 사용하는 작업을 수행합니다
```

볼 수 있듯이, *ExitStack*을 사용하면 `with` 문을 사용하여 컨텍스트 관리자 프로토콜을 스스로 지원하지 않는 임의의 자원을 쉽게 관리할 수 있습니다.

`__enter__` 메서드에서 발생하는 예외 잡기

때때로 `with` 문 본문이나 컨텍스트 관리자의 `__exit__` 메서드에서 발생한 예외를 실수로 포착하지 않으면서, `__enter__` 메서드 구현에서 발생하는 예외를 포착하는 것이 바람직합니다. *ExitStack*을 사용하면 이를 위해 컨텍스트 관리자 프로토콜의 단계를 약간 분리할 수 있습니다:

```
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # __enter__ 예외를 처리합니다
else:
    with stack:
        # 정상적인 경우를 처리합니다
```

실제로 이렇게 할 필요가 있다는 것은 하부 API가 `try/except/finally` 문과 함께 사용하기 위한 직접 자원 관리 인터페이스를 제공해야 함을 나타내지만, 모든 API가 이런 측면에서 잘 설계된 것은 아닙니다. 컨텍

스트 관리자가 유일하게 제공되는 자원 관리 API 일 때, *ExitStack*을 사용하면 with 문에서 직접 처리할 수 없는 다양한 상황을 더 쉽게 처리할 수 있습니다.

__enter__ 구현에서 정리하기

ExitStack.push() 설명서에서 언급했듯이, 이 메서드는 __enter__() 구현의 후반 단계가 실패하면 이미 할당된 자원을 정리하는 데 유용할 수 있습니다.

다음은 선택적 유효성 검증 함수와 함께 자원 확보와 해제 함수를 받아들이고 이를 컨텍스트 관리 프로토콜에 매핑하는 컨텍스트 관리자를 위해 이를 수행하는 예제입니다:

```
from contextlib import contextmanager, AbstractContextManager, ExitStack

class ResourceManager(AbstractContextManager):

    def __init__(self, acquire_resource, release_resource, check_resource_ok=None):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
            def check_resource_ok(resource):
                return True
        self.check_resource_ok = check_resource_ok

    @contextmanager
    def _cleanup_on_error(self):
        with ExitStack() as stack:
            stack.push(self)
            yield
            # 유효성 검사가 통과되어 예외가 발생하지 않았습니다
            # 따라서, 자원을 유지하고 호출자에게 다시
            # 전달하려고 합니다
            stack.pop_all()

    def __enter__(self):
        resource = self.acquire_resource()
        with self._cleanup_on_error():
            if not self.check_resource_ok(resource):
                msg = "Failed validation for {!r}"
                raise RuntimeError(msg.format(resource))
        return resource

    def __exit__(self, *exc_details):
        # 자원 해제 로직을 중복할 필요가 없습니다
        self.release_resource()
```

try-finally와 플래그 변수 사용 교체하기

때때로 보이는 패턴은 finally 절의 본문을 실행할지를 나타내는 플래그 변수가 있는 try-finally 문입니다. 가장 간단한 형태(단지 대신 except 절을 사용하는 것만으로 이미 처리되었을 수 없는)는 다음과 같습니다:

```
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```

다른 try 문 기반 코드와 마찬가지로, 설정 코드와 정리 코드가 임의로 긴 코드 섹션으로 분리될 수 있어서 개발과 검토에 문제가 발생할 수 있습니다.

*ExitStack*을 사용하면 대신 with 문의 끝에서 실행할 콜백을 등록한 다음 나중에 해당 콜백 실행을 건너뛰기로 할 수 있습니다:

```
from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

이를 통해 별도의 플래그 변수를 요구하지 않고, 의도하는 정리 동작을 사전에 명시적으로 만들 수 있습니다.

특정 응용 프로그램에서 이 패턴을 많이 사용한다면, 작은 도우미 클래스를 사용하여 훨씬 더 단순화할 수 있습니다:

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, /, *args, **kwds):
        super().__init__()
        self.callback(callback, *args, **kwds)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

자원 정리가 아직 독립 함수로 깔끔하게 번들 되어 있지 않으면, *ExitStack.callback()*의 데코레이터 형식을 사용하여 자원 정리를 미리 선언할 수 있습니다:

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
    if result:
        stack.pop_all()
```

데코레이터 프로토콜의 작동 방식으로 인해, 이 방법으로 선언된 콜백 함수는 아무런 매개 변수도 취할 수 없습니다. 대신 해제할 모든 자원은 클로저(closure) 변수로 액세스해야 합니다.

함수 데코레이터로 컨텍스트 관리자 사용하기

*ContextDecorator*를 사용하면 일반적인 with 문과 함수 데코레이터 모두로 컨텍스트 관리자를 사용할 수 있습니다.

예를 들어, 진입 시간과 탈출 시간을 추적할 수 있는 로거(logger)로 함수나 문장 그룹을 감싸는 것이 유용할 때가 있습니다. 작업에 대한 함수 데코레이터와 컨텍스트 관리자를 모두 작성하는 대신, *ContextDecorator*를 상속하면 두 가지 기능을 하나의 정의로 제공합니다:

```
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

이 클래스의 인스턴스는 컨텍스트 관리자로 사용할 수 있습니다:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

또한 함수 데코레이터로도 사용할 수 있습니다:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

컨텍스트 관리자를 함수 데코레이터로 사용할 때 한 가지 추가 제한 사항이 있음에 유의하십시오: `__enter__()` 의 반환 값에 액세스 할 수 있는 방법이 없습니다. 이 값이 필요하면, 여전히 명시적인 `with` 문을 사용할 필요가 있습니다.

더 보기

PEP 343- "with" 문
파이썬 `with` 문의 명세, 배경 및 예

29.8.3 일회용, 재사용 가능 및 재진입 가능 컨텍스트 관리자

대부분의 컨텍스트 관리자는 `with` 문에서 한 번만 효과적으로 사용할 수 있다는 것을 의미하는 방식으로 작성됩니다. 이러한 일회용 컨텍스트 관리자는 사용될 때마다 새로 만들어야 합니다 - 두 번째로 사용하려고 하면 예외가 발생하거나 올바르게 작동하지 않습니다.

이 흔한 제한 사항은 일반적으로 컨텍스트 관리자가 사용되는 `with` 문의 헤더에서 컨텍스트 관리자를 직접 만들도록 권고하게 합니다 (위의 모든 사용 예제에서 보이듯이).

파일은 효과적인 일회용 컨텍스트 관리자의 예입니다, 첫 번째 `with` 문이 파일을 닫아서, 해당 파일 객체를 사용하는 추가 IO 연산을 막기 때문입니다.

`contextmanager()` 를 사용하여 만들어진 컨텍스트 관리자도 일회용 컨텍스트 관리자이며, 두 번째로 사용하려는 경우 하부 제너레이터가 산출에 실패하는 것에 대해 불평합니다:

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
    ...
RuntimeError: generator didn't yield
```

재진입 가능 컨텍스트 관리자

더욱 정교한 컨텍스트 관리자는 "재진입" 할 수 있습니다. 이러한 컨텍스트 관리자는 여러 `with` 문에서 사용될 수 있을 뿐만 아니라 이미 같은 컨텍스트 관리자를 사용하는 `with` 문 내부에서 사용될 수도 있습니다.

*threading.RLock*은 *suppress()*, *redirect_stdout()* 및 *chdir()*과 같이 재진입 가능 컨텍스트 관리자의 예입니다. 재진입 사용의 매우 간단한 예는 다음과 같습니다:

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...
>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```

재진입의 실제 예는 서로를 호출하는 여러 함수를 포함할 가능성이 높아서 이 예보다 훨씬 더 복잡합니다.

재진입 가능하다는 것이 스레드 안전하다는 것과 같지 않음에도 유의하십시오. 예를 들어, *redirect_stdout()*은 *sys.stdout*을 다른 스트림으로 연결하여 시스템 상태를 전역적으로 수정하므로 명백히 스레드 안전하지 않습니다.

재사용 가능 컨텍스트 관리자

일회용과 재진입 가능 컨텍스트 관리자와 구별되는 "재사용할 수 있는" 컨텍스트 관리자입니다 (또는 재진입 가능 컨텍스트 관리자도 재사용 가능하므로, 완전히 명시적이려면 "재사용할 수 있지만 재진입할 수 없는" 컨텍스트 관리자입니다). 이러한 컨텍스트 관리자는 여러 번 사용되는 것을 지원하지만, 같은 컨텍스트 관리자 인스턴스가 포함하는 with 문에서 이미 사용되었으면 실패합니다 (또는 올바르게 작동하지 않습니다).

*threading.Lock*은 재사용할 수 있지만 재진입할 수 없는 컨텍스트 관리자의 예입니다 (재진입 가능 록을 위해서는 *threading.RLock*을 대신 사용해야 합니다).

재사용할 수 있지만 재진입할 수 없는 컨텍스트 관리자의 또 다른 예는 *ExitStack*인 데, 콜백이 추가된 위치와 관계없이 with 문을 떠날 때 현재 등록된 콜백을 모두 호출하기 때문입니다:

```
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
...
Leaving first context
Callback: from first context
>>> with stack:
...     stack.callback(print, "Callback: from second context")
...     print("Leaving second context")
...
Leaving second context
Callback: from second context
>>> with stack:
...     stack.callback(print, "Callback: from outer context")
...     with stack:
...         stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Callback: from outer context
Leaving outer context
```

예제의 결과가 보여주듯이, 여러 with 문에서 단일 스택 객체를 재사용하는 것은 올바르게 작동하지만, 중첩을 시도하면 가장 안쪽 with 문 끝에서 스택이 지워지기 때문에 바람직한 동작이 아닙니다.

단일 인스턴스를 재사용하는 대신 별도의 *ExitStack* 인스턴스를 사용하면 이 문제를 피할 수 있습니다:

```
>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...     outer_stack.callback(print, "Callback: from outer context")
...     with ExitStack() as inner_stack:
...         inner_stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```

29.9 abc — 추상 베이스 클래스

소스 코드: Lib/abc.py

이 모듈은, **PEP 3119**에서 설명된 대로, 파이썬에서 추상 베이스 클래스 (ABC) 를 정의하기 위한 기반 구조를 제공합니다; 이것이 왜 파이썬에 추가되었는지는 PEP를 참조하십시오. (ABC를 기반으로 하는 숫자의 형 계층 구조에 관해서는 **PEP 3141**과 *numbers* 모듈을 참고하십시오.)

collections 모듈은 ABC로부터 파생된 몇 가지 구상(concrete) 클래스를 가지고 있습니다; 이것은 물론 더 파생될 수 있습니다. 또한, *collections.abc* 서브 모듈에는 클래스나 인스턴스가 특정 인터페이스를 (예를 들어, 해시 가능이거나 매핑이면) 제공하는지 검사하는 데 사용할 수 있는 ABC가 있습니다.

이 모듈은 ABC를 정의하기 위한 메타 클래스 *ABCMeta*와 상속을 통해 ABC를 정의하는 대안적 방법을 제공하는 도우미 클래스 *ABC*를 제공합니다:

class abc.**ABC**

*ABCMeta*를 메타 클래스로 가지는 도우미 클래스. 이 클래스를 사용하면, 때로 혼란스러운 메타 클래스를 사용하지 않고, 추상 베이스 클래스를 간단히 ABC에서 파생시켜서 만들 수 있습니다. 예를 들어:

```
from abc import ABC

class MyABC(ABC):
    pass
```

ABC 의 형은 여전히 *ABCMeta* 이므로, ABC를 상속할 때는 메타 클래스 사용에 관한 일반적인 주의가 필요한데, 다중 상속이 메타 클래스 충돌을 일으킬 수 있기 때문입니다. metaclass 키워드를 전달하고 ABCMeta를 직접 사용해서 추상 베이스 클래스를 정의할 수도 있습니다, 예를 들어:

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

Added in version 3.4.

class abc.**ABCMeta**

추상 베이스 클래스 (ABC)를 정의하기 위한 메타 클래스.

이 메타 클래스를 사용하여 ABC를 만듭니다. ABC는 직접 서브 클래싱 될 수 있으며 믹스인 클래스의 역할을 합니다. 관련 없는 구상 클래스(심지어 내장 클래스도)와 관련 없는 ABC를 "가상 서브 클래스"로 등록 할 수 있습니다 - 이들과 이들의 서브 클래스는 내장 *issubclass()* 함수에 의해 등록하는 ABC의 서브 클래스로 간주합니다. 하지만 등록하는 ABC는 그들의 MRO (메서드 결정 순서)에 나타나지 않을 것이고, 등록하는 ABC가 정의한 메서드 구현도 호출할 수 없을 것입니다 (*super()*를 통해서도 가능하지 않습니다).[1]

ABCMeta를 메타 클래스로 생성된 클래스는 다음과 같은 메서드를 가집니다:

register(*subclass*)

이 ABC의 "가상 서브 클래스"로 *subclass* 를 등록합니다. 예를 들면:

```
from abc import ABC

class MyABC(ABC):
    pass

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

버전 3.3에서 변경: 클래스 데코레이터로 사용할 수 있도록, 등록된 subclass 돌려줍니다.

버전 3.4에서 변경: register() 호출을 감지하려면, *get_cache_token()* 함수를 사용할 수 있습니다.

추상 베이스 클래스에서 다음 메서드를 재정의할 수도 있습니다:

__subclasshook__(*subclass*)

(반드시 클래스 메서드로 정의되어야 합니다.)

subclass 를 이 ABC의 서브 클래스로 간주할지를 검사합니다. 이것은, ABC의 서브 클래스로 취급하고 싶은 클래스마다 *register()*를 호출할 필요 없이, *issubclass()* 의 행동을 더 사용자

[1] C++ 프로그래머는 파이썬의 가상 베이스 클래스 개념이 C++과 다르다는 것을 알아야 합니다.

정의할 수 있음을 의미합니다. (이 클래스 메서드는 ABC의 `__subclasscheck__()` 메서드에서 호출됩니다.)

이 메서드는 `True`, `False` 또는 *`NotImplemented`* 를 반환해야 합니다. `True` 를 반환하면, *subclass* 를 이 ABC의 서브 클래스로 간주합니다. `False` 를 반환하면, *subclass* 를 ABC의 서브 클래스로 간주하지 않습니다. `NotImplemented` 를 반환하면, 서브 클래스 검사가 일반적인 메커니즘으로 계속됩니다.

이러한 개념들의 시연으로, 이 예제 ABC 정의를 보십시오:

```
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented

MyIterable.register(Foo)
```

ABC `MyIterable` 은 추상 메서드로 `__iter__()` 라는 표준 이터러블 메서드를 정의합니다. 여기에 제공된 구현은 여전히 서브 클래스에서 호출 할 수 있습니다. `get_iterator()` 메서드 또한 `MyIterable` 추상 베이스 클래스의 일부이지만, 추상이 아닌 파생 클래스에서 재정의될 필요는 없습니다.

여기에 정의된 *`__subclasshook__()`* 클래스 메서드는 자신의 (또는 그것의 `__mro__` 리스트를 통해 액세스 되는 베이스 클래스 중 하나의) `__dict__` 에 `__iter__()` 메서드를 가진 모든 클래스도 `MyIterable` 로 간주한다고 말합니다.

마지막으로, 마지막 줄은, `Foo` 가 `__iter__()` 메서드를 정의하지는 않았음에도 불구하고 (이것은 `__len__()` 과 `__getitem__()` 로 정의되는 이전 스타일의 이터러블 프로토콜을 사용합니다), `MyIterable` 의 가상 서브 클래스로 만듭니다. 이렇게 하면 `get_iterator` 가 `Foo` 의 메서드로 사용할 수 있지 않으므로, 별도로 제공됩니다.

`abc` 모듈은 다음 데코레이터도 제공합니다:

`@abc.`**`abstractmethod`**

추상 메서드를 나타내는 데코레이터.

이 데코레이터를 사용하려면 클래스의 메타 클래스가 *`ABCMeta`* 이거나 여기에서 파생된 것이어야 합니다. `ABCMeta` 에서 파생된 메타 클래스를 가진 클래스는 모든 추상 메서드와 프로퍼티가 재정의되지 않는 한 인스턴스로 만들 수 없습니다. 추상 메서드는 일반적인 'super' 호출 메커니즘을 사용하여 호출 할 수 있습니다. `abstractmethod()` 는 프로퍼티와 디스크립터에 대한 추상 메서드를 선언하는 데 사용될 수 있습니다.

클래스에 추상 메서드를 동적으로 추가하거나, 메서드나 클래스가 작성된 후에 추상화 상태를 수정하려고 시도하는 것은 *update_abstractmethods()* 함수를 통해서만 지원됩니다. abstractmethod() 는 정규 상속을 사용하여 파생된 서브 클래스에만 영향을 줍니다; ABC의 *register()* 메서드로 등록된 "가상 서브 클래스" 는 영향을 받지 않습니다.

abstractmethod() 가 다른 메서드 디스크립터와 함께 적용될 때, 다음 사용 예제와 같이 가장 안쪽의 데코레이터로 적용되어야 합니다:

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, arg1):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg2):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg3):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...
    @abstractmethod
    def _set_x(self, val):
        ...
    x = property(_get_x, _set_x)
```

추상 베이스 클래스 장치와 정확하게 상호 작용하기 위해서, 디스크립터는 __isabstractmethod__ 를 사용하여 자신을 추상으로 식별해야 합니다. 일반적으로 이 어트리뷰트는 디스크립터를 구성하는 데 사용된 메서드 중 어느 하나라도 추상이면 True 여야 합니다. 예를 들어, 파이썬의 내장 *property*는 다음과 동등한 일을 합니다:

```
class Descriptor:
    ...
    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False) for
                   f in (self._fget, self._fset, self._fdel))
```

> **참고**
>
> 자바 추상 메서드와 달리, 이 추상 메서드는 구현을 가질 수 있습니다. 이 구현은 그것을 재정의하는 클래스에서 *super()* 메커니즘을 통해 호출 할 수 있습니다. 이는 협업적 다중 상속을 사용하는 프레임워크에서 super-호출의 종점으로 유용 할 수 있습니다.

abc 모듈은 다음 레거시 데코레이터도 지원합니다:

@abc.**abstractclassmethod**

Added in version 3.2.

버전 3.3부터 폐지됨: 이제 *classmethod* 와 *abstractmethod()* 를 함께 사용할 수 있어서, 이 데코레이터는 필요 없습니다.

내장 *classmethod()* 의 서브 클래스로, 추상 classmethod 를 나타냅니다. 그 외에는 *abstractmethod()* 와 유사합니다.

classmethod() 데코레이터가 이제 추상 메서드에 적용될 때 추상으로 정확하게 식별되기 때문에, 이 특별한 경우는 폐지되었습니다.:

```
class C(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg):
        ...
```

@abc.**abstractstaticmethod**

Added in version 3.2.

버전 3.3부터 폐지됨: 이제 *staticmethod* 와 *abstractmethod()* 를 함께 사용할 수 있어서, 이 데코레이터는 필요 없습니다.

내장 *staticmethod()* 의 서브 클래스로, 추상 staticmethod 를 나타냅니다. 그 외에는 *abstractmethod()* 와 유사합니다.

staticmethod() 데코레이터가 이제 추상 메서드에 적용될 때 추상으로 정확하게 식별되기 때문에, 이 특별한 경우는 폐지되었습니다.:

```
class C(ABC):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg):
        ...
```

@abc.**abstractproperty**

버전 3.3부터 폐지됨: 이제 *property*, *property.getter()*, *property.setter()*, *property.deleter()* 와 *abstractmethod()* 를 함께 사용할 수 있어서, 이 데코레이터는 필요 없습니다.

내장 *property()* 의 서브 클래스로, 추상 property를 나타냅니다.

property() 데코레이터가 이제 추상 메서드에 적용될 때 추상으로 정확하게 식별되기 때문에, 이 특별한 경우는 폐지되었습니다.:

```
class C(ABC):
    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
```

위의 예제는 읽기 전용 프로퍼티를 정의합니다; 하나나 그 이상의 하부 메서드를 추상으로 적절하게 표시하여 읽기-쓰기 추상 프로퍼티를 정의할 수도 있습니다:

```
class C(ABC):
    @property
    def x(self):
        ...

    @x.setter
    @abstractmethod
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    def x(self, val):
        ...
```

일부 구성 요소만 추상인 경우, 서브 클래스에서 구상 프로퍼티를 만들기 위해서는 해당 구성 요소만 갱신하면 됩니다:

```
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

abc 모듈은 또한 다음과 같은 기능을 제공합니다:

abc.**get_cache_token**()

현재의 추상 베이스 클래스 캐시 토큰을 반환합니다.

토큰은 가상 서브 클래스를 위한 추상 베이스 클래스 캐시의 현재 버전을 식별하는 (동등성 검사를 지원하는) 불투명 객체입니다. 임의의 ABC에서 `ABCMeta.register()` 가 호출될 때마다 토큰이 변경됩니다.

Added in version 3.4.

abc.**update_abstractmethods**(*cls*)

A function to recalculate an abstract class' s abstraction status. This function should be called if a class' s abstract methods have been implemented or changed after it was created. Usually, this function should be called from within a class decorator.

클래스 데코레이터로 사용할 수 있도록, *cls*를 돌려줍니다.

If *cls* is not an instance of `ABCMeta`, does nothing.

> **참고**
>
> This function assumes that *cls*' s superclasses are already updated. It does not update any subclasses.

Added in version 3.10.

29.10 atexit — 종료 처리기

`atexit` 모듈은 정리 함수를 등록하고 해제하는 함수를 정의합니다. 이렇게 등록된 함수는 정상적인 인터프리터 종료 시 자동으로 실행됩니다. `atexit`는 이 함수들을 등록된 역순 으로 실행합니다; A, B, C 를 등록하면, 인터프리터 종료 시각에 C, B, A 순서로 실행됩니다.

참고: 이 모듈을 통해 등록된 함수는 다음과 같은 경우 호출되지 않습니다. 프로그램이 파이썬이 처리하지 않는 시그널에 의해 종료될 때. 파이썬의 치명적인 내부 에러가 감지되었을 때. `os._exit()` 가 호출될 때.

Note: The effect of registering or unregistering functions from within a cleanup function is undefined.

버전 3.7에서 변경: C-API 서브 인터프리터와 함께 사용될 때, 등록된 함수는 등록된 인터프리터에 국지적입니다.

atexit.**register**(*func*, **args*, ***kwargs*)

func 를 종료 시에 실행할 함수로 등록합니다. *func* 에 전달되어야 하는 선택적 인자는 `register()` 에 인자로 전달되어야 합니다. 같은 함수와 인자를 두 번 이상 등록 할 수 있습니다.

정상적인 프로그램 종료 시에 (예를 들어, `sys.exit()` 가 호출되거나 주 모듈의 실행이 완료된 경우에), 등록된 모든 함수는 후입선출 순서로 호출됩니다. 낮은 수준의 모듈은 일반적으로 상위 수준 모듈보다 먼저 임포트 되기 때문에 나중에 정리해야 한다는 가정입니다.

종료 처리기의 실행 중에 예외가 발생하면 (`SystemExit`가 발생하지 않는다면) 트레이스백이 인쇄되고 예외 정보가 저장됩니다. 모든 종료 처리기가 실행될 기회를 얻은 후에, 발생한 마지막 예외를 다시 일으킵니다.

이 함수는 *func* 을 반환하므로 데코레이터로 사용할 수 있습니다.

> **경고**
>
> Starting new threads or calling `os.fork()` from a registered function can lead to race condition between the main Python runtime thread freeing thread states while internal `threading` routines or the new process try to use that state. This can lead to crashes rather than clean shutdown.

버전 3.12에서 변경: Attempts to start a new thread or `os.fork()` a new process in a registered function now leads to `RuntimeError`.

atexit.**unregister**(*func*)

인터프리터 종료 시 실행할 함수 목록에서 *func* 를 제거합니다. `unregister()` 는 *func* 가 이전에 등록되지 않았다면 아무것도 하지 않습니다. *func*가 두 번 이상 등록되었으면, `atexit` 호출 스택에서 해당 함수를 모두 제거합니다. 등록을 해제하는 동안 내부적으로 동등 비교(==)가 사용되므로, 함수 참조는 대응하는 아이덴티티가 필요 없습니다.

> **더 보기**
>
> 모듈 ***readline***
>
> `readline` 히스토리 파일을 읽고 쓰는 `atexit` 의 유용한 예.

29.10.1 atexit 예제

다음의 간단한 예제는, 모듈이 임포트 될 때 파일에서 카운터를 읽고 프로그램이 종료할 때 프로그램의 명시적인 호출에 의존하지 않고 카운터의 변경된 값을 자동으로 저장하는 방법을 보여줍니다.:

```
try:
    with open('counterfile') as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    with open('counterfile', 'w') as outfile:
        outfile.write('%d' % _count)

import atexit

atexit.register(savecounter)
```

위치 및 키워드 인자가 등록된 함수가 호출될 때 전달되도록 `register()` 에 전달할 수 있습니다:

```
def goodbye(name, adjective):
    print('Goodbye %s, it was %s to meet you.' % (name, adjective))

import atexit
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
atexit.register(goodbye, 'Donny', 'nice')
# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

데코레이터 로 사용하기:

```
import atexit

@atexit.register
def goodbye():
    print('You are now leaving the Python sector.')
```

이 방법은 인자 없이 호출할 수 있는 함수에서만 작동합니다.

29.11 traceback — 스택 트레이스백 인쇄와 조회

소스 코드: Lib/traceback.py

This module provides a standard interface to extract, format and print stack traces of Python programs. It is more flexible than the interpreter' s default traceback display, and therefore makes it possible to configure certain aspects of the output. Finally, it contains a utility for capturing enough information about an exception to print it later, without the need to save a reference to the actual exception. Since exceptions can be the roots of large objects graph, this utility can significantly improve memory management.

The module uses traceback objects — these are objects of type `types.TracebackType`, which are assigned to the `__traceback__` field of `BaseException` instances.

더 보기

모듈 ***faulthandler***
Used to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal.

모듈 ***pdb***
Interactive source code debugger for Python programs.

The module' s API can be divided into two parts:

- Module-level functions offering basic functionality, which are useful for interactive inspection of exceptions and tracebacks.
- `TracebackException` class and its helper classes `StackSummary` and `FrameSummary`. These offer both more flexibility in the output generated and the ability to store the information necessary for later formatting without holding references to actual exception and traceback objects.

Added in version 3.13: Output is colorized by default and can be controlled using environment variables.

29.11.1 모듈 수준 함수

traceback.**print_tb**(*tb*, *limit=None*, *file=None*)

*limit*가 양수면 (호출자 프레임에서 시작하여) 트레이스백 객체 *tb*의 최대 *limit* 개의 스택 트레이스 항목을 인쇄합니다. 그렇지 않으면, 마지막 `abs(limit)` 항목을 인쇄합니다. *limit*가 생략되거나 `None` 이면, 모든 항목이 인쇄됩니다. *file*이 생략되거나 `None` 이면, 출력은 `sys.stderr`로 갑니다; 그렇지 않으면 출력을 받을 열린 파일이나 파일류 객체여야 합니다.

> **참고**
>
> The meaning of the *limit* parameter is different than the meaning of `sys.tracebacklimit`. A negative *limit* value corresponds to a positive value of `sys.tracebacklimit`, whereas the behaviour of a positive *limit* value cannot be achieved with `sys.tracebacklimit`.

버전 3.5에서 변경: 음수 *limit* 지원을 추가했습니다.

`traceback.`**`print_exception`**(*exc*, /, [*value*, *tb*,]*limit=None*, *file=None*, *chain=True*)

예외 정보와 트레이스백 객체 *tb*의 스택 트레이스 항목을 *file*로 인쇄합니다. 이것은 다음과 같은 점에서 `print_tb()`와 다릅니다:

- *tb*가 `None`이 아니면, 헤더 `Traceback (most recent call last):`를 인쇄합니다.
- 스택 트레이스 다음에 예외 형과 *value*를 인쇄합니다
- *type*(*value*)가 `SyntaxError`고 *value*가 적절한 형식을 가지면, 에러의 대략적인 위치를 나타내는 캐럿(caret)과 함께 문법 에러가 발생한 줄을 인쇄합니다.

Since Python 3.10, instead of passing *value* and *tb*, an exception object can be passed as the first argument. If *value* and *tb* are provided, the first argument is ignored in order to provide backwards compatibility.

선택적 *limit* 인자는 `print_tb()`와 같은 의미입니다. *chain*이 참(기본값)이면, 처리되지 않은 예외를 인쇄할 때 인터프리터 자체가 하는 것과 마찬가지로, 연결된 예외(예외의 `__cause__`나 `__context__` 어트리뷰트)도 인쇄됩니다.

버전 3.5에서 변경: *etype* 인자는 무시되고 *value* 형에서 유추됩니다.

버전 3.10에서 변경: The *etype* parameter has been renamed to *exc* and is now positional-only.

`traceback.`**`print_exc`**(*limit=None*, *file=None*, *chain=True*)

이것은 `print_exception(sys.exception(), limit=limit, file=file, chain=chain)`의 줄임 표현입니다.

`traceback.`**`print_last`**(*limit=None*, *file=None*, *chain=True*)

이것은 `print_exception(sys.last_exc, limit=limit, file=file, chain=chain)`의 줄임 표현입니다. 일반적으로 예외가 대화식 프롬프트에 도달한 후에만 작동합니다(`sys.last_exc`을 참조하십시오).

`traceback.`**`print_stack`**(*f=None*, *limit=None*, *file=None*)

*limit*가 양수면 (호출 지점에서 시작하여) 최대 *limit* 개의 스택 트레이스 항목을 인쇄합니다. 그렇지 않으면, 마지막 `abs(limit)` 항목을 인쇄합니다. *limit*가 생략되거나 `None`이면, 모든 항목이 인쇄됩니다. 선택적 *f* 인자를 사용하여 시작할 대체 스택 프레임을 지정할 수 있습니다. 선택적 *file* 인자는 `print_tb()`와 같은 의미입니다.

버전 3.5에서 변경: 음수 *limit* 지원을 추가했습니다.

`traceback.`**`extract_tb`**(*tb*, *limit=None*)

트레이스백 객체 *tb*에서 추출된 "전 처리된" 스택 트레이스 항목의 리스트를 나타내는 `StackSummary` 객체를 반환합니다. 스택 트레이스의 대체 포매팅으로 유용합니다. 선택적 *limit* 인자는 `print_tb()`와 같은 의미입니다. "전 처리된" 스택 트레이스 항목은 일반적으로 스택 트레이스를 위해 인쇄되는 정보를 나타내는 어트리뷰트 `filename`, `lineno`, `name` 및 `line`을 포함하는 `FrameSummary` 객체입니다.

`traceback.`**`extract_stack`**(*f=None*, *limit=None*)

현재 스택 프레임에서 날 트레이스백을 추출합니다. 반환 값은 `extract_tb()`와 같은 형식입니다. 선택적 *f*와 *limit* 인자는 `print_stack()`과 같은 의미입니다.

`traceback.`**`print_list`**(*extracted_list*, *file=None*)

Print the list of tuples as returned by `extract_tb()` or `extract_stack()` as a formatted stack trace to the given file. If *file* is `None`, the output is written to `sys.stderr`.

traceback.**format_list**(*extracted_list*)

extract_tb() 나 extract_stack() 이 반환한 튜플이나 FrameSummary 객체의 리스트가 제공되면, 인쇄할 준비가 된 문자열의 리스트를 반환합니다. 결과 리스트의 각 문자열은 인자 리스트에서 같은 인덱스를 가진 항목에 해당합니다. 각 문자열은 줄 바꿈으로 끝납니다; 소스 텍스트 줄이 None 이 아닌 항목의 경우, 문자열에 내부 줄 바꿈도 포함될 수 있습니다.

traceback.**format_exception_only**(*exc*, */*, [*value*,]*, *show_group=False*)

트레이스백의 예외 부분을 sys.last_value로 주어진 것과 같은 예외 값을 사용하여 포맷합니다. 반환 값은 각각 줄 바꿈으로 끝나는 문자열의 리스트입니다. 리스트는 예외 메시지를 포함하는데, 일반적으로 단일 문자열입니다; 그러나, SyntaxError 예외의 경우, 문법 에러가 발생한 위치에 대한 자세한 정보를 (인쇄될 때) 표시하는 여러 줄을 포함합니다. 어떤 예외가 발생했는지를 나타내는 메시지는 리스트에서 항상 마지막 문자열입니다. 메시지 다음으로, 리스트는 예외의 notes를 포함합니다.

Since Python 3.10, instead of passing *value*, an exception object can be passed as the first argument. If *value* is provided, the first argument is ignored in order to provide backwards compatibility.

When *show_group* is True, and the exception is an instance of BaseExceptionGroup, the nested exceptions are included as well, recursively, with indentation relative to their nesting depth.

버전 3.10에서 변경: The *etype* parameter has been renamed to *exc* and is now positional-only.

버전 3.11에서 변경: The returned list now includes any notes attached to the exception.

버전 3.13에서 변경: *show_group* parameter was added.

traceback.**format_exception**(*exc*, */*, [*value*, *tb*,]*limit=None*, *chain=True*)

스택 트레이스와 예외 정보를 포맷합니다. 인자는 print_exception()의 해당하는 인자와 같은 의미입니다. 반환 값은 각각 줄 바꿈으로 끝나고 일부는 내부 줄 바꿈을 포함하는 문자열의 리스트입니다. 이 줄들을 이어붙여서 인쇄하면, print_exception()과 정확히 같은 텍스트가 인쇄됩니다.

버전 3.5에서 변경: *etype* 인자는 무시되고 *value* 형에서 유추됩니다.

버전 3.10에서 변경: This function' s behavior and signature were modified to match print_exception().

traceback.**format_exc**(*limit=None*, *chain=True*)

이것은 print_exc(limit) 와 비슷하지만, 파일로 인쇄하는 대신 문자열을 반환합니다.

traceback.**format_tb**(*tb*, *limit=None*)

format_list(extract_tb(tb, limit)) 의 줄임 표현입니다.

traceback.**format_stack**(*f=None*, *limit=None*)

format_list(extract_stack(f, limit)) 의 줄임 표현입니다.

traceback.**clear_frames**(*tb*)

각 프레임 객체의 clear() 메서드를 호출하여 트레이스백 *tb*에 있는 모든 스택 프레임의 지역 변수를 지웁니다.

Added in version 3.4.

traceback.**walk_stack**(*f*)

주어진 프레임에서 f.f_back을 따라 스택을 걸어가며 각 프레임의 프레임과 줄 번호를 산출(yield) 합니다. *f*가 None 이면, 현재 스택이 사용됩니다. 이 도우미는 StackSummary.extract()와 함께 사용됩니다.

Added in version 3.5.

버전 3.14에서 변경: This function previously returned a generator that would walk the stack when first iterated over. The generator returned now is the state of the stack when walk_stack is called.

traceback.**walk_tb**(*tb*)

tb_next를 따라 트레이스백을 걸으면서 각 프레임의 프레임과 줄 번호를 산출(yield) 합니다. 이 도우미는 StackSummary.extract()와 함께 사용됩니다.

Added in version 3.5.

29.11.2 TracebackException 객체

Added in version 3.5.

TracebackException objects are created from actual exceptions to capture data for later printing. They offer a more lightweight method of storing this information by avoiding holding references to traceback and frame objects. In addition, they expose more options to configure the output compared to the module-level functions described above.

class traceback.**TracebackException**(*exc_type*, *exc_value*, *exc_traceback*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*, *compact=False*, *max_group_width=15*, *max_group_depth=10*)

나중에 렌더링하기 위해 예외를 포착합니다. *limit*, *lookup_lines* 및 *capture_locals*의 의미는 *StackSummary* 클래스와 같습니다.

If *compact* is true, only data that is required by TracebackException's *format()* method is saved in the class attributes. In particular, the *__context__* field is calculated only if *__cause__* is None and *__suppress_context__* is false.

locals가 포착되면, 트레이스백에도 표시됨에 유의하십시오.

max_group_width and *max_group_depth* control the formatting of exception groups (see *BaseExceptionGroup*). The depth refers to the nesting level of the group, and the width refers to the size of a single exception group's exceptions array. The formatted output is truncated when either limit is exceeded.

버전 3.10에서 변경: Added the *compact* parameter.

버전 3.11에서 변경: Added the *max_group_width* and *max_group_depth* parameters.

__cause__

원래 *__cause__*의 TracebackException.

__context__

원래 *__context__*의 TracebackException.

exceptions

If self represents an *ExceptionGroup*, this field holds a list of TracebackException instances representing the nested exceptions. Otherwise it is None.

Added in version 3.11.

__suppress_context__

원래 예외의 *__suppress_context__* 값.

__notes__

The *__notes__* value from the original exception, or None if the exception does not have any notes. If it is not None is it formatted in the traceback after the exception string.

Added in version 3.11.

stack

트레이스백을 나타내는 *StackSummary*.

exc_type

원래 트레이스백의 클래스.

버전 3.13부터 폐지됨.

exc_type_str

원래 예외 클래스의 문자열 표시.

Added in version 3.13.

filename

문법 에러일 때 - 에러가 발생한 파일 이름.

lineno

문법 에러일 때 - 에러가 발생한 줄 번호.

end_lineno

문법 에러일 때 - 에러가 발생한 끝 줄 번호. 존재하지 않으면 `None` 일 수 있습니다.

Added in version 3.10.

text

문법 에러일 때 - 에러가 발생한 텍스트.

offset

문법 에러일 때 - 에러가 발생한 텍스트에서의 오프셋.

end_offset

문법 에러일 때 - 에러가 발생한 텍스트에서의 끝 오프셋. 존재하지 않으면 `None` 일 수 있습니다.

Added in version 3.10.

msg

문법 에러일 때 - 컴파일러 에러 메시지.

classmethod from_exception(*exc*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

나중에 렌더링하기 위해 예외를 포착합니다. *limit*, *lookup_lines* 및 *capture_locals* 는 `StackSummary` 클래스와 같습니다.

locals 가 포착되면, 트레이스백에도 표시됨에 유의하십시오.

print(*, *file=None*, *chain=True*)

Print to *file* (default `sys.stderr`) the exception information returned by `format()`.

Added in version 3.11.

format(*, *chain=True*)

예외를 포맷합니다.

chain 이 `True` 가 아니면, `__cause__` 와 `__context__` 는 포맷되지 않습니다.

반환 값은 각각 줄 바꿈으로 끝나고 일부는 내부 줄 바꿈을 포함하는 문자열의 제너레이터입니다. `print_exception()` 은 단지 파일에 줄을 인쇄하는 이 메서드를 둘러싸는 래퍼입니다.

format_exception_only(*, *show_group=False*)

트레이스백의 예외 부분을 포맷합니다.

반환 값은 각각 줄 바꿈으로 끝나는 문자열의 제너레이터입니다.

show_group 이 `False` 일 때, 제너레이터는 예외 메시지와 (노트가 있다면) 그 뒤를 따르는 노트를 방출합니다. 예외 메시지는 일반적으로 단일 문자열입니다; 그러나 `SyntaxError` 예외의 경우, 문법 에러가 발생한 위치에 대한 자세한 정보를 (인쇄할 때) 표시하는 여러 줄로 구성됩니다.

When *show_group* is `True`, and the exception is an instance of `BaseExceptionGroup`, the nested exceptions are included as well, recursively, with indentation relative to their nesting depth.

버전 3.11 에서 변경: The exception' s `notes` are now included in the output.

버전 3.13 에서 변경: Added the *show_group* parameter.

29.11.3 `StackSummary` 객체

Added in version 3.5.

`StackSummary` 객체는 포맷 준비가 된 호출 스택을 나타냅니다.

class traceback.**StackSummary**

classmethod extract(*frame_gen*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

(`walk_stack()`이나 `walk_tb()`에서 반환된 것과 같은) 프레임 제너레이터로 `StackSummary` 객체를 생성합니다.

*limit*가 제공되면, *frame_gen*에서 이 수 만큼의 프레임만 취합니다. *lookup_lines*가 `False`이면, 반환된 `FrameSummary` 객체는 아직 해당 줄을 읽지 않아서 `StackSummary`를 만드는 비용을 줄입니다 (실제로 포맷되지 않을 수 있다면 유용 할 수 있습니다). *capture_locals*가 `True`이면, 각 `FrameSummary`의 지역 변수는 객체 표현(representation)으로 포착됩니다.

버전 3.12에서 변경: Exceptions raised from `repr()` on a local variable (when *capture_locals* is `True`) are no longer propagated to the caller.

classmethod from_list(*a_list*)

제공된 `FrameSummary` 객체의 리스트나 이전 스타일의 튜플 리스트로 `StackSummary` 객체를 생성합니다. 각 튜플은 *filename*, *lineno*, *name*, *line*을 요소로 하는 4-튜플이어야 합니다.

format()

인쇄 준비가 된 문자열의 리스트를 반환합니다. 결과 리스트의 각 문자열은 스택의 단일 프레임에 해당합니다. 각 문자열은 줄 바꿈으로 끝납니다; 소스 텍스트 줄이 있는 항목의 경우, 문자열에 내부 줄 바꿈도 포함될 수 있습니다.

같은 프레임과 줄의 긴 시퀀스의 경우, 처음 몇 번의 반복이 표시된 다음, 정확한 추가의 반복 횟수를 나타내는 요약 줄이 표시됩니다.

버전 3.6에서 변경: 반복되는 프레임의 긴 시퀀스가 이제 축약됩니다.

format_frame_summary(*frame_summary*)

Returns a string for printing one of the frames involved in the stack. This method is called for each `FrameSummary` object to be printed by `StackSummary.format()`. If it returns `None`, the frame is omitted from the output.

Added in version 3.11.

29.11.4 FrameSummary 객체

Added in version 3.5.

`FrameSummary` 객체는 트레이스백에서 단일 프레임을 나타냅니다.

class traceback.**FrameSummary**(*filename*, *lineno*, *name*, *, *lookup_line=True*, *locals=None*, *line=None*, *end_lineno=None*, *colno=None*, *end_colno=None*)

포맷되거나 인쇄 중인 트레이스백이나 스택의 단일 프레임을 나타냅니다. 선택적으로 문자열로 변환된 버전의 프레임 지역 변수를 포함할 수 있습니다. *lookup_line*이 `False`이면, `FrameSummary`의 `line` 어트리뷰트에 액세스할 때까지 (`tuple`로 캐스트 할 때도 발생합니다) 소스 코드를 찾지 않습니다. `line`은 직접 제공될 수 있으며, 줄 조회가 전혀 발생하지 않도록 합니다. *locals*는 선택적 지역 변수 매핑이며, 제공되면 변수 표현(representation)은 나중에 표시할 수 있도록 요약에 저장됩니다.

`FrameSummary` instances have the following attributes:

filename

The filename of the source code for this frame. Equivalent to accessing `f.f_code.co_filename` on a frame object *f*.

lineno

The line number of the source code for this frame.

name

Equivalent to accessing `f.f_code.co_name` on a frame object *f*.

line

A string representing the source code for this frame, with leading and trailing whitespace stripped. If the source is not available, it is `None`.

end_lineno

The last line number of the source code for this frame. By default, it is set to `lineno` and indexation starts from 1.

버전 3.13에서 변경: The default value changed from `None` to `lineno`.

colno

The column number of the source code for this frame. By default, it is `None` and indexation starts from 0.

end_colno

The last column number of the source code for this frame. By default, it is `None` and indexation starts from 0.

29.11.5 Examples of Using the Module-Level Functions

이 간단한 예제는 표준 파이썬 대화식 인터프리터 루프와 비슷하지만 (하지만 덜 유용한) 기본적인 읽기-평가-인쇄 루프를 구현합니다. 인터프리터 루프의 더욱 완전한 구현은 *code* 모듈을 참조하십시오.

```
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
while True:
    run_user_code(envdir)
```

다음 예제는 예외와 추적을 인쇄하고 포맷하는 다양한 방법을 보여줍니다:

```
import sys, traceback

def lumberjack():
    bright_side_of_life()

def bright_side_of_life():
    return tuple()[0]

try:
    lumberjack()
except IndexError as exc:
    print("*** print_tb:")
    traceback.print_tb(exc.__traceback__, limit=1, file=sys.stdout)
    print("*** print_exception:")
    traceback.print_exception(exc, limit=2, file=sys.stdout)
    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    print(repr(traceback.format_exception(exc)))
    print("*** extract_tb:")
    print(repr(traceback.extract_tb(exc.__traceback__)))
    print("*** format_tb:")
    print(repr(traceback.format_tb(exc.__traceback__)))
    print("*** tb_lineno:", exc.__traceback__.tb_lineno)
```

예제의 출력은 다음과 유사합니다:

```
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest default[0]>", line 10, in <module>\n    lumberjack()\n    ~~~~~~
↪~~~~^^\n',
 '  File "<doctest default[0]>", line 4, in lumberjack\n    bright_side_of_life()\
↪n    ~~~~~~~~~~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 7, in bright_side_of_life\n    return␣
↪tuple()[0]\n           ~~~~~~~^^^\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[<FrameSummary file <doctest...>, line 10 in <module>>,
 <FrameSummary file <doctest...>, line 4 in lumberjack>,
 <FrameSummary file <doctest...>, line 7 in bright_side_of_life>]
*** format_tb:
['  File "<doctest default[0]>", line 10, in <module>\n    lumberjack()\n    ~~~~~~
↪~~~~^^\n',
 '  File "<doctest default[0]>", line 4, in lumberjack\n    bright_side_of_life()\
↪n    ~~~~~~~~~~~~~~~~~~~^^\n',
 '  File "<doctest default[0]>", line 7, in bright_side_of_life\n    return␣
↪tuple()[0]\n           ~~~~~~~^^^\n']
*** tb_lineno: 10
```

다음 예제는 스택을 인쇄하고 포맷하는 다양한 방법을 보여줍니다:

```
>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print(repr(traceback.extract_stack()))
...     print(repr(traceback.format_stack()))
...
>>> another_function()
  File "<doctest>", line 10, in <module>
    another_function()
  File "<doctest>", line 3, in another_function
    lumberstack()
  File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print(repr(traceback.extract_stack()))')]
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print(repr(traceback.format_
↪stack()))\n']
```

이 마지막 예제는 마지막 몇 가지 포매팅 함수를 예시합니다:

```
>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return "bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(an_error)
['IndexError: tuple index out of range\n']
```

29.11.6 `TracebackException`을 사용하는 예제

With the helper class, we have more options:

```
>>> import sys
>>> from traceback import TracebackException
>>>
>>> def lumberjack():
...     bright_side_of_life()
...
>>> def bright_side_of_life():
...     t = "bright", "side", "of", "life"
...     return t[5]
...
>>> try:
...     lumberjack()
... except IndexError as e:
...     exc = e
...
>>> try:
...     try:
...         lumberjack()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     except:
...         1/0
... except Exception as e:
...     chained_exc = e
...
>>> # limit 는 모듈 수준 함수처럼 작동합니다
>>> TracebackException.from_exception(exc, limit=-2).print()
Traceback (most recent call last):
  File "<python-input-1>", line 6, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
  File "<python-input-1>", line 10, in bright_side_of_life
    return t[5]
           ~^^^
IndexError: tuple index out of range

>>> # capture_locals 는 프레임에 지역 변수를 추가합니다
>>> TracebackException.from_exception(exc, limit=-2, capture_locals=True).print()
Traceback (most recent call last):
  File "<python-input-1>", line 6, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
  File "<python-input-1>", line 10, in bright_side_of_life
    return t[5]
           ~^^^
    t = ("bright", "side", "of", "life")
IndexError: tuple index out of range

>>> # print() 의 *chain* 키워드 인자는 연결된 예외를
>>> # 표시할지를 제어합니다
>>> TracebackException.from_exception(chained_exc).print()
Traceback (most recent call last):
  File "<python-input-19>", line 4, in <module>
    lumberjack()
    ~~~~~~~~~~^^
  File "<python-input-8>", line 7, in lumberjack
    bright_side_of_life()
    ~~~~~~~~~~~~~~~~~~~^^
  File "<python-input-8>", line 11, in bright_side_of_life
    return t[5]
           ~^^^
IndexError: tuple index out of range

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<python-input-19>", line 6, in <module>
    1/0
    ~^~
ZeroDivisionError: division by zero

>>> TracebackException.from_exception(chained_exc).print(chain=False)
Traceback (most recent call last):
  File "<python-input-19>", line 6, in <module>
    1/0
    ~^~
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
ZeroDivisionError: division by zero
```

29.12 __future__ — 퓨처 문 정의

소스 코드: Lib/__future__.py

Imports of the form `from __future__ import feature` are called future statements. These are special-cased by the Python compiler to allow the use of new Python features in modules containing the future statement before the release in which the feature becomes standard.

While these future statements are given additional special meaning by the Python compiler, they are still executed like any other import statement and the *__future__* exists and is handled by the import system the same way any other Python module would be. This design serves three purposes:

- 임포트 문을 분석하고 임포트하는 모듈을 발견하리라고 기대하는 기존 도구가 혼동하지 않게 하려고.
- 호환되지 않는 변경 사항이 도입된 시점과 그것이 필수적일 때를 — 또는 이미 필수적으로 된 때를 — 문서로 만들기 위해. 이것은 실행 가능한 문서 형식이며, *__future__* 를 임포트 해서 내용을 들여다봄으로써 프로그래밍 방식으로 검사 할 수 있습니다.
- 파이썬 2.1 이전의 배포에서 퓨처 문 을 실행하면 최소한 실행시간 예외를 일으키도록 보장하기 위해 (2.1 이전에는 그런 이름의 모듈이 없으므로 *__future__* 임포트는 실패합니다).

29.12.1 Module Contents

어떤 기능 설명도 *__future__* 에서 삭제되지 않습니다. 파이썬 2.1에서 소개된 이후로 이 메커니즘을 사용하여 다음과 같은 기능이 언어에 도입되었습니다:

기능	선택적 버전	필수적 버전	효과
`__future__.`**`nested_scopes`**	2.1.0b1	2.2	**PEP 227**: 정적으로 중첩된 스코프
`__future__.`**`generators`**	2.2.0a1	2.3	**PEP 255**: 단순 제너레이터
`__future__.`**`division`**	2.2.0a2	3.0	**PEP 238**: 나누기 연산자 변경
`__future__.`**`absolute_import`**	2.5.0a1	3.0	**PEP 328**: 임포트: 복수 줄 및 절대/상대
`__future__.`**`with_statement`**	2.5.0a1	2.6	**PEP 343**: *The "with" Statement*
`__future__.`**`print_function`**	2.6.0a2	3.0	**PEP 3105**: *print*를 함수로 만들기
`__future__.`**`unicode_literals`**	2.6.0a2	3.0	**PEP 3112**: 파이썬 *3000*의 바이트열 리터럴
`__future__.`**`generator_stop`**	3.5.0b1	3.7	**PEP 479**: 제너레이터 내부의 *StopIteration* 처리
`__future__.`**`annotations`**	3.7.0b1	Never[1]	**PEP 563**: *Postponed evaluation of annotations*, **PEP 649**: *Deferred evaluation of annotations using descriptors*

class `__future__.`**`_Feature`**

`__future__.py` 의 각 문장은 다음과 같은 형식입니다:

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                       CompilerFlag)
```

보통, *OptionalRelease* 는 *MandatoryRelease* 보다 작으며, 둘 다 `sys.version_info`와 같은 형태의 5-튜플입니다:

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
 PY_MINOR_VERSION, # the 1; an int
 PY_MICRO_VERSION, # the 0; an int
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
 PY_RELEASE_SERIAL # the 3; an int
)
```

[1] `from __future__ import annotations` was previously scheduled to become mandatory in Python 3.10, but the change was delayed and ultimately canceled. This feature will eventually be deprecated and removed. See **PEP 649** and **PEP 749**.

_Feature.**getOptionalRelease**()

OptionalRelease 는 해당 기능이 승인된 첫 번째 배포를 기록합니다.

_Feature.**getMandatoryRelease**()

MandatoryRelease 가 아직 배포되지 않은 경우, *MandatoryRelease* 는 해당 기능이 언어 일부가 될 배포를 예측합니다.

그렇지 않으면 *MandatoryRelease* 는 기능이 언어 일부가 된 때를 기록합니다; 그 배포와 그 이후의 배포에서, 모듈이 해당 기능을 사용하기 위해 더 퓨처 문을 요구하지 않지만, 그러한 임포트는 계속 사용할 수 있습니다.

MandatoryRelease 는 `None` 일 수도 있습니다. 이는 계획된 기능이 삭제되었거나 아직 결정되지 않았음을 의미합니다.

_Feature.**compiler_flag**

CompilerFlag 은 동적으로 컴파일되는 코드에서 해당 기능을 활성화하기 위해, 내장 함수 *compile()* 의 네 번째 인자로 전달되어야 하는 (비트 필드) 플래그입니다. 이 플래그는 *_Feature* 인스턴스의 *_Feature.compiler_flag* 어트리뷰트에 저장됩니다.

더 보기

future
컴파일러가 퓨처 임포트를 처리하는 방법.

PEP 236 - Back to the __future__
The original proposal for the __future__ mechanism.

29.13 gc — 가비지 수거기 인터페이스

이 모듈은 선택적인 가비지 수거기에 대한 인터페이스를 제공합니다. 수거기를 비활성화하고, 수거 빈도를 조정하며, 디버깅 옵션을 설정하는 기능을 제공합니다. 또한 수거기가 발견했지만 해제할 수 없는 도달 불가능한 객체에 대한 액세스를 제공합니다. 수거기는 파이썬에서 이미 사용된 참조 횟수 추적을 보충하므로, 프로그램이 참조 순환을 만들지 않는다고 확신한다면 수거기를 비활성화 할 수 있습니다. `gc.disable()` 을 호출하여 자동 수거를 비활성화 할 수 있습니다. 누수가 발생하는 프로그램을 디버그 하려면, `gc.set_debug(gc.DEBUG_LEAK)` 을 호출하십시오. 이것은 `gc.DEBUG_SAVEALL`을 포함하므로, 가비지 수거된 객체가 검사를 위해 gc.garbage에 저장되도록 함에 유의하십시오.

gc 모듈은 다음 함수를 제공합니다:

gc.**enable**()

자동 가비지 수거를 활성화합니다.

gc.**disable**()

자동 가비지 수거를 비활성화합니다.

gc.**isenabled**()

자동 수거가 활성화되었으면 `True`를 반환합니다.

gc.**collect**(*generation=2*)

Perform a collection. The optional argument *generation* may be an integer specifying which generation to collect (from 0 to 2). A *ValueError* is raised if the generation number is invalid. The sum of collected objects and uncollectable objects is returned.

Calling `gc.collect(0)` will perform a GC collection on the young generation.

Calling `gc.collect(1)` will perform a GC collection on the young generation and an increment of the old generation.

Calling `gc.collect(2)` or `gc.collect()` performs a full collection

여러 내장형을 위해 유지되는 자유 목록(free list)은 전체 수거나 최고 세대(2)의 수거가 실행될 때마다 지워집니다. 특정 구현(특히 `float`)으로 인해, 일부 자유 목록에서 모든 항목이 해제되지는 않을 수 있습니다.

The effect of calling `gc.collect()` while the interpreter is already performing a collection is undefined.

버전 3.14에서 변경: `generation=1` performs an increment of collection.

`gc.`**`set_debug`**(*flags*)

가비지 수거 디버깅 플래그를 설정합니다. 디버깅 정보가 `sys.stderr`에 기록됩니다. 디버깅을 제어하기 위해 비트 연산을 사용하여 결합할 수 있는 디버깅 플래그 목록은 아래를 참조하십시오.

`gc.`**`get_debug`**()

현재 설정된 디버깅 플래그를 반환합니다.

`gc.`**`get_objects`**(*generation=None*)

Returns a list of all objects tracked by the collector, excluding the list returned. If *generation* is not `None`, return only the objects as follows:

- 0: All objects in the young generation
- 1: No objects, as there is no generation 1 (as of Python 3.14)
- 2: All objects in the old generation

버전 3.8에서 변경: 새로운 *generation* 매개 변수.

버전 3.14에서 변경: Generation 1 is removed

인자 `generation`으로 감사 이벤트 `gc.get_objects`를 발생시킵니다.

`gc.`**`get_stats`**()

인터프리터가 시작된 이후의 수거 통계를 포함하는 세 개의 세대별 딕셔너리의 리스트를 반환합니다. 향후 키 수는 변경될 수 있지만, 현재 각 딕셔너리에는 다음과 같은 항목이 포함됩니다:

- `collections`는 이 세대가 수거된 횟수입니다.
- `collected`는 이 세대 내에서 수거된 총 객체 수입니다.
- `uncollectable`은 이 세대 내에서 수거할 수 없는 (따라서 `garbage` 리스트로 이동된) 것으로 확인된 총 객체 수입니다.

Added in version 3.4.

`gc.`**`set_threshold`**(*threshold0*[, *threshold1*[, *threshold2*]])

가비지 수거 임곗값(수거 빈도)을 설정합니다. *threshold0*을 0으로 설정하면 수거가 비활성화됩니다.

The GC classifies objects into two generations depending on whether they have survived a collection. New objects are placed in the young generation. If an object survives a collection it is moved into the old generation.

In order to decide when to run, the collector keeps track of the number of object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds *threshold0*, collection starts. For each collection, all the objects in the young generation and some fraction of the old generation is collected.

In the free-threaded build, the increase in process memory usage is also checked before running the collector. If the memory usage has not increased by 10% since the last collection and the net number of object allocations has not exceeded 40 times *threshold0*, the collection is not run.

The fraction of the old generation that is collected is **inversely** proportional to *threshold1*. The larger *threshold1* is, the slower objects in the old generation are collected. For the default value of 10, 1% of the old generation is scanned during each collection.

threshold2 is ignored.

See Garbage collector design for more information.

버전 3.14에서 변경: *threshold2* is ignored

gc.**get_count**()

현재 수거 횟수를 (count0, count1, count2)의 튜플로 반환합니다.

gc.**get_threshold**()

현재 수거 임곗값을 (threshold0, threshold1, threshold2)의 튜플로 반환합니다.

gc.**get_referrers**(**objs*)

objs에 있는 것을 직접 참조하는 객체의 리스트를 반환합니다. 이 함수는 가비지 수거를 지원하는 컨테이너만 찾습니다; 다른 객체를 참조하지만, 가비지 수거를 지원하지 않는 확장형은 찾을 수 없습니다.

이미 참조 해제되었지만, 순환에 참여해서 가비지 수거기에 의해 아직 수거되지 않은 객체는 결과 참조자(referrer)에 나열될 수 있음에 유의하십시오. 현재 살아있는 객체만 가져오려면, *get_referrers()*를 호출하기 전에 *collect()*를 호출하십시오.

> **경고**
>
> *get_referrers()*에서 반환된 객체를 사용할 때는, 그중 일부는 아직 생성 중이라서 일시적으로 유효하지 않은 상태일 수 있기 때문에 주의해야 합니다. 디버깅 이외의 목적으로 *get_referrers()*를 사용하지 마십시오.

인자 objs로 감사 이벤트 gc.get_referrers를 발생시킵니다.

gc.**get_referents**(**objs*)

인자로 제공된 객체가 직접 참조하는 객체의 리스트를 반환합니다. 반환된 피 참조자(referent)는 인자의 C 수준 tp_traverse 메서드(있다면)가 방문한 객체이며, 실제로 직접 도달할 수 있는 모든 객체는 아닐 수 있습니다. tp_traverse 메서드는 가비지 수거를 지원하는 객체에서만 지원되며, 순환에 참여하는 객체만 방문하면 됩니다. 그래서, 예를 들어, 인자에서 정수에 직접 도달 할 수 있으면, 해당 정수 객체가 결과 목록에 나타날 수도 그렇지 않을 수도 있습니다.

인자 objs로 감사 이벤트 gc.get_referents를 발생시킵니다.

gc.**is_tracked**(*obj*)

가비지 수거기가 객체를 현재 추적하고 있으면 True를, 그렇지 않으면 False를 반환합니다. 일반적인 규칙으로, 원자 형(atomic type)의 인스턴스는 추적하지 않고, 원자 형이 아닌 인스턴스(컨테이너, 사용자 정의 객체…)는 추적합니다. 그러나 간단한 인스턴스의 가비지 수거기 크기를 줄이기 위해 일부 형별 최적화가 존재할 수 있습니다(예를 들어, 원자적 키와 값만 포함하는 딕셔너리):

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
True
```

Added in version 3.1.

gc.**is_finalized**(*obj*)

주어진 객체가 가비지 수거기에 의해 파이널라이즈 되었으면 True를, 그렇지 않으면 False를 반환합니다.

```
>>> x = None
>>> class Lazarus:
...     def __del__(self):
...         global x
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...         x = self
...
>>> lazarus = Lazarus()
>>> gc.is_finalized(lazarus)
False
>>> del lazarus
>>> gc.is_finalized(x)
True
```

Added in version 3.9.

gc.**freeze**()

가비지 수거기가 추적한 모든 객체를 고정합니다; 그들을 영구 세대(permanent generation)로 이동하고 향후 모든 수거에서 그들을 무시합니다.

If a process will `fork()` without `exec()`, avoiding unnecessary copy-on-write in child processes will maximize memory sharing and reduce overall memory usage. This requires both avoiding creation of freed "holes" in memory pages in the parent process and ensuring that GC collections in child processes won' t touch the `gc_refs` counter of long-lived objects originating in the parent process. To accomplish both, call `gc.disable()` early in the parent process, `gc.freeze()` right before `fork()`, and `gc.enable()` early in child processes.

Added in version 3.7.

gc.**unfreeze**()

영구 세대(permanent generation)의 객체를 고정 해제하고, 가장 나이 든 세대로 되돌립니다.

Added in version 3.7.

gc.**get_freeze_count**()

영구 세대(permanent generation)에 있는 객체 수를 반환합니다.

Added in version 3.7.

다음 변수가 전용 액세스로 제공됩니다 (값을 변경할 수는 있지만, 다시 연결해서는 안 됩니다):

gc.**garbage**

수거기가 발견했지만 해제할 수 없는 도달 불가능한 객체의 리스트 (수거할 수 없는 객체). 파이썬 3.4부터, `NULL`이 아닌 `tp_del` 슬롯이 있는 C 확장형의 인스턴스를 사용할 때를 제외하고, 이 리스트는 대체로 비어 있어야 합니다.

*DEBUG_SAVEALL*이 설정되면, 도달할 수 없는 모든 객체가 해제되지 않고 이 목록에 추가됩니다.

버전 3.2에서 변경: 인터프리터 종료 시 이 목록이 비어 있지 않으면, *ResourceWarning*이 발생하는데, 기본적으로 조용(silent)합니다. *DEBUG_UNCOLLECTABLE*이 설정되면, 추가로 모든 수거 할 수 없는 객체가 인쇄됩니다.

버전 3.4에서 변경: **PEP 442**에 따라, `__del__()` 메서드를 가진 객체는 더는 *gc.garbage*에 들어가지 않습니다.

gc.**callbacks**

수거 전후에 가비지 수거기가 호출할 콜백의 리스트입니다. 콜백은 두 인자로 호출됩니다, *phase*와 *info*.

*phase*는 다음 두 값 중 하나일 수 있습니다:

"start": 가비지 수거를 시작하려고 합니다.

"stop": 가비지 수거가 완료되었습니다.

*info*는 콜백에 추가 정보를 제공하는 딕셔너리입니다. 현재 다음 키가 정의되어 있습니다:

"generation": 수거되는 가장 나이 든 세대.

"collected": *phase*가 "stop" 일 때, 성공적으로 수거된 객체 수.

"uncollectable": *phase*가 "stop" 일 때, 수거할 수 없어서 `garbage`에 들어간 객체 수.

응용 프로그램은 이 리스트에 자체 콜백을 추가 할 수 있습니다. 주요 사용 사례는 다음과 같습니다:

다양한 세대가 수거되는 빈도와 수거에 걸린 시간과 같은 가비지 수거에 대한 통계 수집.

응용 프로그램이 자신의 수거할 수 없는 형이 `garbage`에 나타날 때 식별하고 지울 수 있도록 합니다.

Added in version 3.3.

`set_debug()`와 함께 사용하기 위해 다음 상수가 제공됩니다:

gc.**DEBUG_STATS**

수거 중 통계를 인쇄합니다. 이 정보는 수거 빈도를 조정할 때 유용 할 수 있습니다.

gc.**DEBUG_COLLECTABLE**

발견된 수거 가능한 객체에 대한 정보를 인쇄합니다.

gc.**DEBUG_UNCOLLECTABLE**

발견된 수거 할 수 없는 객체에 대한 정보를 인쇄합니다 (도달 할 수 있지만, 수거기가 해제할 수 없는 객체). 이 객체는 `garbage` 리스트에 추가됩니다.

버전 3.2에서 변경: 인터프리터 종료 시에 `garbage` 리스트가 비어있지 않으면 내용을 인쇄하기도 합니다.

gc.**DEBUG_SAVEALL**

설정하면, 발견된 모든 도달할 수 없는 객체를 해제하는 대신 *garbage*에 추가합니다. 누수가 있는 프로그램을 디버깅하는 데 유용 할 수 있습니다.

gc.**DEBUG_LEAK**

수거기가 누수가 있는 프로그램에 대한 정보를 인쇄하도록 하는 데 필요한 디버깅 플래그 (`DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL`과 같습니다).

29.14 inspect — 라이브 객체 검사

소스 코드: Lib/inspect.py

`inspect` 모듈은 모듈, 클래스, 메서드, 함수, 트레이스백, 프레임 객체 및 코드 객체와 같은 라이브 객체에 대한 정보를 얻는 데 도움이 되는 몇 가지 유용한 함수를 제공합니다. 예를 들어, 클래스의 내용을 검사하거나, 메서드의 소스 코드를 꺼내오거나, 함수의 인자 리스트를 추출하고 포맷하거나, 자세한 트레이스백을 표시하는 데 필요한 모든 정보를 얻는 데 도움이 될 수 있습니다.

이 모듈은 4가지 종류의 주요 서비스를 제공합니다: 형 검사, 소스 코드 가져오기, 클래스와 함수 검사, 인터프리터 스택 검사.

29.14.1 형과 멤버

`getmembers()` 함수는 클래스나 모듈과 같은 객체의 멤버를 검색합니다. 이름이 "is" 로 시작하는 함수는 주로 `getmembers()`의 두 번째 인자로 쓰기에 편리하도록 제공됩니다. 또한 다음과 같은 특수 어트리뷰트를 언제 찾을 수 있는지 결정하는 데 도움이 됩니다 (모듈 어트리뷰트에 관해서는 import-mod-attrs를 참조하세요):

형	어트리뷰트	설명
클래스	__doc__	도큐멘테이션 문자열
	__name__	이 클래스가 정의된 이름
	__qualname__	정규화된 이름
	__module__	이 클래스가 정의된 모듈의 이름
	__type_params__	A tuple containing the type parameters of a generic class

표 2 – 이전 페이지에서 계속

형	어트리뷰트	설명
메서드	__doc__	도큐멘테이션 문자열
	__name__	이 메서드가 정의된 이름
	__qualname__	정규화된 이름
	__func__	메서드의 구현을 포함하는 함수 객체
	__self__	이 메서드가 연결된 인스턴스, 또는 None
	__module__	이 메서드가 정의된 모듈의 이름
함수	__doc__	도큐멘테이션 문자열
	__name__	이 함수가 정의된 이름
	__qualname__	정규화된 이름
	__code__	컴파일된 함수 바이트 코드를 포함하는 코드 객체
	__defaults__	위치나 키워드 매개 변수에 대한 기본값의 튜플
	__kwdefaults__	키워드 전용 매개 변수의 기본값 매핑
	__globals__	이 함수가 정의된 전역 이름 공간
	__builtins__	내장 이름 공간
	__annotations__	매개 변수 이름에서 어노테이션으로의 매핑; "return" 키는 반환 값 어노테이
	__type_params__	A tuple containing the type parameters of a generic function
	__module__	이 함수가 정의된 모듈의 이름
트레이스백	tb_frame	이 수준의 프레임 객체
	tb_lasti	바이트 코드에서 마지막으로 시도한 명령의 인덱스
	tb_lineno	파이썬 소스 코드의 현재 줄 번호
	tb_next	(이 수준에서 호출된) 다음 내부(inner) 트레이스백 객체
프레임	f_back	다음 외부(outer) 프레임 객체 (이 프레임의 호출자)
	f_builtins	이 프레임이 보는 내장 이름 공간
	f_code	이 프레임에서 실행되는 코드 객체
	f_globals	이 프레임이 보는 전역 이름 공간
	f_lasti	바이트 코드에서 마지막으로 시도한 명령의 인덱스
	f_lineno	파이썬 소스 코드의 현재 줄 번호
	f_locals	이 프레임이 보는 지역 이름 공간
	f_generator	returns the generator or coroutine object that owns this frame, or None if the frame is
	f_trace	이 프레임의 추적 함수(tracing function), 또는 None
	f_trace_lines	indicate whether a tracing event is triggered for each source source line
	f_trace_opcodes	indicate whether per-opcode events are requested
	clear()	used to clear all references to local variables
코드	co_argcount	인자 개수 (키워드 전용 인자, * 또는 ** 인자는 포함하지 않습니다)
	co_code	컴파일된 날 바이트 코드의 문자열
	co_cellvars	(포함하는 스코프가 참조하는) 셀 변수 이름의 튜플
	co_consts	바이트 코드에서 사용되는 상수의 튜플
	co_filename	이 코드 객체가 만들어진 파일의 이름
	co_firstlineno	파이썬 소스 코드의 첫 줄 번호
	co_flags	CO_* 플래그의 비트맵, 여기를 더 읽어보십시오
	co_lnotab	줄 번호에서 바이트 코드 인덱스로의 인코딩된 매핑
	co_freevars	(함수의 클로저를 통해 참조되는) 자유 변수(free variables) 이름의 튜플
	co_posonlyargcount	위치 전용 인자의 개수
	co_kwonlyargcount	키워드 전용 인자의 개수 (** 인자는 제외합니다)
	co_name	이 코드 객체가 정의된 이름
	co_qualname	이 코드 객체가 정의된 완전히 정규화된 이름
	co_names	인자와 함수 지역이 아닌 이름의 튜플
	co_nlocals	지역 변수의 개수
	co_stacksize	필요한 가상 기계 스택 공간
	co_varnames	인자와 지역 변수 이름의 튜플
	co_lines()	returns an iterator that yields successive bytecode ranges
	co_positions()	returns an iterator of source code positions for each bytecode instruction
	replace()	returns a copy of the code object with new values
제너레이터	__name__	이름
	__qualname__	정규화된 이름

표 2 – 이전 페이지에서 계속

형	어트리뷰트	설명
	gi_frame	프레임
	gi_running	제너레이터가 실행 중입니까?
	gi_suspended	is the generator suspended?
	gi_code	코드
	gi_yieldfrom	`yield from`에 의해 이터레이트 중인 객체, 또는 `None`
비동기 제너레이터	__name__	이름
	__qualname__	정규화된 이름
	ag_await	어웨이트 중인 객체, 또는 `None`
	ag_frame	프레임
	ag_running	제너레이터가 실행 중입니까?
	ag_code	코드
코루틴	__name__	이름
	__qualname__	정규화된 이름
	cr_await	어웨이트 중인 객체, 또는 `None`
	cr_frame	프레임
	cr_running	코루틴이 실행 중입니까?
	cr_code	코드
	cr_origin	코루틴이 생성된 곳, 또는 `None`. *sys.set_coroutine_origin_tracking_de*
내장	__doc__	도큐멘테이션 문자열
	__name__	이 함수나 메서드의 원래 이름
	__qualname__	정규화된 이름
	__self__	메서드가 연결된 인스턴스, 또는 `None`

버전 3.5에서 변경: `__qualname__`과 `gi_yieldfrom` 어트리뷰트를 제너레이터에 추가합니다.

제너레이터의 `__name__` 어트리뷰트는 이제 코드 이름 대신 함수 이름에서 설정되며, 이제 수정할 수 있습니다.

버전 3.7에서 변경: 코루틴에 `cr_origin` 어트리뷰트를 추가합니다.

버전 3.10에서 변경: 함수에 `__builtins__` 어트리뷰트를 추가합니다.

버전 3.14에서 변경: Add `f_generator` attribute to frames.

`inspect.`**`getmembers`**(*object*[, *predicate*])

이름으로 정렬된 `(name, value)` 쌍의 리스트로 객체(object)의 모든 멤버를 반환합니다. 각 멤버의 `value` 객체로 호출될 선택적 *predicate* 인자가 제공되면, predicate가 참값을 반환하는 멤버만 포함됩니다.

> **참고**
>
> *`getmembers()`*는 인자가 클래스이고 해당 클래스 어트리뷰트가 메타 클래스의 사용자 정의 `__dir__()`에서 나열될 때만 메타 클래스에 정의된 클래스 어트리뷰트를 반환합니다.

`inspect.`**`getmembers_static`**(*object*[, *predicate*])

Return all the members of an object in a list of `(name, value)` pairs sorted by name without triggering dynamic lookup via the descriptor protocol, __getattr__ or __getattribute__. Optionally, only return members that satisfy a given predicate.

> **참고**
>
> *`getmembers_static()`*은 getmembers 가 가져올 수 있는 모든 멤버를 조회하지 못할 수 있으며 (가령 동적으로 만들어진 어트리뷰트), getmembers 가 가져올 수 없는 멤버를 찾을 수 있습니다 (가령 AttributeError를 발생시키는 디스크립터). 어떤 경우에는 인스턴스 멤버 대신 디스크립터 객체를 반환할 수도 있습니다.

Added in version 3.11.

inspect.**getmodulename**(*path*)

감싸고 있는 패키지 이름 없이, 파일 경로(*path*)가 가리키는 모듈의 이름을 반환합니다. 파일 확장자는 `importlib.machinery.all_suffixes()`의 모든 항목에 대해 점검됩니다. 일치하면, 확장명이 제거된 최종 경로 구성 요소가 반환됩니다. 그렇지 않으면, `None`이 반환됩니다.

이 함수는 오직 실제 파이썬 모듈로 의미 있는 이름만 반환합니다. 잠재적으로 파이썬 패키지를 가리키는 경로는 여전히 `None`을 반환합니다.

버전 3.3에서 변경: 이 함수는 `importlib`에 직접 기반합니다.

inspect.**ismodule**(*object*)

객체가 모듈이면 `True`를 반환합니다.

inspect.**isclass**(*object*)

객체가 (내장이거나 파이썬 코드로 만든) 클래스이면 `True`를 반환합니다.

inspect.**ismethod**(*object*)

객체가 파이썬으로 작성된 연결된(bound) 메서드면 `True`를 반환합니다.

inspect.**ispackage**(*object*)

Return `True` if the object is a *package*.

Added in version 3.14.

inspect.**isfunction**(*object*)

객체가 파이썬 함수이면 `True`를 반환합니다. 람다 표현식으로 만든 함수를 포함합니다.

inspect.**isgeneratorfunction**(*object*)

객체가 파이썬 제너레이터 함수이면 `True`를 반환합니다.

버전 3.8에서 변경: 래핑 된 함수가 파이썬 제너레이터 함수일 때 `functools.partial()`로 래핑 된 함수는 이제 `True`를 반환합니다.

버전 3.13에서 변경: 래핑 된 함수가 파이썬 제너레이터 함수일 때 `functools.partialmethod()`로 래핑 된 함수는 이제 `True`를 반환합니다.

inspect.**isgenerator**(*object*)

객체가 제너레이터이면 `True`를 반환합니다.

inspect.**iscoroutinefunction**(*object*)

객체가 코루틴 함수 (`async def` 문법으로 정의된 함수), 코루틴 함수를 감싸는 `functools.partial()` 또는 `markcoroutinefunction()`으로 표시된 동기 함수이면 `True`를 반환합니다.

Added in version 3.5.

버전 3.8에서 변경: 래핑 된 함수가 코루틴 함수일 때 `functools.partial()`로 래핑 된 함수는 이제 `True`를 반환합니다.

버전 3.12에서 변경: Sync functions marked with `markcoroutinefunction()` now return `True`.

버전 3.13에서 변경: 래핑 된 함수가 코루틴 함수일 때 `functools.partialmethod()`로 래핑 된 함수는 이제 `True`를 반환합니다.

inspect.**markcoroutinefunction**(*func*)

Decorator to mark a callable as a *coroutine function* if it would not otherwise be detected by `iscoroutinefunction()`.

This may be of use for sync functions that return a *coroutine*, if the function is passed to an API that requires `iscoroutinefunction()`.

When possible, using an `async def` function is preferred. Also acceptable is calling the function and testing the return with `iscoroutine()`.

Added in version 3.12.

inspect.**iscoroutine**(*object*)

객체가 `async def` 함수가 만든 코루틴이면 `True`를 반환합니다.

Added in version 3.5.

inspect.**isawaitable**(*object*)

`await` 표현식에서 객체를 사용할 수 있으면 `True`를 반환합니다.

제너레이터 기반 코루틴을 일반 제너레이터와 구별하는 데 사용할 수도 있습니다:

```
import types

def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

Added in version 3.5.

inspect.**isasyncgenfunction**(*object*)

객체가 비동기 제너레이터 함수이면 `True`를 반환합니다, 예를 들면:

```
>>> async def agen():
...     yield 1
...
>>> inspect.isasyncgenfunction(agen)
True
```

Added in version 3.6.

버전 3.8에서 변경: 래핑 된 함수가 비동기 제너레이터 함수일 때 *functools.partial()*로 래핑 된 함수는 이제 `True`를 반환합니다.

버전 3.13에서 변경: 래핑 된 함수가 코루틴 함수일 때 *functools.partialmethod()*로 래핑 된 함수는 이제 `True`를 반환합니다.

inspect.**isasyncgen**(*object*)

객체가 *asynchronous generator* 함수가 만든 비동기 제너레이터 이터레이터이면 `True`를 반환합니다.

Added in version 3.6.

inspect.**istraceback**(*object*)

객체가 트레이스백이면 `True`를 반환합니다.

inspect.**isframe**(*object*)

객체가 프레임이면 `True`를 반환합니다.

inspect.**iscode**(*object*)

객체가 코드이면 `True`를 반환합니다.

inspect.**isbuiltin**(*object*)

객체가 내장 함수나 연결된(bound) 내장 메서드이면 `True`를 반환합니다.

inspect.**ismethodwrapper**(*object*)

객체의 형이 *MethodWrapperType*이면 `True`를 반환합니다.

These are instances of *MethodWrapperType*, such as `__str__()`, `__eq__()` and `__repr__()`.

Added in version 3.11.

inspect.**isroutine**(*object*)

객체가 사용자 정의이거나 내장 함수나 메서드이면 True를 반환합니다.

inspect.**isabstract**(*object*)

객체가 추상 베이스 클래스이면 True를 반환합니다.

inspect.**ismethoddescriptor**(*object*)

객체가 메서드 디스크립터이면 True를 반환하지만, *ismethod()*, *isclass()*, *isfunction()* 또는 *isbuiltin()*이 참일 때는 그렇지 않습니다.

예를 들어, 이것은 int.__add__에 대해서 참입니다. 이 테스트를 통과한 객체에는 __get__() 메서드가 있지만 __set__() 메서드나 __delete__() 메서드는 없습니다. 하지만 그 외의 어트리뷰트 집합은 달라집니다. *__name__* 어트리뷰트는 보통 존재하고, *__doc__*도 종종 그렇습니다.

다른 테스트 중 하나를 통과하는 디스크립터로 구현된 메서드는 *ismethoddescriptor()* 테스트에서 False를 반환합니다. 단순히 다른 테스트가 더 많은 것을 약속하기 때문입니다 – 예를 들어, 객체가 *ismethod()*를 통과할 때 __func__ 어트리뷰트(등)가 있다고 기대할 수 있습니다.

버전 3.13에서 변경: This function no longer incorrectly reports objects with __get__() and __delete__(), but not __set__(), as being method descriptors (such objects are data descriptors, not method descriptors).

inspect.**isdatadescriptor**(*object*)

객체가 데이터 디스크립터이면 True를 반환합니다.

데이터 디스크립터에는 __set__이나 __delete__ 메서드가 있습니다. 예는 (파이썬으로 정의한) 프로퍼티, getset 및 멤버입니다. 뒤의 두 가지는 C로 정의되며 해당 형에 대해 더 구체적인 테스트가 있으며, 이는 다른 파이썬 구현에서도 지원됩니다. 일반적으로 데이터 디스크립터는 *__name__*과 __doc__ 어트리뷰트를 갖지만 (프로퍼티, getet 및 멤버는 이 두 어트리뷰트를 모두 갖습니다), 이것이 보장되지는 않습니다.

inspect.**isgetsetdescriptor**(*object*)

객체가 getset 디스크립터이면 True를 반환합니다.

getset은 PyGetSetDef 구조체를 통해 확장 모듈에서 정의된 어트리뷰트입니다. 이러한 형이 없는 파이썬 구현에서, 이 메서드는 항상 False를 반환합니다.

inspect.**ismemberdescriptor**(*object*)

객체가 멤버 디스크립터이면 True를 반환합니다.

멤버 디스크립터는 PyMemberDef 구조체를 통해 확장 모듈에서 정의된 어트리뷰트입니다. 이러한 형이 없는 파이썬 구현에서, 이 메서드는 항상 False를 반환합니다.

29.14.2 소스 코드 가져오기

inspect.**getdoc**(*object*)

*cleandoc()*으로 정리된 객체의 독스트링을 가져옵니다. 객체가 독스트링을 제공하지 않고 객체가 클래스, 메서드, 프로퍼티 또는 디스크립터이면, 상속 계층 구조에서 독스트링을 가져옵니다. 독스트링이 유효하지 않거나 누락되었으면 None을 반환합니다.

버전 3.5에서 변경: 이제 재정의되지 않으면 독스트링이 상속됩니다.

inspect.**getcomments**(*object*)

객체의 소스 코드 바로 앞(클래스, 함수 또는 메서드일 때)이나 파이썬 소스 파일의 최상단 (객체가 모듈일 때) 주석 줄들을 단일 문자열로 반환합니다. 객체의 소스 코드를 사용할 수 없으면, None을 반환합니다. 객체가 C나 대화식 셸에서 정의될 때 이런 일이 일어날 수 있습니다.

inspect.**getfile**(*object*)

객체가 정의된 (텍스트나 바이너리) 파일의 이름을 반환합니다. 객체가 내장 모듈, 클래스 또는 함수이면 *TypeError*로 실패합니다.

inspect.**getmodule**(*object*)

객체가 정의된 모듈을 추측합니다. 모듈이 결정될 수 없으면 None을 반환합니다.

inspect.**getsourcefile**(*object*)

객체가 정의된 파이썬 소스 파일의 이름을 반환합니다. 또는 소스를 얻을 수 있는 방법이 없으면 None을 반환합니다. 객체가 내장 모듈, 클래스 또는 함수이면 *TypeError*로 실패합니다.

inspect.**getsourcelines**(*object*)

객체의 소스 줄의 리스트와 시작 줄 번호를 반환합니다. 인자는 모듈, 클래스, 메서드, 함수, 트레이스백, 프레임 또는 코드 객체일 수 있습니다. 소스 코드는 객체에 해당하는 줄 리스트로 반환되며 줄 번호는 원본 소스 파일에서 첫 번째 코드 줄이 발견되는 위치를 나타냅니다. 소스 코드를 가져올 수 없으면 *OSError*가 발생합니다. 객체가 내장 모듈, 클래스 또는 함수면 *TypeError*가 발생합니다.

버전 3.3에서 변경: *IOError* 대신 *OSError*가 발생합니다. 이제 IOError는 OSError의 별칭입니다.

inspect.**getsource**(*object*)

객체의 소스 코드 텍스트를 반환합니다. 인자는 모듈, 클래스, 메서드, 함수, 트레이스백, 프레임 또는 코드 객체일 수 있습니다. 소스 코드는 단일 문자열로 반환됩니다. 소스 코드를 가져올 수 없으면 *OSError*가 발생합니다. 객체가 내장 모듈, 클래스 또는 함수면 *TypeError*가 발생합니다.

버전 3.3에서 변경: *IOError* 대신 *OSError*가 발생합니다. 이제 IOError는 OSError의 별칭입니다.

inspect.**cleandoc**(*doc*)

코드 블록과 일치하도록 들여쓰기 된 독스트링에서 들여쓰기를 정리합니다.

모든 선행 공백이 첫 번째 줄에서 제거됩니다. 두 번째 줄부터 균일하게 제거할 수 있는 선행 공백이 제거됩니다. 시작과 끝의 빈 줄은 그다음에 제거됩니다. 또한, 모든 탭이 스페이스로 확장됩니다.

29.14.3 Signature 객체로 콜러블 검사하기

Added in version 3.3.

Signature 객체는 콜러블 객체의 호출 서명과 반환 값 어노테이션을 나타냅니다. Signature 객체를 가져오려면, signature() 함수를 사용하십시오.

inspect.**signature**(*callable*, *, *follow_wrapped=True*, *globals=None*, *locals=None*, *eval_str=False*, *annotation_format=Format.VALUE*)

주어진 *callable*에 대한 *Signature* 객체를 반환합니다:

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b: int, **kwargs)'

>>> str(sig.parameters['b'])
'b: int'

>>> sig.parameters['b'].annotation
<class 'int'>
```

일반 함수와 클래스에서 *functools.partial()* 객체에 이르기까지 광범위한 파이썬 콜러블을 받아들입니다.

If some of the annotations are strings (e.g., because from __future__ import annotations was used), *signature()* will attempt to automatically un-stringize the annotations using *annotationlib.get_annotations()*. The *globals*, *locals*, and *eval_str* parameters are passed into annotationlib.get_annotations() when resolving the annotations; see the documentation for annotationlib.get_annotations() for instructions on how to use these parameters. A member of the *annotationlib.Format* enum can be passed to the *annotation_format* parameter to control the format of the returned annotations. For example, use annotation_format=annotationlib.Format.STRING to return annotations in string format.

Raises `ValueError` if no signature can be provided, and `TypeError` if that type of object is not supported. Also, if the annotations are stringized, and *eval_str* is not false, the `eval()` call(s) to un-stringize the annotations in `annotationlib.get_annotations()` could potentially raise any kind of exception.

A slash (/) in the signature of a function denotes that the parameters prior to it are positional-only. For more info, see the FAQ entry on positional-only parameters.

버전 3.5에서 변경: *follow_wrapped* 매개 변수를 추가했습니다. 구체적으로 *callable*의 서명을 가져오려면 `False`를 전달하십시오 (데코레이트 된 콜러블의 래핑을 풀기 위해 `callable.__wrapped__`를 사용하지 않게 됩니다).

버전 3.10에서 변경: The *globals*, *locals*, and *eval_str* parameters were added.

버전 3.14에서 변경: The *annotation_format* parameter was added.

> **참고**
>
> 특정 파이썬 구현에서 일부 콜러블은 검사할 수 없습니다. 예를 들어, CPython에서, C로 정의된 일부 내장 함수는 인자에 대한 메타 데이터를 제공하지 않습니다.

CPython 구현 상세: If the passed object has a `__signature__` attribute, we may use it to create the signature. The exact semantics are an implementation detail and are subject to unannounced changes. Consult the source code for current semantics.

class `inspect.`**`Signature`**(*parameters=None, *, return_annotation=Signature.empty*)

`Signature` 객체는 함수의 호출 서명과 반환 값 어노테이션을 나타냅니다. 함수가 받아들이는 각 매개 변수에 대해 `parameters` 컬렉션에 `Parameter` 객체를 저장합니다.

선택적 *parameters* 인자는 `Parameter` 객체의 시퀀스이며, 중복된 이름을 가진 매개 변수가 없는지, 매개 변수가 올바른 순서인지 (즉, 위치 전용이 먼저 온 후, 위치-키워드가 그다음에 오는지), 기본값이 있는 매개 변수가 그렇지 않은 매개 변수 뒤에 오는지 검사합니다.

선택적 *return_annotation* 인자는 임의의 파이썬 객체일 수 있습니다. 콜러블의 "반환 값" 어노테이션을 나타냅니다.

`Signature` 객체는 불변(*immutable*) 입니다. 수정된 사본을 만들려면 `Signature.replace()`나 `copy.replace()`를 사용하십시오.

버전 3.5에서 변경: `Signature` 객체는 이제 피클 가능하고 해시 가능합니다.

empty

반환 값 어노테이션이 없음을 지정하는 특수 클래스 수준 마커입니다.

parameters

매개 변수 이름에서 해당 `Parameter` 객체로의 순서 있는 매핑. 키워드 전용 매개 변수를 포함하여, 매개 변수는 엄격한 정의 순서대로 나타납니다.

버전 3.7에서 변경: 실제로 파이썬 3에서 항상 유지되었습니다만, 파이썬은 버전 3.7부터 키워드 전용 매개 변수의 선언 순서를 유지한다는 것을 명시적으로 보장합니다.

return_annotation

콜러블의 "반환 값" 어노테이션. 콜러블에 "반환 값" 어노테이션이 없으면, 이 어트리뷰트는 `Signature.empty`로 설정됩니다.

bind(**args, **kwargs*)

위치와 키워드 인자에서 매개 변수로의 매핑을 만듭니다. `*args`와 `**kwargs`가 서명과 일치하면 `BoundArguments`를 반환하고, 그렇지 않으면 `TypeError`를 발생시킵니다.

bind_partial(**args, **kwargs*)

`Signature.bind()`와 같은 방식으로 작동하지만, 일부 필수 인자를 생략 할 수 있습니다 (`functools.partial()` 동작을 흉내 냅니다). `BoundArguments`를 반환하거나, 전달된 인자가 서명과 일치하지 않으면 `TypeError`를 발생시킵니다.

replace(*[, *parameters*][, *return_annotation*])

`replace()`가 호출된 인스턴스를 기반으로 새 `Signature` 인스턴스를 만듭니다. 다른 *parameters* 나 *return_annotation* 또는 둘 모두를 전달하여 기반 서명의 해당 속성을 재정의할 수 있습니다. 복사된 `Signature`에서 `return_annotation`을 제거하려면, `Signature.empty`를 전달하십시오.

```
>>> def test(a, b):
...     pass
...
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new return anno")
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```

`Signature` objects are also supported by the generic function `copy.replace()`.

format(*, *max_width=None*, *quote_annotation_strings=True*)

`Signature` 객체의 문자열 표현을 만듭니다.

If *max_width* is passed, the method will attempt to fit the signature into lines of at most *max_width* characters. If the signature is longer than *max_width*, all parameters will be on separate lines.

If *quote_annotation_strings* is False, *annotations* in the signature are displayed without opening and closing quotation marks if they are strings. This is useful if the signature was created with the `STRING` format or if `from __future__ import annotations` was used.

Added in version 3.13.

버전 3.14에서 변경: The *unquote_annotations* parameter was added.

classmethod from_callable(*obj*, *, *follow_wrapped=True*, *globals=None*, *locals=None*, *eval_str=False*)

주어진 콜러블 *obj*에 대한 `Signature` (또는 그 서브 클래스) 객체를 반환합니다:

이 메서드는 `Signature`의 서브 클래싱을 단순화합니다:

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(sum)
assert isinstance(sig, MySignature)
```

Its behavior is otherwise identical to that of `signature()`.

Added in version 3.5.

버전 3.10에서 변경: The *globals*, *locals*, and *eval_str* parameters were added.

class inspect.**Parameter**(*name*, *kind*, *, *default=Parameter.empty*, *annotation=Parameter.empty*)

`Parameter` 객체는 불변(*immutable*)입니다. `Parameter` 객체를 수정하는 대신, 수정된 사본을 만들려면 `Parameter.replace()`나 `copy.replace()`를 사용하십시오.

버전 3.5에서 변경: Parameter 객체는 이제 피클 가능하고 해시 가능합니다.

empty

기본값과 어노테이션이 없음을 지정하는 특수 클래스 수준 마커.

name

문자열로 표현한 매개 변수의 이름. 이름은 유효한 파이썬 식별자여야 합니다.

CPython은 컴프리헨션과 제너레이터 표현식을 구현하는 데 사용되는 코드 객체에서 `.0` 형식의 묵시적 매개 변수 이름을 생성합니다.

버전 3.6에서 변경: 이 매개 변수 이름은 이제 이 모듈에서 `implicit0`과 같은 이름으로 노출됩니다.

default

매개 변수의 기본값. 매개 변수에 기본값이 없으면, 이 어트리뷰트는 *Parameter.empty*로 설정됩니다.

annotation

매개 변수의 어노테이션. 매개 변수에 어노테이션이 없으면, 이 어트리뷰트는 *Parameter.empty*로 설정됩니다.

kind

인자 값이 매개 변수에 연결되는 방법을 설명합니다. 가능한 값은 (`Parameter.KEYWORD_ONLY` 처럼) *Parameter*를 통해 액세스할 수 있고, 다음과 같은 순서로 비교와 순서를 지원합니다:

이름	의미
POSITIONAL_ONLY	위치 인자로만 값을 제공해야 합니다. 위치 전용 매개 변수는 파이썬 함수 정의에서 / 항목 (있다면) 앞에 나오는 매개 변수입니다.
POSITIONAL_OR_KEYWORD	값은 키워드나 위치 인자로 제공될 수 있습니다 (이것이 파이썬으로 구현된 함수의 표준 연결 동작입니다).
VAR_POSITIONAL	다른 매개 변수에 연결되지 않은 위치 인자의 튜플. 이것은 파이썬 함수 정의의 `*args` 매개 변수에 해당합니다.
KEYWORD_ONLY	키워드 인자로만 값을 제공해야 합니다. 키워드 전용 매개 변수는 파이썬 함수 정의에서 `*`나 `*args` 항목 다음에 나오는 매개 변수입니다.
VAR_KEYWORD	다른 매개 변수에 연결되지 않은 키워드 인자의 딕셔너리. 이것은 파이썬 함수 정의의 `**kwargs` 매개 변수에 해당합니다.

예: 기본값이 없는 모든 키워드 전용 인자를 인쇄합니다:

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...                        param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

`kind.`**description**

*Parameter.kind*의 열거형 값을 설명합니다.

Added in version 3.8.

예: 모든 인자의 설명을 인쇄합니다:

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     print(param.kind.description)
positional or keyword
positional or keyword
keyword-only
keyword-only
```

replace(*[, *name*][, *kind*][, *default*][, *annotation*])

replace가 호출된 인스턴스를 기반으로 새 *Parameter* 인스턴스를 만듭니다. Parameter 어트리뷰트를 재정의하려면, 해당 인자를 전달하십시오. Parameter에서 기본값이나 어노테이션, 또는 둘 다 제거하려면 *Parameter.empty*를 전달하십시오.

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
>>> str(param)
'foo=42'

>>> str(param.replace()) # 'param'의 얕은 사본을 만듭니다
'foo=42'

>>> str(param.replace(default=Parameter.empty, annotation='spam'))
"foo: 'spam'"
```

Parameter objects are also supported by the generic function *copy.replace()*.

버전 3.4에서 변경: 파이썬 3.3에서, kind가 POSITIONAL_ONLY로 설정되었을 때 *Parameter* 객체의 name을 None으로 설정할 수 있었습니다. 이는 더는 허용되지 않습니다.

class inspect.**BoundArguments**

*Signature.bind()*나 *Signature.bind_partial()* 호출의 결과. 인자에서 함수의 매개 변수로의 매핑을 보관합니다.

arguments

매개 변수 이름에서 인자 값으로의 가변 매핑. 명시적으로 연결된 인자만 포함합니다. *arguments*의 변경 사항은 *args*와 *kwargs*에 반영됩니다.

인자 처리 목적으로 *Signature.parameters*와 함께 사용해야 합니다.

> **참고**
>
> *Signature.bind()*나 *Signature.bind_partial()*이 기본값에 의존하는 인자는 건너뜁니다. 그러나, 필요하다면 *BoundArguments.apply_defaults()*를 사용하여 추가하십시오.

버전 3.9에서 변경: *arguments*는 이제 *dict* 형입니다. 이전에는, *collections.OrderedDict* 형이었습니다.

args

위치 인자 값의 튜플. *arguments* 어트리뷰트에서 동적으로 계산됩니다.

kwargs

키워드 인자 값의 딕셔너리. *arguments* 어트리뷰트에서 동적으로 계산됩니다. 위치적으로 전달될 수 있는 인자는 대신 *args*에 포함됩니다.

signature

부모 *Signature* 객체에 대한 참조.

apply_defaults()

누락된 인자에 대한 기본값을 설정합니다.

가변 위치 인자(*args)의 기본값은 빈 튜플입니다.

가변 변수 키워드 인자(**kwargs)의 기본값은 빈 딕셔너리입니다.

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> ba.arguments
{'a': 'spam', 'b': 'ham', 'args': ()}
```

Added in version 3.5.

*args*와 *kwargs* 프로퍼티를 사용하여 함수를 호출할 수 있습니다:

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

더 보기

PEP 362 - 함수 Signature 객체.
자세한 명세, 구현 세부 사항 및 예

29.14.4 클래스와 함수

inspect.**getclasstree**(*classes, unique=False*)

주어진 클래스 리스트를 중첩된 리스트의 계층 구조로 배치합니다. 중첩된 리스트가 나타나면, 리스트 바로 앞에 나오는 항목의 클래스에서 파생된 클래스를 포함합니다. 각 항목은 클래스와 베이스 클래스의 튜플을 포함하는 2-튜플입니다. *unique* 인자가 참이면, 주어진 리스트의 각 클래스가 반환된 구조에 정확히 하나의 항목으로 나타납니다. 그렇지 않으면, 다중 상속을 사용하는 클래스와 그 자식들이 여러 번 나타납니다.

inspect.**getfullargspec**(*func*)

파이썬 함수 매개 변수의 이름과 기본값을 가져옵니다. 네임드 튜플이 반환됩니다:

FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations)

*args*는 위치 매개 변수 이름의 리스트입니다. *varargs*는 * 매개 변수의 이름이거나 임의의 위치 인자가 허용되지 않으면 None 입니다. *varkw*는 ** 매개 변수의 이름이거나 임의의 키워드 인자가 허용되지 않으면 None 입니다. *defaults*는 마지막 *n* 개의 위치 매개 변수에 해당하는 기본 인자 값의 *n*-튜플이거나, 이러한 기본값이 정의되지 않으면 None 입니다. *kwonlyargs*는 선언 순서를 따르는 키워드 전용 매개 변수 이름 리스트입니다. *kwonlydefaults*는 *kwonlyargs*의 매개 변수 이름에서 인자가 제공되지 않을 때 사용되는 기본값으로의 딕셔너리 매핑입니다. *annotations*는 매개 변수 이름에서 어노테이션으로의 딕셔너리 매핑입니다. 특수키 "return"은 함수 반환 값 어노테이션(있다면)을 보고하는 데 사용됩니다.

*signature()*와 *Signature* 객체가 콜러블 내부 검사에 권장되는 API를 제공하고, 확장 모듈 API에서 종종 등장하는 추가 동작(위치 전용 인자와 같은)을 지원함에 유의하십시오. 이 함수는 주로 파이썬 2 inspect 모듈 API와의 호환성을 유지해야 하는 코드에서 사용하기 위해 유지됩니다.

버전 3.4에서 변경: 이 함수는 이제 *signature()*를 기반으로 하지만, 여전히 __wrapped__ 어트리뷰트를 무시하고 연결된(bound) 메서드의 서명 출력에 이미 연결된 첫 번째 매개 변수를 포함합니다.

버전 3.6에서 변경: This method was previously documented as deprecated in favour of *signature()* in Python 3.5, but that decision has been reversed in order to restore a clearly supported standard interface for single-source Python 2/3 code migrating away from the legacy getargspec() API.

버전 3.7에서 변경: 실제로 파이썬 3에서 항상 유지되었습니다만, 파이썬은 버전 3.7부터 키워드 전용 매개 변수의 선언 순서를 유지한다는 것을 명시적으로 보장합니다.

inspect.**getargvalues**(*frame*)

특정 프레임으로 전달된 인자에 대한 정보를 얻습니다. 네임드 튜플 ArgInfo(args, varargs,

keywords, locals) 가 반환됩니다. *args*는 인자 이름의 리스트입니다. *varargs*와 *keywords*는 * 와 ** 인자의 이름이거나 None 입니다. *locals*는 주어진 프레임의 지역 딕셔너리입니다.

> **참고**
>
> 이 함수는 실수로 파이썬 3.5에서 폐지된 것으로 표시되었습니다.

inspect.**formatargvalues**(*args*[, *varargs*, *varkw*, *locals*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*])

getargvalues() 가 반환한 4개의 값으로 예쁜 인자 명세를 포맷합니다. format* 인자는 해당 이름과 값을 문자열로 변환하기 위해 호출되는 선택적 포매팅 함수입니다.

> **참고**
>
> 이 함수는 실수로 파이썬 3.5에서 폐지된 것으로 표시되었습니다.

inspect.**getmro**(*cls*)

클래스 cls의 베이스 클래스의 튜플(cls를 포함합니다)을 메서드 결정 순서로 반환합니다. 이 튜플에는 클래스가 두 번 이상 나타나지 않습니다. 메서드 결정 순서는 cls의 형에 따라 다릅니다. 매우 독특한 사용자 정의 메타 형을 사용하지 않는 한, cls는 튜플의 첫 번째 요소가 됩니다.

inspect.**getcallargs**(*func*, /, **args*, ***kwds*)

*args*와 *kwds*를 마치 이들로 호출된 것처럼 파이썬 함수나 메서드 *func*의 인자 이름에 연결합니다. 연결된 메서드의 경우, 첫 번째 인자(일반적으로 self 라고 합니다)도 해당 인스턴스에 연결합니다. 인자 이름(있다면, * 와 ** 인자의 이름도 포함합니다)을 *args*와 *kwds*의 값으로 매핑하는 딕셔너리가 반환됩니다. *func*를 잘못 호출하는 경우, 즉 호환되지 않는 서명으로 인해 func(*args, **kwds) 가 예외를 발생시키게 될 때마다, 같은 형의 예외가 같거나 유사한 메시지로 발생합니다. 예를 들면:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
...
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
True
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() missing 1 required positional argument: 'a'
```

Added in version 3.2.

버전 3.5부터 폐지됨: 대신 *Signature.bind()* 와 *Signature.bind_partial()* 을 사용하십시오.

inspect.**getclosurevars**(*func*)

파이썬 함수나 메서드 *func*에 있는 외부 이름 참조에서 현재 값으로의 매핑을 얻습니다. 네임드 튜플 ClosureVars(nonlocals, globals, builtins, unbound) 가 반환됩니다. *nonlocals*는 참조된 이름을 어휘 클로저(closure) 변수로, *globals*는 함수의 모듈 전역으로, *builtins*는 함수 바디에서 볼 수 있는 내장으로 매핑합니다. *unbound*는 현재 모듈 전역과 내장에서 전혀 결정할 수 없는 함수에서 참조된 이름 집합입니다.

*func*가 파이썬 함수나 메서드가 아니면 *TypeError*가 발생합니다.

Added in version 3.3.

inspect.**unwrap**(*func*, *, *stop=None*)

*func*로 래핑 된 객체를 가져옵니다. 체인의 마지막 객체를 반환하는 `__wrapped__` 어트리뷰트의 체인을 따라갑니다.

*stop*은 래퍼 체인의 객체를 유일한 인자로 받아들이는 선택적 콜백으로, 콜백이 참값을 반환할 때 언래핑을 조기에 종료 할 수 있도록 합니다. 콜백이 참값을 반환하지 않으면, 체인의 마지막 객체가 평소처럼 반환됩니다. 예를 들어, `signature()`는 이것을 사용하여 체인에 있는 객체에 `__signature__` 어트리뷰트가 정의되면 언 래핑을 중지합니다.

순환이 발견되면 `ValueError`가 발생합니다.

Added in version 3.4.

inspect.**get_annotations**(*obj*, *, *globals=None*, *locals=None*, *eval_str=False*, *format=annotationlib.Format.VALUE*)

Compute the annotations dict for an object.

This is an alias for `annotationlib.get_annotations()`; see the documentation of that function for more information.

> **조심**
>
> This function may execute arbitrary code contained in annotations. See *Security implications of introspecting annotations* for more information.

Added in version 3.10.

버전 3.14에서 변경: This function is now an alias for `annotationlib.get_annotations()`. Calling it as `inspect.get_annotations` will continue to work.

29.14.5 인터프리터 스택

Some of the following functions return `FrameInfo` objects. For backwards compatibility these objects allow tuple-like operations on all attributes except `positions`. This behavior is considered deprecated and may be removed in the future.

class inspect.**FrameInfo**

frame

The frame object that the record corresponds to.

filename

The file name associated with the code being executed by the frame this record corresponds to.

lineno

The line number of the current line associated with the code being executed by the frame this record corresponds to.

function

The function name that is being executed by the frame this record corresponds to.

code_context

A list of lines of context from the source code that' s being executed by the frame this record corresponds to.

index

The index of the current line being executed in the `code_context` list.

positions

A `dis.Positions` object containing the start line number, end line number, start column offset, and end column offset associated with the instruction being executed by the frame this record corresponds to.

버전 3.5에서 변경: *tuple* 대신 네임드 튜플을 반환합니다.

버전 3.11에서 변경: `FrameInfo` is now a class instance (that is backwards compatible with the previous *named tuple*).

class inspect.**Traceback**

filename

The file name associated with the code being executed by the frame this traceback corresponds to.

lineno

The line number of the current line associated with the code being executed by the frame this traceback corresponds to.

function

The function name that is being executed by the frame this traceback corresponds to.

code_context

A list of lines of context from the source code that' s being executed by the frame this traceback corresponds to.

index

The index of the current line being executed in the *code_context* list.

positions

A *dis.Positions* object containing the start line number, end line number, start column offset, and end column offset associated with the instruction being executed by the frame this traceback corresponds to.

버전 3.11에서 변경: `Traceback` is now a class instance (that is backwards compatible with the previous *named tuple*).

참고

이러한 함수가 반환하는 프레임 레코드의 첫 번째 요소에서 발견되는 것처럼, 프레임 객체에 대한 참조를 유지하면 프로그램이 참조 순환을 만들 수 있습니다. 일단 참조 순환이 생성되면, 파이썬의 선택적 순환 검출기가 활성화되어 있어도, 순환을 형성하는 객체에서 액세스 할 수 있는 모든 객체의 수명이 훨씬 더 길어질 수 있습니다. 이러한 순환을 만들어야만 하면, 명시적으로 끊어서 객체의 지연된 파괴와 메모리 소비 증가를 피하는 것이 중요합니다.

순환 감지기가 이를 잡기는 하겠지만, `finally` 절에서 순환을 제거하여 프레임(과 지역 변수)의 파괴를 결정적(deterministic)으로 만들 수 있습니다. 파이썬을 컴파일할 때나 *gc.disable()*을 사용해서 순환 감지기를 비활성화했을 때도 중요합니다. 예를 들면:

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # frame으로 뭔가 합니다
    finally:
        del frame
```

프레임을 계속 유지하려면 (예를 들어 나중에 트레이스백을 인쇄하려고) `frame.clear()` 메서드를 사용하여 참조 순환을 끊을 수도 있습니다.

이 함수들 대부분이 지원하는 선택적 *context* 인자는 반환할 문맥(context) 줄 수를 지정합니다. 이 줄들은 현재 줄을 중심으로 합니다.

inspect.**getframeinfo**(*frame*, *context=1*)

프레임이나 트레이스백 객체에 대한 정보를 얻습니다. *Traceback* 객체가 반환됩니다.

버전 3.11에서 변경: A *Traceback* object is returned instead of a named tuple.

inspect.**getouterframes**(*frame*, *context=1*)

frame과 모든 외부(outer) 프레임에 대한 `FrameInfo` 객체 리스트를 얻습니다. 이 프레임들은 *frame*을 만들도록 한 호출을 나타냅니다. 반환된 리스트의 첫 번째 항목은 *frame*을 나타냅니다; 마지막 항목은 *frame*의 스택에서 가장 바깥쪽 호출을 나타냅니다.

버전 3.5에서 변경: 네임드 튜플 `FrameInfo(frame, filename, lineno, function, code_context, index)`의 리스트가 반환됩니다.

버전 3.11에서 변경: A list of `FrameInfo` objects is returned.

inspect.**getinnerframes**(*traceback*, *context=1*)

traceback의 프레임과 모든 내부(inner) 프레임에 대한 `FrameInfo` 객체 리스트를 얻습니다. 이 프레임들은 *frame*의 결과로 만들어진 호출을 나타냅니다. 리스트의 첫 번째 항목은 *traceback*을 나타냅니다; 마지막 항목은 예외가 발생한 위치를 나타냅니다.

버전 3.5에서 변경: 네임드 튜플 `FrameInfo(frame, filename, lineno, function, code_context, index)`의 리스트가 반환됩니다.

버전 3.11에서 변경: A list of `FrameInfo` objects is returned.

inspect.**currentframe**()

호출자의 스택 프레임에 대한 프레임 객체를 반환합니다.

이 함수는 인터프리터의 파이썬 스택 프레임 지원에 의존하며, 모든 파이썬 구현에서 제공된다고 보장되는 것은 아닙니다. 파이썬 스택 프레임 지원이 없는 구현에서 실행하면, 이 함수는 `None`을 반환합니다.

inspect.**stack**(*context=1*)

호출자의 스택에 대한 `FrameInfo` 객체 리스트를 반환합니다. 반환된 리스트의 첫 번째 항목은 호출자를 나타냅니다; 마지막 항목은 스택에서 가장 바깥쪽 호출을 나타냅니다.

버전 3.5에서 변경: 네임드 튜플 `FrameInfo(frame, filename, lineno, function, code_context, index)`의 리스트가 반환됩니다.

버전 3.11에서 변경: A list of `FrameInfo` objects is returned.

inspect.**trace**(*context=1*)

현재 프레임과 현재 처리 중인 예외가 발생한 프레임 사이의 스택에 대한 `FrameInfo` 객체 리스트를 반환합니다. 리스트의 첫 번째 항목은 호출자를 나타냅니다; 마지막 항목은 예외가 발생한 위치를 나타냅니다.

버전 3.5에서 변경: 네임드 튜플 `FrameInfo(frame, filename, lineno, function, code_context, index)`의 리스트가 반환됩니다.

버전 3.11에서 변경: A list of `FrameInfo` objects is returned.

29.14.6 정적으로 어트리뷰트 가져오기

`getattr()`과 `hasattr()`은 모두 어트리뷰트를 가져오거나 존재하는지 확인할 때 코드 실행을 유발할 수 있습니다. 프로퍼티와 같은 디스크립터가 호출되고 `__getattr__()`과 `__getattribute__()`가 호출될 수 있습니다.

문서화 도구처럼 수동적인(passive) 검사를 원할 때는 불편할 수 있습니다. `getattr_static()`은 `getattr()`과 같은 서명을 갖지만 어트리뷰트를 가져올 때 코드 실행을 피합니다.

inspect.**getattr_static**(*obj*, *attr*, *default=None*)

디스크립터 프로토콜, `__getattr__()` 또는 `__getattribute__()`를 통한 동적 조회를 일으키지 않고 어트리뷰트를 조회합니다.

참고: 이 함수는 getattr이 가져올 수 있는 모든 어트리뷰트를 조회하지 못할 수 있으며 (가령 동적으로 만들어진 어트리뷰트), getattr이 가져올 수 없는 어트리뷰트를 찾을 수 있습니다 (가령 AttributeError를 발생시키는 디스크립터). 또한 인스턴스 멤버 대신 디스크립터 객체를 반환할 수도 있습니다.

인스턴스 `__dict__`가 다른 멤버(예를 들어 프로퍼티)에 의해 가려지면 이 함수는 인스턴스 멤버를 찾을 수 없습니다.

Added in version 3.2.

*getattr_static()*은 디스크립터를 해석하지 않습니다, 예를 들어 C로 구현된 객체의 슬롯 디스크립터나 getset 디스크립터. 하부 어트리뷰트 대신 디스크립터 객체가 반환됩니다.

다음과 같은 코드로 이를 처리할 수 있습니다. 임의의 getset 디스크립터에 대해 이를 호출하면 코드 실행이 유발될 수 있음에 유의하십시오:

```
# 내장 디스크립터 형을 해석하기 위한 예제 코드
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descriptor)

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # 디스크립터는 AttributeError를 발생 시켜
        # 하부 값이 없음을 나타냅니다.
        # 이 경우 디스크립터 자체가
        # 수행해야 합니다.
        pass
```

29.14.7 제너레이터, 코루틴 및 비동기 제너레이터의 현재 상태

코루틴 스케줄러를 구현할 때와 기타 제너레이터의 고급 사용을 위해, 제너레이터가 현재 실행 중인지, 시작, 재개 또는 실행을 대기하는 중인지, 또는 이미 종료되었는지를 판별하는 것이 유용합니다. *getgeneratorstate()*를 사용하면 제너레이터의 현재 상태를 쉽게 확인할 수 있습니다.

inspect.**getgeneratorstate**(*generator*)

제너레이터-이터레이터의 현재 상태를 가져옵니다.

가능한 상태는 다음과 같습니다:

- GEN_CREATED: 실행 시작을 기다리는 중입니다.
- GEN_RUNNING: 현재 인터프리터에서 실행 중입니다.
- GEN_SUSPENDED: 현재 일드(yield) 표현식에서 일시 중지되었습니다.
- GEN_CLOSED: 실행이 완료되었습니다.

Added in version 3.2.

inspect.**getcoroutinestate**(*coroutine*)

코루틴 객체의 현재 상태를 가져옵니다. 이 함수는 `async def` 함수가 만든 코루틴 객체와 함께 사용하기 위한 것이지만, `cr_running`과 `cr_frame` 어트리뷰트가 있는 임의의 코루틴류 객체를 허용합니다.

가능한 상태는 다음과 같습니다:

- CORO_CREATED: 실행 시작을 기다리는 중입니다.
- CORO_RUNNING: 현재 인터프리터에서 실행 중입니다.
- CORO_SUSPENDED: 현재 어웨이트(await) 표현식에서 일시 중지되었습니다.
- CORO_CLOSED: 실행이 완료되었습니다.

Added in version 3.5.

inspect.**getasyncgenstate**(*agen*)

비동기 제너레이터 객체의 현재 상태를 가져옵니다. 이 함수는 yield 문을 사용하는 async def 함수가 만든 비동기 이터레이터 객체와 함께 사용하기 위한 것이지만, ag_running과 ag_frame 어트리뷰트가 있는 임의의 비동기 제너레이터류 객체를 허용합니다.

가능한 상태는 다음과 같습니다:

- AGEN_CREATED: 실행 시작을 기다리는 중입니다.
- AGEN_RUNNING: 현재 인터프리터에서 실행 중입니다.
- AGEN_SUSPENDED: 현재 일드(yield) 표현식에서 일시 중지되었습니다.
- AGEN_CLOSED: 실행이 완료되었습니다.

Added in version 3.12.

제너레이터의 현재 내부 상태도 조회할 수 있습니다. 이는 주로 내부 상태가 예상대로 갱신되었는지 확인하는 테스트 목적으로 유용합니다:

inspect.**getgeneratorlocals**(*generator*)

*generator*의 라이브 로컬 변수에서 그것의 현재 값으로의 매핑을 얻습니다. 변수 이름을 값으로 매핑하는 딕셔너리가 반환됩니다. 이것은 제너레이터 바디에서 *locals()*를 호출하는 것과 동등하며, 같은 경고가 모두 적용됩니다.

*generator*가 현재 연결된 프레임이 없는 제너레이터이면, 빈 딕셔너리가 반환됩니다. *generator*가 파이썬 제너레이터 객체가 아니면 *TypeError*가 발생합니다.

이 함수는 내부 검사를 위해 파이썬 스택 프레임을 노출하는 제너레이터에 의존하며, 모든 파이썬 구현에서 보장되는 것은 아닙니다. 그럴 경우, 이 함수는 항상 빈 딕셔너리를 반환합니다.

Added in version 3.3.

inspect.**getcoroutinelocals**(*coroutine*)

이 함수는 *getgeneratorlocals()*와 유사하지만, async def 함수가 만든 코루틴 객체에 작동합니다.

Added in version 3.5.

inspect.**getasyncgenlocals**(*agen*)

이 함수는 *getgeneratorlocals()*와 유사하지만, yield 문을 사용하는 async def 함수가 만든 비동기 제너레이터 객체에 작동합니다.

Added in version 3.12.

29.14.8 코드 객체 비트 플래그

파이썬 코드 객체에는 co_flags 어트리뷰트가 있으며, 이는 다음 플래그의 비트맵입니다:

inspect.**CO_OPTIMIZED**

코드 객체는 빠른 locals(fast locals)를 사용하여 최적화되었습니다.

inspect.**CO_NEWLOCALS**

설정되면, 코드 객체가 실행될 때 프레임의 f_locals에 대한 새 딕셔너리가 만들어집니다.

inspect.**CO_VARARGS**

코드 객체에는 (*args 같은) 가변 위치 매개 변수가 있습니다.

inspect.**CO_VARKEYWORDS**

코드 객체에는 (**kwargs 같은) 가변 키워드 매개 변수가 있습니다.

inspect.**CO_NESTED**

코드 객체가 중첩 함수일 때 이 플래그가 설정됩니다.

inspect.**CO_GENERATOR**

코드 객체가 제너레이터 함수일 때, 즉 코드 객체가 실행될 때 제너레이터 객체를 반환할 때 이 플래그가 설정됩니다.

inspect.**CO_COROUTINE**

코드 객체가 코루틴 함수일 때 이 플래그가 설정됩니다. 코드 객체가 실행될 때 코루틴 객체를 반환합니다. 자세한 내용은 **PEP 492**를 참조하십시오.

Added in version 3.5.

inspect.**CO_ITERABLE_COROUTINE**

이 플래그는 제너레이터를 제너레이터 기반 코루틴으로 변환하는 데 사용됩니다. 이 플래그가 있는 제너레이터 객체는 `await` 표현식에 사용될 수 있으며, 코루틴 객체를 `yield from` 할 수 있습니다. 자세한 내용은 **PEP 492**를 참조하십시오.

Added in version 3.5.

inspect.**CO_ASYNC_GENERATOR**

코드 객체가 비동기 제너레이터 함수일 때 이 플래그가 설정됩니다. 코드 객체가 실행될 때 비동기 제너레이터 객체가 반환됩니다. 자세한 내용은 **PEP 525**를 참조하십시오.

Added in version 3.6.

inspect.**CO_HAS_DOCSTRING**

The flag is set when there is a docstring for the code object in the source code. If set, it will be the first item in `co_consts`.

Added in version 3.14.

inspect.**CO_METHOD**

The flag is set when the code object is a function defined in class scope.

Added in version 3.14.

> **참고**
>
> 이 플래그들은 CPython에만 해당하며, 다른 파이썬 구현에서는 정의되지 않을 수 있습니다. 또한 플래그는 구현 세부 사항이며, 향후 파이썬 배포에서 제거되거나 폐지될 수 있습니다. 모든 내부 검사에는 *inspect* 모듈의 공개 API를 사용하는 것이 좋습니다.

29.14.9 Buffer flags

class inspect.**BufferFlags**

This is an *enum.IntFlag* that represents the flags that can be passed to the `__buffer__()` method of objects implementing the buffer protocol.

The meaning of the flags is explained at buffer-request-types.

SIMPLE

WRITABLE

FORMAT

ND

STRIDES

C_CONTIGUOUS

F_CONTIGUOUS

ANY_CONTIGUOUS

INDIRECT

CONTIG

CONTIG_RO

STRIDED

STRIDED_RO

RECORDS

RECORDS_RO

FULL

FULL_RO

READ

WRITE

Added in version 3.12.

29.14.10 명령 줄 인터페이스

`inspect` 모듈은 명령 줄에서 기본 내부 검사 기능을 제공하기도 합니다.

기본적으로, 모듈의 이름을 받아들이고 해당 모듈의 소스를 인쇄합니다. 콜론과 대상 객체의 정규화된 이름을 덧붙여, 대신 모듈 내의 클래스나 함수를 인쇄 할 수 있습니다.

--details

소스 코드 대신에 지정된 객체에 대한 정보를 인쇄합니다

29.15 `annotationlib` — Functionality for introspecting annotations

Added in version 3.14.

Source code: Lib/annotationlib.py

The `annotationlib` module provides tools for introspecting *annotations* on modules, classes, and functions.

Annotations are lazily evaluated and often contain forward references to objects that are not yet defined when the annotation is created. This module provides a set of low-level tools that can be used to retrieve annotations in a reliable way, even in the presence of forward references and other edge cases.

This module supports retrieving annotations in three main formats (see `Format`), each of which works best for different use cases:

- `VALUE` evaluates the annotations and returns their value. This is most straightforward to work with, but it may raise errors, for example if the annotations contain references to undefined names.
- `FORWARDREF` returns `ForwardRef` objects for annotations that cannot be resolved, allowing you to inspect the annotations without evaluating them. This is useful when you need to work with annotations that may contain unresolved forward references.
- `STRING` returns the annotations as a string, similar to how it would appear in the source file. This is useful for documentation generators that want to display annotations in a readable way.

The `get_annotations()` function is the main entry point for retrieving annotations. Given a function, class, or module, it returns an annotations dictionary in the requested format. This module also provides functionality for working directly with the *annotate function* that is used to evaluate annotations, such as `get_annotate_from_class_namespace()` and `call_annotate_function()`, as well as the `call_evaluate_function()` function for working with *evaluate functions*.

조심

Most functionality in this module can execute arbitrary code; see *the security section* for more information.

더 보기

PEP 649 proposed the current model for how annotations work in Python.

PEP 749 expanded on various aspects of **PEP 649** and introduced the `annotationlib` module.

annotations-howto provides best practices for working with annotations.

typing-extensions provides a backport of `get_annotations()` that works on earlier versions of Python.

29.15.1 Annotation semantics

The way annotations are evaluated has changed over the history of Python 3, and currently still depends on a future import. There have been execution models for annotations:

- *Stock semantics* (default in Python 3.0 through 3.13; see **PEP 3107** and **PEP 526**): Annotations are evaluated eagerly, as they are encountered in the source code.
- *Stringified annotations* (used with `from __future__ import annotations` in Python 3.7 and newer; see **PEP 563**): Annotations are stored as strings only.
- *Deferred evaluation* (default in Python 3.14 and newer; see **PEP 649** and **PEP 749**): Annotations are evaluated lazily, only when they are accessed.

As an example, consider the following program:

```
def func(a: Cls) -> None:
    print(a)

class Cls: pass

print(func.__annotations__)
```

This will behave as follows:

- Under stock semantics (Python 3.13 and earlier), it will throw a `NameError` at the line where `func` is defined, because `Cls` is an undefined name at that point.
- Under stringified annotations (if `from __future__ import annotations` is used), it will print `{'a': 'Cls', 'return': 'None'}`.
- Under deferred evaluation (Python 3.14 and later), it will print `{'a': <class 'Cls'>, 'return': None}`.

Stock semantics were used when function annotations were first introduced in Python 3.0 (by **PEP 3107**) because this was the simplest, most obvious way to implement annotations. The same execution model was used when variable annotations were introduced in Python 3.6 (by **PEP 526**). However, stock semantics caused problems when using annotations as type hints, such as a need to refer to names that are not yet defined when the annotation is encountered. In addition, there were performance problems with executing annotations at module import time. Therefore, in Python 3.7, **PEP 563** introduced the ability to store annotations as strings using the `from __future__ import annotations` syntax. The plan at the time was to eventually make this behavior the default, but a problem appeared:

stringified annotations are more difficult to process for those who introspect annotations at runtime. An alternative proposal, **PEP 649**, introduced the third execution model, deferred evaluation, and was implemented in Python 3.14. Stringified annotations are still used if `from __future__ import annotations` is present, but this behavior will eventually be removed.

29.15.2 Classes

class annotationlib.**Format**

An *IntEnum* describing the formats in which annotations can be returned. Members of the enum, or their equivalent integer values, can be passed to *get_annotations()* and other functions in this module, as well as to `__annotate__` functions.

VALUE = 1

Values are the result of evaluating the annotation expressions.

VALUE_WITH_FAKE_GLOBALS = 2

Special value used to signal that an annotate function is being evaluated in a special environment with fake globals. When passed this value, annotate functions should either return the same value as for the *Format.VALUE* format, or raise *NotImplementedError* to signal that they do not support execution in this environment. This format is only used internally and should not be passed to the functions in this module.

FORWARDREF = 3

Values are real annotation values (as per *Format.VALUE* format) for defined values, and *ForwardRef* proxies for undefined values. Real objects may contain references to *ForwardRef* proxy objects.

STRING = 4

Values are the text string of the annotation as it appears in the source code, up to modifications including, but not restricted to, whitespace normalizations and constant values optimizations.

The exact values of these strings may change in future versions of Python.

Added in version 3.14.

class annotationlib.**ForwardRef**

A proxy object for forward references in annotations.

Instances of this class are returned when the *FORWARDREF* format is used and annotations contain a name that cannot be resolved. This can happen when a forward reference is used in an annotation, such as when a class is referenced before it is defined.

__forward_arg__

A string containing the code that was evaluated to produce the *ForwardRef*. The string may not be exactly equivalent to the original source.

evaluate(*, *owner=None*, *globals=None*, *locals=None*, *type_params=None*, *format=Format.VALUE*)

Evaluate the forward reference, returning its value.

If the *format* argument is *VALUE* (the default), this method may throw an exception, such as *NameError*, if the forward reference refers to a name that cannot be resolved. The arguments to this method can be used to provide bindings for names that would otherwise be undefined. If the *format* argument is *FORWARDREF*, the method will never throw an exception, but may return a *ForwardRef* instance. For example, if the forward reference object contains the code `list[undefined]`, where `undefined` is a name that is not defined, evaluating it with the *FORWARDREF* format will return `list[ForwardRef('undefined')]`. If the *format* argument is *STRING*, the method will return *__forward_arg__*.

The *owner* parameter provides the preferred mechanism for passing scope information to this method. The owner of a *ForwardRef* is the object that contains the annotation from which the *ForwardRef* derives, such as a module object, type object, or function object.

The *globals*, *locals*, and *type_params* parameters provide a more precise mechanism for influencing the names that are available when the *ForwardRef* is evaluated. *globals* and *locals* are passed to *eval()*,

representing the global and local namespaces in which the name is evaluated. The *type_params* parameter is relevant for objects created using the native syntax for generic classes and functions. It is a tuple of type parameters that are in scope while the forward reference is being evaluated. For example, if evaluating a `ForwardRef` retrieved from an annotation found in the class namespace of a generic class `C`, *type_params* should be set to `C.__type_params__`.

`ForwardRef` instances returned by `get_annotations()` retain references to information about the scope they originated from, so calling this method with no further arguments may be sufficient to evaluate such objects. `ForwardRef` instances created by other means may not have any information about their scope, so passing arguments to this method may be necessary to evaluate them successfully.

If no *owner*, *globals*, *locals*, or *type_params* are provided and the `ForwardRef` does not contain information about its origin, empty globals and locals dictionaries are used.

Added in version 3.14.

29.15.3 Functions

`annotationlib.`**`annotations_to_string`**(*annotations*)

Convert an annotations dict containing runtime values to a dict containing only strings. If the values are not already strings, they are converted using `type_repr()`. This is meant as a helper for user-provided annotate functions that support the `STRING` format but do not have access to the code creating the annotations.

For example, this is used to implement the `STRING` for `typing.TypedDict` classes created through the functional syntax:

```
>>> from typing import TypedDict
>>> Movie = TypedDict("movie", {"name": str, "year": int})
>>> get_annotations(Movie, format=Format.STRING)
{'name': 'str', 'year': 'int'}
```

Added in version 3.14.

`annotationlib.`**`call_annotate_function`**(*annotate*, *format*, **, *owner=None*)

Call the *annotate function* *annotate* with the given *format*, a member of the `Format` enum, and return the annotations dictionary produced by the function.

This helper function is required because annotate functions generated by the compiler for functions, classes, and modules only support the `VALUE` format when called directly. To support other formats, this function calls the annotate function in a special environment that allows it to produce annotations in the other formats. This is a useful building block when implementing functionality that needs to partially evaluate annotations while a class is being constructed.

owner is the object that owns the annotation function, usually a function, class, or module. If provided, it is used in the `FORWARDREF` format to produce a `ForwardRef` object that carries more information.

> **더 보기**
>
> **PEP 649** contains an explanation of the implementation technique used by this function.

Added in version 3.14.

`annotationlib.`**`call_evaluate_function`**(*evaluate*, *format*, **, *owner=None*)

Call the *evaluate function* *evaluate* with the given *format*, a member of the `Format` enum, and return the value produced by the function. This is similar to `call_annotate_function()`, but the latter always returns a dictionary mapping strings to annotations, while this function returns a single value.

This is intended for use with the evaluate functions generated for lazily evaluated elements related to type aliases and type parameters:

- `typing.TypeAliasType.evaluate_value()`, the value of type aliases

- `typing.TypeVar.evaluate_bound()`, the bound of type variables
- `typing.TypeVar.evaluate_constraints()`, the constraints of type variables
- `typing.TypeVar.evaluate_default()`, the default value of type variables
- `typing.ParamSpec.evaluate_default()`, the default value of parameter specifications
- `typing.TypeVarTuple.evaluate_default()`, the default value of type variable tuples

owner is the object that owns the evaluate function, such as the type alias or type variable object.

format can be used to control the format in which the value is returned:

```
>>> type Alias = undefined
>>> call_evaluate_function(Alias.evaluate_value, Format.VALUE)
Traceback (most recent call last):
...
NameError: name 'undefined' is not defined
>>> call_evaluate_function(Alias.evaluate_value, Format.FORWARDREF)
ForwardRef('undefined')
>>> call_evaluate_function(Alias.evaluate_value, Format.STRING)
'undefined'
```

Added in version 3.14.

annotationlib.**get_annotate_from_class_namespace**(*namespace*)

Retrieve the *annotate function* from a class namespace dictionary *namespace*. Return `None` if the namespace does not contain an annotate function. This is primarily useful before the class has been fully created (e.g., in a metaclass); after the class exists, the annotate function can be retrieved with `cls.__annotate__`. See *below* for an example using this function in a metaclass.

Added in version 3.14.

annotationlib.**get_annotations**(*obj*, **, *globals=None*, *locals=None*, *eval_str=False*, *format=Format.VALUE*)

Compute the annotations dict for an object.

obj may be a callable, class, module, or other object with `__annotate__` or `__annotations__` attributes. Passing any other object raises `TypeError`.

The *format* parameter controls the format in which annotations are returned, and must be a member of the `Format` enum or its integer equivalent. The different formats work as follows:

- VALUE: `object.__annotations__` is tried first; if that does not exist, the `object.__annotate__` function is called if it exists.
- FORWARDREF: If `object.__annotations__` exists and can be evaluated successfully, it is used; otherwise, the `object.__annotate__` function is called. If it does not exist either, `object.__annotations__` is tried again and any error from accessing it is re-raised.
- STRING: If `object.__annotate__` exists, it is called first; otherwise, `object.__annotations__` is used and stringified using `annotations_to_string()`.

Returns a dict. `get_annotations()` returns a new dict every time it' s called; calling it twice on the same object will return two different but equivalent dicts.

This function handles several details for you:

- If *eval_str* is true, values of type `str` will be un-stringized using `eval()`. This is intended for use with stringized annotations (`from __future__ import annotations`). It is an error to set *eval_str* to true with formats other than `Format.VALUE`.
- If *obj* doesn' t have an annotations dict, returns an empty dict. (Functions and methods always have an annotations dict; classes, modules, and other types of callables may not.)

- Ignores inherited annotations on classes, as well as annotations on metaclasses. If a class doesn' t have its own annotations dict, returns an empty dict.
- All accesses to object members and dict values are done using `getattr()` and `dict.get()` for safety.

eval_str controls whether or not values of type `str` are replaced with the result of calling `eval()` on those values:

- If eval_str is true, `eval()` is called on values of type `str`. (Note that `get_annotations()` doesn' t catch exceptions; if `eval()` raises an exception, it will unwind the stack past the `get_annotations()` call.)
- If *eval_str* is false (the default), values of type `str` are unchanged.

globals and *locals* are passed in to `eval()`; see the documentation for `eval()` for more information. If *globals* or *locals* is `None`, this function may replace that value with a context-specific default, contingent on `type(obj)`:

- If *obj* is a module, *globals* defaults to `obj.__dict__`.
- If *obj* is a class, *globals* defaults to `sys.modules[obj.__module__].__dict__` and *locals* defaults to the *obj* class namespace.
- If *obj* is a callable, *globals* defaults to `obj.__globals__`, although if *obj* is a wrapped function (using `functools.update_wrapper()`) or a `functools.partial` object, it is unwrapped until a non-wrapped function is found.

Calling `get_annotations()` is best practice for accessing the annotations dict of any object. See annotations-howto for more information on annotations best practices.

```
>>> def f(a: int, b: str) -> float:
...     pass
>>> get_annotations(f)
{'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'float'>}
```

Added in version 3.14.

`annotationlib.`**`type_repr`**(*value*)

Convert an arbitrary Python value to a format suitable for use by the `STRING` format. This calls `repr()` for most objects, but has special handling for some objects, such as type objects.

This is meant as a helper for user-provided annotate functions that support the `STRING` format but do not have access to the code creating the annotations. It can also be used to provide a user-friendly string representation for other objects that contain values that are commonly encountered in annotations.

Added in version 3.14.

29.15.4 Recipes

Using annotations in a metaclass

A metaclass may want to inspect or even modify the annotations in a class body during class creation. Doing so requires retrieving annotations from the class namespace dictionary. For classes created with `from __future__ import annotations`, the annotations will be in the `__annotations__` key of the dictionary. For other classes with annotations, `get_annotate_from_class_namespace()` can be used to get the annotate function, and `call_annotate_function()` can be used to call it and retrieve the annotations. Using the `FORWARDREF` format will usually be best, because this allows the annotations to refer to names that cannot yet be resolved when the class is created.

To modify the annotations, it is best to create a wrapper annotate function that calls the original annotate function, makes any necessary adjustments, and returns the result.

Below is an example of a metaclass that filters out all `typing.ClassVar` annotations from the class and puts them in a separate attribute:

```
import annotationlib
import typing

class ClassVarSeparator(type):
   def __new__(mcls, name, bases, ns):
      if "__annotations__" in ns:  # from __future__ import annotations
         annotations = ns["__annotations__"]
         classvar_keys = {
            key for key, value in annotations.items()
            # Use string comparison for simplicity; a more robust solution
            # could use annotationlib.ForwardRef.evaluate
            if value.startswith("ClassVar")
         }
         classvars = {key: annotations[key] for key in classvar_keys}
         ns["__annotations__"] = {
            key: value for key, value in annotations.items()
            if key not in classvar_keys
         }
         wrapped_annotate = None
      elif annotate := annotationlib.get_annotate_from_class_namespace(ns):
         annotations = annotationlib.call_annotate_function(
            annotate, format=annotationlib.Format.FORWARDREF
         )
         classvar_keys = {
            key for key, value in annotations.items()
            if typing.get_origin(value) is typing.ClassVar
         }
         classvars = {key: annotations[key] for key in classvar_keys}

         def wrapped_annotate(format):
            annos = annotationlib.call_annotate_function(annotate, format,␣
↪owner=typ)
            return {key: value for key, value in annos.items() if key not in␣
↪classvar_keys}

      else:  # no annotations
         classvars = {}
         wrapped_annotate = None
      typ = super().__new__(mcls, name, bases, ns)

      if wrapped_annotate is not None:
         # Wrap the original __annotate__ with a wrapper that removes ClassVars
         typ.__annotate__ = wrapped_annotate
      typ.classvars = classvars  # Store the ClassVars in a separate attribute
      return typ
```

29.15.5 Limitations of the STRING format

The *STRING* format is meant to approximate the source code of the annotation, but the implementation strategy used means that it is not always possible to recover the exact source code.

First, the stringifier of course cannot recover any information that is not present in the compiled code, including comments, whitespace, parenthesization, and operations that get simplified by the compiler.

Second, the stringifier can intercept almost all operations that involve names looked up in some scope, but it cannot intercept operations that operate fully on constants. As a corollary, this also means it is not safe to request the STRING format on untrusted code: Python is powerful enough that it is possible to achieve arbitrary code execution even with no access to any globals or builtins. For example:

```
>>> def f(x: (1).__class__.__base__.__subclasses__()[-1].__init__.__builtins__[
↪"print"]("Hello world")): pass
...
>>> annotationlib.get_annotations(f, format=annotationlib.Format.STRING)
Hello world
{'x': 'None'}
```

참고

This particular example works as of the time of writing, but it relies on implementation details and is not guaranteed to work in the future.

Among the different kinds of expressions that exist in Python, as represented by the `ast` module, some expressions are supported, meaning that the `STRING` format can generally recover the original source code; others are unsupported, meaning that they may result in incorrect output or an error.

The following are supported (sometimes with caveats):

- `ast.BinOp`
- `ast.UnaryOp`
  - `ast.Invert` (~), `ast.UAdd` (+), and `ast.USub` (-) are supported
  - `ast.Not` (`not`) is not supported
- `ast.Dict` (except when using `**` unpacking)
- `ast.Set`
- `ast.Compare`
  - `ast.Eq` and `ast.NotEq` are supported
  - `ast.Lt`, `ast.LtE`, `ast.Gt`, and `ast.GtE` are supported, but the operand may be flipped
  - `ast.Is`, `ast.IsNot`, `ast.In`, and `ast.NotIn` are not supported
- `ast.Call` (except when using `**` unpacking)
- `ast.Constant` (though not the exact representation of the constant; for example, escape sequences in strings are lost; hexadecimal numbers are converted to decimal)
- `ast.Attribute` (assuming the value is not a constant)
- `ast.Subscript` (assuming the value is not a constant)
- `ast.Starred` (`*` unpacking)
- `ast.Name`
- `ast.List`
- `ast.Tuple`
- `ast.Slice`

The following are unsupported, but throw an informative error when encountered by the stringifier:

- `ast.FormattedValue` (f-strings; error is not detected if conversion specifiers like `!r` are used)
- `ast.JoinedStr` (f-strings)

The following are unsupported and result in incorrect output:

- `ast.BoolOp` (`and` and `or`)
- `ast.IfExp`
- `ast.Lambda`

- *ast.ListComp*
- *ast.SetComp*
- *ast.DictComp*
- *ast.GeneratorExp*

The following are disallowed in annotation scopes and therefore not relevant:

- *ast.NamedExpr* (:=)
- *ast.Await*
- *ast.Yield*
- *ast.YieldFrom*

29.15.6 Limitations of the FORWARDREF format

The *FORWARDREF* format aims to produce real values as much as possible, with anything that cannot be resolved replaced with *ForwardRef* objects. It is affected by broadly the same Limitations as the *STRING* format: annotations that perform operations on literals or that use unsupported expression types may raise exceptions when evaluated using the *FORWARDREF* format.

Below are a few examples of the behavior with unsupported expressions:

```
>>> from annotationlib import get_annotations, Format
>>> def zerodiv(x: 1 / 0): ...
>>> get_annotations(zerodiv, format=Format.STRING)
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
>>> get_annotations(zerodiv, format=Format.FORWARDREF)
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
>>> def ifexp(x: 1 if y else 0): ...
>>> get_annotations(ifexp, format=Format.STRING)
{'x': '1'}
```

29.15.7 Security implications of introspecting annotations

Much of the functionality in this module involves executing code related to annotations, which can then do arbitrary things. For example, *get_annotations()* may call an arbitrary *annotate function*, and *ForwardRef.evaluate()* may call *eval()* on an arbitrary string. Code contained in an annotation might make arbitrary system calls, enter an infinite loop, or perform any other operation. This is also true for any access of the __annotations__ attribute, and for various functions in the *typing* module that work with annotations, such as *typing.get_type_hints()*.

Any security issue arising from this also applies immediately after importing code that may contain untrusted annotations: importing code can always cause arbitrary operations to be performed. However, it is unsafe to accept strings or other input from an untrusted source and pass them to any of the APIs for introspecting annotations, for example by editing an __annotations__ dictionary or directly creating a *ForwardRef* object.

29.16 site — 사이트별 구성 훅

소스 코드: Lib/site.py

이 모듈은 초기화 중에 자동으로 임포트 됩니다. 인터프리터의 -S 옵션을 사용하여 자동 임포트를 억제할 수 있습니다.

이 모듈을 임포트하면 일반적으로 사이트별 경로를 모듈 검색 경로에 추가하고 *help()*를 포함한 콜러블들을 내장 이름 공간에 추가합니다. 그러나 시작 옵션 -S는 이를 막고 모듈 검색 경로의 자동 수정이나 내장의 추가 없이 이 모듈을 안전하게 임포트할 수 있습니다. 일반적인 사이트별 추가를 명시적으로 트리거하려면, *main()* 함수를 호출하십시오.

버전 3.3에서 변경: -S를 사용하는 경우에도 모듈을 임포트 하면 경로 조작을 트리거 했습니다.

머리와 꼬리 부분에서 최대 4개의 디렉터리를 구성하는 것으로 시작합니다. 머리 부분에는, sys.prefix와 sys.exec_prefix를 사용합니다; 빈 머리는 건너뜁니다. 꼬리 부분에는, 빈 문자열을 사용하고 lib/site-packages(윈도우에서)나 lib/python*X.Y[t]*/site-packages(유닉스와 맥 OS에서)를 사용합니다. (선택적 접미사 "t" 는 자유 스레딩 빌드를 나타내며, "t"가 *sys.abiflags* 상수에 있으면 추가됩니다.) 각 머리-꼬리 조합마다, 존재하는 디렉터리를 참조하는지 확인하고, 그렇다면 디렉터리를 sys.path에 추가하고 새로 추가된 경로에서 구성 파일을 검사합니다.

버전 3.5에서 변경: "site-python" 디렉터리에 대한 지원이 제거되었습니다.

버전 3.13에서 변경: On Unix, *Free threading* Python installations are identified by the "t" suffix in the version-specific directory name, such as lib/python3.13t/.

버전 3.14에서 변경: *site* is no longer responsible for updating *sys.prefix* and *sys.exec_prefix* on *Virtual Environments*. This is now done during the *path initialization*. As a result, under *Virtual Environments*, *sys.prefix* and *sys.exec_prefix* no longer depend on the *site* initialization, and are therefore unaffected by -S. When running under a *virtual environment*, the pyvenv.cfg file in *sys.prefix* is checked for site-specific configurations. If the include-system-site-packages key exists and is set to true (case-insensitive), the system-level prefixes will be searched for site-packages, otherwise they won' t.

경로 구성 파일은 이름이 *name*.pth 인 파일이며, 위에서 언급한 4개의 디렉터리 중 하나에 존재합니다; 내용은 sys.path에 추가될 추가 항목(한 줄에 하나씩)입니다. 존재하지 않는 항목은 sys.path에 추가되지 않으며, 항목이 파일이 아닌 디렉터리를 참조하는지 확인하지 않습니다. 어떤 항목도 sys.path에 두 번 추가되지 않습니다. 빈 줄과 #으로 시작하는 줄은 건너뜁니다. import로 시작하는 (공백이나 탭이 뒤따르는) 줄은 실행됩니다.

> **참고**
>
> .pth 파일의 실행 줄은 특정 모듈이 실제 사용될지에 관계없이 모든 파이썬 시작 시에 실행됩니다. 따라서 영향을 최소화해야 합니다. 실행 줄의 주요 목적은 해당 모듈을 임포트 가능하게 만드는 것입니다 (제삼자 임포트 훅 로드, PATH 조정 등). 다른 초기화는 모듈의 실제 임포트에서 수행된다고 간주합니다, (임포트 한다면 그리고 임포트 할 때). 코드 청크를 한 줄로 제한하는 것은 여기에 더 복잡한 것을 넣지 않도록 하려는 의도입니다.

버전 3.13에서 변경: The .pth files are now decoded by UTF-8 at first and then by the *locale encoding* if it fails.

예를 들어, sys.prefix와 sys.exec_prefix가 /usr/local로 설정되었다고 가정하십시오. 그러면 파이썬 X.Y 라이브러리는 /usr/local/lib/python*X.Y*에 설치되어 있습니다. 여기에 foo, bar 및 spam 이라는 세 개의 서브 디렉터리와, foo.pth와 bar.pth라는 두 개의 경로 구성 파일이 있는 서브 디렉터리 /usr/local/lib/python*X.Y*/site-packages가 있다고 가정하십시오. foo.pth에 다음이 포함되어 있고:

```
# foo package configuration

foo
bar
bletch
```

bar.pth는 다음을 포함한다고 가정하십시오:

```
# bar package configuration

bar
```

그러면 다음과 같은 버전 별 디렉터리가 이 순서대로 sys.path에 추가됩니다:

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

bletch가 존재하지 않기 때문에 생략되었음에 유의하십시오; bar.pth가 알파벳순으로 foo.pth 앞에 오기 때문에 bar 디렉터리가 foo 디렉터리보다 앞에 옵니다; spam은 경로 구성 파일에 언급되어 있지 않기 때문에 생략되었습니다.

29.16.1 sitecustomize

이러한 경로 조작 후, 임의의 사이트별 사용자 정의를 수행할 수 있는 *sitecustomize*라는 모듈을 임포트 하려고 시도합니다. 일반적으로 시스템 관리자가 site-packages 디렉터리에 만듭니다. 이 임포트가 *ImportError*나 이것의 서브 클래스 예외로 실패하고, 예외의 *name* 어트리뷰트가 'sitecustomize'와 같으면, 조용히 무시됩니다. 윈도우에서 pythonw.exe(IDLE을 시작하는 데 기본적으로 사용됩니다)처럼, 사용 가능한 출력 스트림 없이 파이썬을 시작하면, *sitecustomize*의 출력 시도는 무시됩니다. 다른 예외는 절차의 조용한 그리고 아마도 정체불명의 실패로 이어집니다.

29.16.2 usercustomize

그런 다음, *ENABLE_USER_SITE*가 참이면, 임의의 사용자별 사용자 정의를 수행할 수 있는 *usercustomize*라는 모듈을 임포트 하려고 시도합니다. 이 파일은 사용자 site-packages 디렉터리(아래를 보십시오)에 만들어지는 것이 관례입니다, -s에 의해 비활성화되지 않는 한 sys.path의 일부입니다. 이 임포트가 *ImportError*나 이것의 서브 클래스 예외로 실패하고, 예외의 *name* 어트리뷰트가 'usercustomize'와 같으면, 조용히 무시됩니다.

유닉스가 아닌 일부 시스템에서는, sys.prefix와 sys.exec_prefix가 비어 있고, 경로 조작을 건너뜀에 유의하십시오; 하지만 *sitecustomize*와 *usercustomize* 임포트는 여전히 시도됩니다.

29.16.3 Readline 구성

*readline*을 지원하는 시스템에서, 파이썬이 대화형 모드로 -S 옵션 없이 시작되면, 이 모듈은 *rlcompleter* 모듈을 임포트하고 구성합니다. 기본 동작은 탭 완성을 활성화하고 ~/.python_history를 히스토리 저장 파일로 사용하는 것입니다. 이를 비활성화하려면, *sitecustomize*나 *usercustomize* 모듈 또는 PYTHONSTARTUP 파일에서 *sys.__interactivehook__* 어트리뷰트를 삭제(또는 재정의)하십시오.

버전 3.4에서 변경: rlcompleter와 히스토리 활성화가 자동으로 이루어졌습니다.

29.16.4 모듈 내용

site.**PREFIXES**

site-packages 디렉터리의 접두사 리스트.

site.**ENABLE_USER_SITE**

사용자 site-packages 디렉터리의 상태를 나타내는 플래그. True는 활성화되어 sys.path에 추가되었음을 의미합니다. False는 사용자 요청(-s나 PYTHONNOUSERSITE로)에 의해 비활성화되었음을 의미합니다. None은 보안상의 이유(사용자나 그룹 id와 유효(effective) id가 일치하지 않음)로 또는 관리자에 의해 비활성화되었음을 의미합니다.

site.**USER_SITE**

실행 중인 파이썬의 사용자 site-packages 경로. *getusersitepackages()*가 아직 호출되지 않았으면 None일 수 있습니다. 기본값은 유닉스와 비 프레임워크 맥 OS 빌드의 경우 ~/.local/lib/python*X.Y*[t]/site-packages, 맥 OS 프레임워크 빌드의 경우 ~/Library/Python/*X.Y*/lib/python/site-packages, 윈도우의 경우 %*APPDATA*%\Python\Python*XY*\site-packages입니다. 선택적 "t" 는 자유 스레딩 빌드를 나타냅니다. 이 디렉터리는 사이트 디렉터리이며, 이는 그 안에 있는 .pth 파일이 처리됨을 의미합니다.

site.**USER_BASE**

사용자 site-packages의 베이스 디렉터리에 대한 경로. *getuserbase()*가 아직 호출되지 않았으면 None일 수 있습니다. 기본값은 유닉스와 맥 OS 비 프레임워크 빌드의 경우 ~/.local, 맥 OS 프레임워크 빌드의 경우 ~/Library/Python/*X.Y*, 윈도우의 경우 %*APPDATA*%\Python입니다. 이 값은 사용

자 설치 스킴의 스크립트, 데이터 파일, 파이썬 모듈 등의 설치 디렉터리를 계산하는 데 사용됩니다. PYTHONUSERBASE 도 참조하십시오.

site.**main**()

모든 표준 사이트별 디렉터리를 모듈 검색 경로에 추가합니다. 파이썬 인터프리터가 -S 플래그로 시작되지 않았으면, 이 모듈이 임포트 될 때 이 함수가 자동으로 호출됩니다.

버전 3.3에서 변경: 이 함수는 무조건 호출되었습니다.

site.**addsitedir**(*sitedir*, *known_paths=None*)

sys.path 에 디렉터리를 추가하고 .pth 파일을 처리합니다. 일반적으로 *sitecustomize*나 *usercustomize*에서 사용됩니다 (위를 참조하십시오).

site.**getsitepackages**()

모든 전역 site-packages 디렉터리를 포함하는 리스트를 반환합니다.

Added in version 3.2.

site.**getuserbase**()

사용자 베이스 디렉터리 *USER_BASE*의 경로를 반환합니다. 아직 초기화되지 않았으면, 이 함수는 PYTHONUSERBASE를 따라 설정합니다.

Added in version 3.2.

site.**getusersitepackages**()

사용자별 site-packages 디렉터리 *USER_SITE*의 경로를 반환합니다. 아직 초기화되지 않았으면, 이 함수는 *USER_BASE*를 따라 설정합니다. 사용자별 site-packages가 sys.path에 추가되었는지 확인하려면 *ENABLE_USER_SITE*를 사용해야 합니다.

Added in version 3.2.

29.16.5 명령 줄 인터페이스

site 모듈은 명령 줄에서 사용자 디렉터리를 얻는 방법도 제공합니다:

```
$ python -m site --user-site
/home/user/.local/lib/python3.11/site-packages
```

인자 없이 호출되면, 표준 출력에 *sys.path*의 내용을 인쇄한 다음, *USER_BASE*의 값과 디렉터리가 존재하는지를 인쇄하고, *USER_SITE*에 대해 같은 것을 인쇄하고, 마지막으로 *ENABLE_USER_SITE*의 값을 인쇄합니다.

--user-base

사용자 베이스 디렉터리의 경로를 인쇄합니다.

--user-site

사용자 site-packages 디렉터리의 경로를 인쇄합니다.

두 옵션이 모두 제공되면, *os.pathsep*으로 구분하여, 사용자 베이스와 사용자 사이트를 (항상 이 순서대로) 인쇄합니다.

어떤 옵션이건 제공되면, 스크립트는 다음 값 중 하나로 종료됩니다: 사용자 site-packages 디렉터리가 활성화되었으면 0, 사용자에 의해 비활성화되었으면 1, 보안상의 이유나 관리자에 의해 비활성화되었으면 2, 그리고 에러가 있으면 2보다 큰 값.

더 보기

- **PEP 370** – 사용자별 site-packages 디렉터리
- *The initialization of the sys.path module search path* – The initialization of *sys.path*.

더 보기

- See the `concurrent.interpreters` module, which similarly exposes core runtime functionality.

CHAPTER 30

사용자 정의 파이썬 인터프리터

이 장에서 설명하는 모듈은 파이썬의 대화형 인터프리터와 비슷한 인터페이스를 작성할 수 있도록 합니다. 파이썬 언어 외에도 몇 가지 특별한 기능을 지원하는 파이썬 인터프리터를 원한다면, *code* 모듈을 살펴보아야 합니다. (*codeop* 모듈은 더 저수준이며, 불완전할 수도 있는 파이썬 코드 조각의 컴파일을 지원하는 데 사용됩니다.)

이 장에서 설명하는 모듈의 전체 목록은 다음과 같습니다:

30.1 code — 인터프리터 베이스 클래스

소스 코드: Lib/code.py

---

code 모듈은 파이썬에서 REPL(read-eval-print loop)을 구현하는 기능을 제공합니다. 대화형 인터프리터 프롬프트를 제공하는 응용 프로그램을 만드는 데 사용할 수 있는 두 개의 클래스와 편리 함수들이 포함되어 있습니다.

class code.**InteractiveInterpreter**(*locals=None*)

이 클래스는 구문 분석과 인터프리터 상태(사용자의 이름 공간)를 처리합니다; 입력 버퍼링이나 프롬프트 또는 입력 파일 이름 지정을 처리하지 않습니다(파일명은 항상 명시적으로 전달됩니다). 선택적 *locals* 인자는 코드가 실행될 이름 공간으로 사용될 매핑을 지정합니다; 기본값은 키 '__name__'이 '__console__'로 설정되고 키 '__doc__'이 None으로 설정된 새로 만들어진 딕셔너리입니다.

Note that functions and classes objects created under an InteractiveInterpreter instance will belong to the namespace specified by *locals*. They are only pickleable if *locals* is the namespace of an existing module.

class code.**InteractiveConsole**(*locals=None, filename='<console>', local_exit=False*)

대화형 파이썬 인터프리터의 동작을 가깝게 흉내 냅니다. 이 클래스는 *InteractiveInterpreter*를 기반으로 하며 친숙한 sys.ps1과 sys.ps2를 사용하는 프롬프트와 입력 버퍼링을 추가합니다. *local_exit*가 참이면, 콘솔에서의 exit() 와 quit() 는 *SystemExit*를 발생시키지 않고, 대신 호출 코드로 돌아갑니다.

버전 3.13에서 변경: *local_exit* 매개 변수가 추가되었습니다.

code.**interact**(*banner=None, readfunc=None, local=None, exitmsg=None, local_exit=False*)

REPL(read-eval-print loop)를 실행하는 편리 함수. 이것은 *InteractiveConsole*의 새 인스턴스를 만들고, 제공된다면 *readfunc*가 *InteractiveConsole.raw_input()* 메서드로 사용되도록 설정합니다. *local*이 제공되면 인터프리터 루프의 기본 이름 공간으로 사용하기 위해 *InteractiveConsole* 생성자로 전달됩니다. *local_exit*가 제공되면 *InteractiveConsole* 생성자로 전달됩니다. 그런 다음

인스턴스의 `interact()` 메서드를 실행하는데, 제공된다면 *banner*와 *exitmsg*를 각각 배너와 종료 메시지로 사용하도록 전달합니다. 콘솔 객체는 사용 후에 폐기됩니다.

버전 3.6에서 변경: *exitmsg* 매개 변수가 추가되었습니다.

버전 3.13에서 변경: *local_exit* 매개 변수가 추가되었습니다.

code.**compile_command**(*source*, *filename*='<*input*>', *symbol*='*single*')

이 함수는 파이썬의 인터프리터 메인 루프(소위 REPL)를 흉내 내고 싶은 프로그램에 유용합니다. 까다로운 부분은 사용자가 후에 추가의 텍스트를 입력해서 완성할 수 있는 불완전한 명령을 입력했는지를 결정하는 것입니다(완전한 명령이나 문법 에러가 아니라). 이 함수는 거의 항상 실제 인터프리터 메인 루프와 같은 결정을 내립니다.

*source*는 소스 문자열입니다; *filename*은 소스를 읽어 들인 선택적 파일명이며, 기본값은 `'<input>'`입니다; 그리고 *symbol*은 선택적 문법 시작 기호이며 `'single'` (기본값), `'eval'` 또는 `'exec'` 중 하나여야 합니다.

명령이 완전하고 유효하면 코드 객체(`compile(source, filename, symbol)`와 같습니다)를 반환합니다; 명령이 불완전하면 `None`을 반환합니다; 명령이 완전하고 문법 에러가 있으면 `SyntaxError`를 발생시키고, 명령에 유효하지 않은 리터럴이 포함되었으면 `OverflowError`나 `ValueError`를 발생시킵니다.

30.1.1 대화형 인터프리터 객체

InteractiveInterpreter.**runsource**(*source*, *filename*='<*input*>', *symbol*='*single*')

인터프리터에서 소스를 컴파일하고 실행합니다. 인자는 `compile_command()`와 같습니다; *filename*의 기본값은 `'<input>'`이고, *symbol*의 기본값은 `'single'`입니다. 여러 가지 중 하나가 발생할 수 있습니다:

- 입력이 잘못되었습니다; `compile_command()`가 예외(`SyntaxError`나 `OverflowError`)를 발생시켰습니다. 문법 트레이스백이 `showsyntaxerror()` 메서드를 호출하여 인쇄됩니다. `runsource()`는 `False`를 반환합니다.
- 입력이 불완전하고, 더 많은 입력이 필요합니다; `compile_command()`가 `None`을 반환했습니다. `runsource()`는 `True`를 반환합니다.
- 입력이 완전합니다; `compile_command()`가 코드 객체를 반환했습니다. 코드는 `runcode()`(`SystemExit`를 제외한 실행 시간 예외도 처리합니다)를 호출하여 실행됩니다. `runsource()`는 `False`를 반환합니다.

반환 값은 다음 줄의 프롬프트에 `sys.ps1`과 `sys.ps2` 중 어느 것을 사용할지 결정하는 데 사용될 수 있습니다.

InteractiveInterpreter.**runcode**(*code*)

코드 객체를 실행합니다. 예외가 발생하면, `showtraceback()`가 호출되어 트레이스백을 표시합니다. 전파가 허락된 `SystemExit`를 제외한 모든 예외를 잡습니다.

`KeyboardInterrupt`에 대한 노트: 이 예외는 이 코드의 어딘가에서 발생할 수 있으며, 항상 잡히지는 않습니다. 호출자는 이것을 처리할 준비가 되어 있어야 합니다.

InteractiveInterpreter.**showsyntaxerror**(*filename*=*None*)

방금 발생한 문법 에러를 표시합니다. 스택 트레이스는 표시하지 않습니다, 문법 에러에는 그런 것이 없기 때문입니다. *filename*이 주어지면, 파이썬 구문 분석기가 제공하는 기본 파일명 대신에 예외에 채워집니다, 문자열에서 읽을 때는 항상 `'<string>'`을 사용하기 때문입니다. 출력은 `write()` 메서드로 기록됩니다.

InteractiveInterpreter.**showtraceback**()

방금 발생한 예외를 표시합니다. 첫 번째 스택 항목을 제거합니다, 그것은 인터프리터 객체 구현에 속하기 때문입니다. 출력은 `write()` 메서드로 기록됩니다.

버전 3.5에서 변경: 단지 기본(primary) 트레이스백이 아니라 전체 연결된(chained) 트레이스백이 표시됩니다.

`InteractiveInterpreter.write(data)`

문자열을 표준 에러 스트림(`sys.stderr`)에 기록합니다. 파생 클래스는 필요에 따라 적절한 출력 처리를 제공하기 위해 이것을 재정의해야 합니다.

30.1.2 대화형 콘솔 객체

`InteractiveConsole` 클래스는 `InteractiveInterpreter`의 서브 클래스이므로, 인터프리터 객체의 모든 메서드와 다음과 같은 추가 메서드를 제공합니다.

`InteractiveConsole.interact(banner=None, exitmsg=None)`

대화형 파이썬 콘솔을 가깝게 흉내 냅니다. 선택적 *banner* 인자는 첫 번째 상호 작용 전에 인쇄할 배너를 지정합니다; 기본적으로 표준 파이썬 인터프리터가 출력하는 것과 비슷한 배너를 출력한 다음 괄호 안에 콘솔 객체의 클래스 이름을 출력합니다 (실제 인터프리터와 혼동하지 않도록 하기 위함입니다 – 너무 비슷합니다!).

선택적 *exitmsg* 인자는 종료할 때 인쇄되는 종료 메시지를 지정합니다. 종료 메시지를 표시하지 않으려면 빈 문자열을 전달하십시오. *exitmsg*가 주어지지 않았거나 `None`이면, 기본 메시지가 인쇄됩니다.

버전 3.4에서 변경: 배너 인쇄를 억제하려면, 빈 문자열을 전달하십시오.

버전 3.6에서 변경: 종료할 때 종료 메시지를 인쇄합니다.

`InteractiveConsole.push(line)`

소스 텍스트 줄을 인터프리터로 밀어 넣습니다. line에는 후행 줄 바꿈이 없어야 합니다; 내부 줄 바꿈은 있을 수 있습니다. 줄은 버퍼에 추가되고 인터프리터의 `runsource()` 메서드가 이어붙인 버퍼의 내용을 소스로 하여 호출됩니다. 이것이 명령이 실행되었거나 유효하지 않았다고 알리면 버퍼는 재설정됩니다; 그렇지 않고 명령이 불완전하다면, 버퍼는 줄을 추가한 상태로 유지됩니다. 반환 값은 추가 입력이 필요하면 `True`이고, 어떤 식으로든 줄이 처리되었으면 `False`입니다 (`runsource()`와 같습니다).

`InteractiveConsole.resetbuffer()`

처리되지 않은 소스 텍스트를 입력 버퍼에서 제거합니다.

`InteractiveConsole.raw_input(prompt='')`

프롬프트를 기록하고 줄을 읽습니다. 반환된 줄에는 후행 줄 바꿈이 포함되지 않습니다. 사용자가 EOF 키 시퀀스를 입력하면, `EOFError`가 발생합니다. 기본 구현은 `sys.stdin`에서 읽습니다; 서브 클래스는 이것을 다른 구현으로 바꿀 수 있습니다.

30.2 codeop — 파이썬 코드 컴파일

소스 코드: Lib/codeop.py

`codeop` 모듈은 `code` 모듈에서와같이 파이썬 읽기-평가-인쇄 루프를 에뮬레이트 할 수 있는 유틸리티를 제공합니다. 결과적으로, 모듈을 직접 사용하고 싶지 않을 것입니다; 여러분의 프로그램에 이러한 루프를 포함 시키려면 대신 `code` 모듈을 사용하는 것이 좋습니다.

이 작업에는 두 가지 부분이 있습니다:

1. 입력 줄이 파이썬 문장을 완성하는지 알려주는 것: 간단히 말해서, '>>>' 나 '...' 를 다음에 인쇄할지 알려주기.
2. 사용자가 입력한 퓨처 문을 기억해서, 후속 입력을 컴파일할 때 이것들이 효과가 있도록 하기.

`codeop` 모듈은 이들을 각각 수행하는 방법과 이들을 모두 수행하는 방법을 제공합니다.

단지 전자를 수행하려면:

`codeop.compile_command(source, filename='<input>', symbol='single')`

*source*를 컴파일하려고 시도합니다. *source*는 파이썬 코드의 문자열이어야 하며, *source*가 유효한 파이썬 코드면 코드 객체를 반환합니다. 이 경우, 코드 객체의 filename 어트리뷰트는 *filename*가 되

는데, 기본값은 '<input>'입니다. *source*가 유효한 파이썬 코드가 *아니지만 유효한 파이썬 코드의 앞부분이면 None을 반환합니다.

*source*에 문제가 있으면, 예외가 발생합니다. 유효하지 않은 파이썬 구문이 있으면 *SyntaxError*가 발생하고, 유효하지 않은 리터럴이 있으면 *OverflowError* 나 *ValueError*가 발생합니다.

symbol 인자는 *source*가 문장('single', 기본값)으로 컴파일되는지, 문장의 시퀀스('exec')로 컴파일되는지 또는 표현식('eval')으로 컴파일되는지 결정합니다. 다른 값을 지정하면 *ValueError*가 발생합니다.

> **참고**
>
> 구문 분석기가 source의 끝에 도달하기 전에 성공적인 결과로 구문 분석을 중지하는 것이 가능합니다 (하지만 대체로 그렇지 않습니다); 이 경우, 뒤따르는 기호는 에러를 유발하는 대신 무시될 수 있습니다. 예를 들어, 백 슬래시 뒤에 두 개의 개행이 오면 그 뒤에 임의의 가비지가 올 수 있습니다. 구문 분석기를 위한 API가 개선되면 이 문제가 해결될 것입니다.

class codeop.**Compile**

이 클래스의 인스턴스는 내장 함수 *compile()*와 같은 서명의 __call__() 메서드를 갖지만, 인스턴스가 *__future__* 문을 포함하는 프로그램 텍스트를 컴파일하면 인스턴스가 이를 '기억' 하고 모든 후속 프로그램 텍스트를 이 문장의 효과 아래에서 컴파일한다는 차이점이 있습니다.

class codeop.**CommandCompiler**

이 클래스의 인스턴스는 *compile_command()*와 같은 서명의 __call__() 메서드를 갖습니다; 차이점은, 인스턴스가 *__future__* 문을 포함하는 프로그램 텍스트를 컴파일하면 인스턴스가 이를 '기억' 하고 모든 후속 프로그램 텍스트를 이 문장의 효과 아래에서 컴파일한다는 것입니다.

CHAPTER 31

모듈 임포트 하기

이 장에서 설명하는 모듈은 다른 파이썬 모듈을 임포트하는 새로운 방법과 임포트 절차를 사용자 정의하기 위한 훅을 제공합니다.

이 장에서 설명하는 모듈의 전체 목록은 다음과 같습니다:

31.1 `zipimport` — Zip 저장소에서 모듈 임포트

소스 코드: Lib/zipimport.py

이 모듈은 파이썬 모듈(`*.py`, `*.pyc`)과 패키지를 ZIP-형식 저장소에서 임포트하는 기능을 추가합니다. 일반적으로 *`zipimport`* 모듈을 명시적으로 사용할 필요는 없습니다; ZIP 저장소 경로가 *`sys.path`* 항목에 있으면 내장 `import` 메커니즘에 의해 자동으로 사용됩니다.

일반적으로, *`sys.path`*는 문자열 디렉터리 이름의 리스트입니다. 이 모듈은 또한 *`sys.path`* 항목이 ZIP 파일 저장소를 명명하는 문자열이 될 수 있도록 합니다. ZIP 저장소에는 패키지 임포트를 지원하는 하위 디렉터리 구조가 포함될 수 있으며, 저장소 내의 경로를 지정하여 하위 디렉터리에서만 임포트 되도록 할 수 있습니다. 예를 들어, 경로 `example.zip/lib/`는 저장소 내의 `lib/` 서브 디렉터리에서만 임포트하도록 합니다.

어떤 파일이든 ZIP 저장소에 있을 수 있지만, 임포터는 `.py` 와 `.pyc` 파일에 대해서만 호출됩니다. 동적 모듈(`.pyd`, `.so`)의 ZIP 임포트는 허용되지 않습니다. 저장소에 `.py` 파일만 포함되어있으면, 파이썬은 해당 `.pyc` 파일을 추가하여 저장소를 수정하지 않습니다. 즉, ZIP 저장소에 `.pyc` 파일이 포함되어 있지 않으면, 임포트가 다소 느릴 수 있습니다.

버전 3.13에서 변경: ZIP64 를 지원합니다

버전 3.8에서 변경: 전에는, 저장소 주석이 포함된 ZIP 저장소는 지원되지 않았습니다.

더 보기

PKZIP Application Note
사용된 형식과 알고리즘 저자인 Phil Katz의 ZIP 파일 형식에 관한 설명서.

PEP 273 - Zip 저장소에서 모듈 임포트
구현도 제공한 James C. Ahlstrom이 작성했습니다. 파이썬 2.3은 **PEP 273**의 명세를 따르지만, Just van Rossum이 작성한 구현을 사용하는데 **PEP 302**에 설명된 임포트 훅을 사용합니다.

> *importlib* - **The implementation of the import machinery**
> Package providing the relevant protocols for all importers to implement.

이 모듈은 예외를 정의합니다:

exception zipimport.**ZipImportError**

zipimporter 객체가 발생시키는 예외. *ImportError*의 서브 클래스이므로, *ImportError*로도 잡힐 수 있습니다.

31.1.1 zipimporter 객체

*zipimporter*는 ZIP 파일을 임포트하는 클래스입니다.

class zipimport.**zipimporter**(*archivepath*)

새로운 zipimporter 인스턴스를 만듭니다. *archivepath*는 ZIP 파일의 경로이거나, ZIP 파일 내의 특정 경로여야 합니다. 예를 들어, *archivepath* `foo/bar.zip/lib`는 ZIP 파일 `foo/bar.zip` 내의 `lib` 디렉터리에 있는 모듈을 찾습니다 (존재한다면).

*archivepath*가 유효한 ZIP 저장소를 가리키지 않으면 *ZipImportError*가 발생합니다.

버전 3.12에서 변경: Methods `find_loader()` and `find_module()`, deprecated in 3.10 are now removed. Use *find_spec()* instead.

create_module(*spec*)

Implementation of *importlib.abc.Loader.create_module()* that returns *None* to explicitly request the default semantics.

Added in version 3.10.

exec_module(*module*)

Implementation of *importlib.abc.Loader.exec_module()*.

Added in version 3.10.

find_spec(*fullname*, *target=None*)

An implementation of *importlib.abc.PathEntryFinder.find_spec()*.

Added in version 3.10.

get_code(*fullname*)

지정된 모듈의 코드 객체를 반환합니다. 모듈을 임포트할 수 없으면 *ZipImportError*를 발생시킵니다.

get_data(*pathname*)

*pathname*와 관련된 데이터를 반환합니다. 파일을 찾을 수 없으면 *OSError*를 발생시킵니다.

버전 3.3에서 변경: *IOError* 를 발생시켜왔습니다, 이제 *OSError*의 별칭입니다.

get_filename(*fullname*)

지정한 모듈이 임포트될 때 설정될 `__file__`의 값을 반환합니다. 모듈을 임포트할 수 없으면 *ZipImportError*를 발생시킵니다.

Added in version 3.1.

get_source(*fullname*)

지정된 모듈의 소스 코드를 반환합니다. 모듈을 찾을 수 없으면 *ZipImportError*를 발생시키고, 저장소에 모듈이 있지만, 소스가 없으면 *None*을 반환합니다.

is_package(*fullname*)

*fullname*으로 지정된 모듈이 패키지면 `True`를 반환합니다. 모듈을 찾을 수 없으면 *ZipImportError*를 발생시킵니다.

load_module(*fullname*)

*fullname*으로 지정된 모듈을 로드 합니다. *fullname*은 완전히 정규화된 (점으로 구분된) 모듈 이름이어야 합니다. 성공하면 임포트 된 모듈을 반환하고, 실패하면 `ZipImportError`를 발생시킵니다.

Deprecated since version 3.10, will be removed in version 3.15: Use `exec_module()` instead.

invalidate_caches()

Clear out the internal cache of information about files found within the ZIP archive.

Added in version 3.10.

archive

있을 수도 있는 하위 경로를 제외한, 임포터와 연결된 ZIP 파일의 파일 이름.

prefix

모듈이 검색되는 ZIP 파일 내의 하위 경로. ZIP 파일의 루트를 가리키는 zipimporter 객체에서는 빈 문자열입니다.

`archive`와 `prefix` 어트리뷰트는, 슬래시로 결합 될 때, `zipimporter` 생성자에 지정된 원래 *archivepath* 인자와 같습니다.

31.1.2 예제

다음은 ZIP 저장소에서 모듈을 임포트하는 예제입니다 - `zipimport` 모듈이 명시적으로 사용되지 않음에 유의하십시오.

```
$ unzip -l example_archive.zip
Archive:  example_archive.zip
  Length      Date    Time    Name
 --------    ----    ----    ----
     8467  01-01-00 12:30   example.py
 --------                   -------
     8467                   1 file
```

```
>>> import sys
>>> # Add the archive to the front of the module search path
>>> sys.path.insert(0, 'example_archive.zip')
>>> import example
>>> example.__file__
'example_archive.zip/example.py'
```

31.2 pkgutil — 패키지 확장 유틸리티

소스 코드: Lib/pkgutil.py

이 모듈은 임포트 시스템, 특히 패키지 지원을 위한 유틸리티를 제공합니다.

class pkgutil.**ModuleInfo**(*module_finder*, *name*, *ispkg*)

모듈 정보에 대한 간략한 요약을 담고 있는 네임드 튜플.

Added in version 3.6.

pkgutil.**extend_path**(*path*, *name*)

패키지를 구성하는 모듈의 검색 경로를 확장합니다. 의도된 사용법은 패키지의 `__init__.py`에 다음 코드를 삽입하는 것입니다:

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

sys.path에 있는 디렉터리들의 모든 서브 디렉터리 중, 패키지의 이름과 일치하는 것들을 패키지의 __path__에 추가합니다. 이것은 하나의 논리적 패키지의 부분들을 여러 디렉터리로 분배하려고 할 때 유용합니다.

*가 *name* 인자와 일치하는 *.pkg 파일도 찾습니다. 이 기능은 import로 시작하는 줄을 특수하게 다루지 않는다는 점을 제외하면, *.pth 파일과 유사합니다 (자세한 내용은 site 모듈을 참조하십시오). *.pkg 파일을 액면 그대로 신뢰합니다: 빈 줄을 건너뛰고 주석을 무시하지만, *.pkg 파일에 있는 모든 항목은 파일 시스템에 있는지에 관계없이 경로에 추가됩니다. (이것은 기능입니다.)

입력 경로가 리스트가 아니면 (프로즌 패키지의 경우처럼) 변경되지 않은 상태로 반환됩니다. 입력 경로는 수정되지 않습니다; 확장한 사본이 반환됩니다. 항목은 사본의 끝에 추가되기만 합니다.

sys.path기 시퀀스라고 가정합니다. 존재하는 디렉터리를 참조하는 문자열이 아닌 sys.path 항목은 무시됩니다. 파일명으로 사용될 때 에러를 일으키는 sys.path의 유니코드 항목은 이 함수가 예외를 발생시키도록 할 수 있습니다 (os.path.isdir() 동작과 일치합니다).

pkgutil.**get_importer**(*path_item*)

주어진 *path_item*에 대한 파인더를 가져옵니다.

반환된 파인더는 경로 훅 때문에 새로 만들어지면 sys.path_importer_cache에 캐시 됩니다.

sys.path_hooks의 재검색이 필요하면, 캐시(또는 그 일부)를 수동으로 지울 수 있습니다.

버전 3.3에서 변경: 패키지 내부 **PEP 302** 임포트 에뮬레이션에 의존하는 대신, importlib에 직접 기반하도록 갱신되었습니다.

pkgutil.**iter_importers**(*fullname=''*)

주어진 모듈 이름에 대해 파인더 객체를 산출(yield)합니다.

If *fullname* contains a '.', the finders will be for the package containing *fullname*, otherwise they will be all registered top level finders (i.e. those on both sys.meta_path and sys.path_hooks).

명명된 모듈이 패키지에 있으면, 이 함수를 호출하는 부작용으로 그 패키지를 임포트 합니다.

모듈 이름을 지정하지 않으면, 모든 최상위 수준 파인더가 생성됩니다.

버전 3.3에서 변경: 패키지 내부 **PEP 302** 임포트 에뮬레이션에 의존하는 대신, importlib에 직접 기반하도록 갱신되었습니다.

pkgutil.**iter_modules**(*path=None, prefix=''*)

*path*의 모든 서브 모듈에 대한 ModuleInfo를, 또는 *path*가 None이면, sys.path에 있는 모든 최상위 모듈을 산출(yield)합니다.

*path*는 None이거나 모듈을 찾을 경로의 리스트이어야 합니다.

*prefix*는 출력 시 모든 모듈 이름 앞에 출력할 문자열입니다.

> **참고**
>
> iter_modules() 메서드를 정의하는 파인더에서만 작동합니다. 이 인터페이스는 비표준이므로, 모듈은 importlib.machinery.FileFinder와 zipimport.zipimporter에 대한 구현도 제공합니다.

버전 3.3에서 변경: 패키지 내부 **PEP 302** 임포트 에뮬레이션에 의존하는 대신, importlib에 직접 기반하도록 갱신되었습니다.

pkgutil.**walk_packages**(*path=None, prefix='', onerror=None*)

*path*에 재귀적으로 포함된 모든 모듈이나, *path*가 None이면 모든 액세스할 수 있는 모듈에 대한 ModuleInfo를 산출(yield)합니다.

*path*는 None이거나 모듈을 찾을 경로의 리스트이어야 합니다.

*prefix*는 출력 시 모든 모듈 이름 앞에 출력할 문자열입니다.

서브 모듈 검색을 위한 __path__ 어트리뷰트에 액세스하기 위해, 이 함수는 주어진 *path*에 있는 모든 패키지(모든 모듈이 아닙니다!)를 임포트 해야 함에 유의하십시오.

*onerror*는 패키지 임포트를 시도하는 동안 예외가 발생하면 하나의 인자(임포트 하려는 패키지의 이름)로 호출되는 함수입니다. *onerror* 함수가 제공되지 않으면, `ImportError`는 잡아서 무시하고, 다른 모든 예외는 전파되어 검색이 종료됩니다.

예제:

```
# 파이썬이 액세스할 수 있는 모든 모듈을 나열합니다
walk_packages()

# ctypes의 모든 서브 모듈을 나열합니다
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

> **참고**
>
> `iter_modules()` 메서드를 정의하는 파인더에서만 작동합니다. 이 인터페이스는 비표준이므로, 모듈은 `importlib.machinery.FileFinder`와 `zipimport.zipimporter`에 대한 구현도 제공합니다.

버전 3.3에서 변경: 패키지 내부 **PEP 302** 임포트 에뮬레이션에 의존하는 대신, `importlib`에 직접 기반하도록 갱신되었습니다.

`pkgutil.`**`get_data`**(*package*, *resource*)

패키지에서 리소스를 가져옵니다.

이것은 로더 `get_data` API에 대한 래퍼입니다. *package* 인자는 표준 모듈 형식(`foo.bar`)의 패키지 이름이어야 합니다. *resource* 인자는 `/`를 경로 분리자로 사용하는 상대 파일명의 형식이어야 합니다. 상위 디렉터리 이름 `..`는 허용되지 않으며, 루트에서 시작하는(`/`로 시작하는) 이름도 허용되지 않습니다.

이 함수는 지정된 리소스의 내용인 바이트열을 반환합니다.

파일시스템에 있는 패키지(이미 임포트 되었습니다)의 경우, 이것은 대략 다음과 동등합니다:

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

패키지를 찾거나 로드 할 수 없거나, 패키지가 `get_data`를 지원하지 않는 로더를 사용하면, `None`이 반환됩니다. 특히, 이름 공간 패키지를 위한 로더는 `get_data`를 지원하지 않습니다.

`pkgutil.`**`resolve_name`**(*name*)

이름을 객체로 해석합니다.

이 기능은 표준 라이브러리의 여러 곳에서 사용됩니다 (bpo-12915를 참조하십시오) - 그리고 동등한 기능이 setuptools, Django 및 Pyramid와 같은 널리 사용되는 제삼자 패키지에도 있습니다.

*name*은 다음 형식 중 하나의 문자열 일 것으로 기대됩니다, 여기서 W는 유효한 파이썬 식별자를 나타내는 줄임 표현이며 점은 이러한 의사 정규식에서 리터럴 마침표를 나타냅니다:

- `W(.W)*`
- `W(.W)*:(W(.W)*)?`

첫 번째 형식은 이전 버전과의 호환성을 위해서만 사용됩니다. 점으로 구분된 이름의 일부는 패키지이고, 나머지는 패키지 내의 어딘가에 있는 객체이며, 다른 객체 안에 중첩되었을 수 있습니다. 패키지가 멈추고 객체 계층 구조가 시작되는 위치는 보는 것 만으로는 유추할 수 없습니다, 이 형식으로는 임포트 시도를 반복해서 수행해야합니다.

두 번째 형식에서, 호출자는 단일 콜론을 제공하여 구분 지점을 명확하게 만듭니다: 콜론 왼쪽의 점으로 구분된 이름은 임포트할 패키지이고, 오른쪽의 점으로 구분된 이름은 해당 패키지 내의 객체 계층 구조입니다. 이 형식에서는 한 번의 임포트만 필요합니다. 콜론으로 끝나면, 모듈 객체가 반환됩니다.

이 함수는 객체(모듈일 수 있습니다)를 반환하거나, 다음 예외 중 하나를 발생시킵니다:

ValueError – *name*이 인식되는 형식이 아니면.

ImportError – 그러지 말아야할 때 임포트가 실패하면.

AttributeError – 임포트한 패키지 내에서 객체 계층 구조를 탐색하여 원하는 객체에 도달하는 도중 실패가 발생할 때.

Added in version 3.9.

31.3 modulefinder — 스크립트에서 사용되는 모듈 찾기

소스 코드: Lib/modulefinder.py

이 모듈은 스크립트가 임포트 한 모듈 집합을 판단하는 데 사용할 수 있는 *ModuleFinder* 클래스를 제공합니다. `modulefinder.py`는 스크립트로 실행될 수도 있습니다, 인자로 파이썬 스크립트의 파일 이름을 지정하면, 임포트 된 모듈의 보고서가 인쇄됩니다.

`modulefinder.`**`AddPackagePath`**(*pkg_name*, *path*)

지정된 *path*에서 *pkg_name* 패키지를 찾을 수 있음을 기록합니다.

`modulefinder.`**`ReplacePackage`**(*oldname*, *newname*)

oldname 라는 이름의 모듈이 실제로는 *newname*라는 이름의 패키지라는 것을 지정할 수 있도록 합니다.

class `modulefinder.`**`ModuleFinder`**(*path=None*, *debug=0*, *excludes=[]*, *replace_paths=[]*)

이 클래스는 스크립트가 임포트하는 모듈 집합을 판단하는 *run_script()* 와 *report()* 메서드를 제공합니다. *path*는 모듈을 검색할 디렉터리 리스트일 수 있습니다; 지정되지 않으면, `sys.path`가 사용됩니다. *debug*는 디버깅 수준을 설정합니다; 값이 크면 클래스가 수행 중인 작업에 대한 디버깅 메시지를 인쇄합니다. *excludes*는 분석에서 제외할 모듈 이름 리스트입니다. *replace_paths*는 모듈 경로에서 교체될 `(oldpath, newpath)` 튜플의 리스트입니다.

`report`()

빠지거나 빠진 것으로 보이는 모듈뿐 아니라, 스크립트가 임포트하는 모듈과 그들의 경로의 목록에 관한 보고서를 표준 출력으로 인쇄합니다.

`run_script`(*pathname*)

파이썬 코드를 포함하는, *pathname* 파일의 내용을 분석합니다.

`modules`

모듈 이름을 모듈에 매핑하는 딕셔너리. *ModuleFinder의 사용 예*를 참조하십시오.

31.3.1 ModuleFinder의 사용 예

나중에 분석할 스크립트 (bacon.py):

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

bacon.py의 보고서를 출력하는 스크립트:

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('Loaded modules:')
for name, mod in finder.modules.items():
    print('%s: ' % name, end='')
    print(',.join(list(mod.globalnames.keys())[:3]))

print('-'*50)
print('Modules not imported:')
print('\n'.join(finder.badmodules.keys()))
```

표본 출력(아키텍처에 따라 다를 수 있습니다):

```
Loaded modules:
_types:
copyreg:  _inverted_registry,_slotnames,__all__
re._compiler:  isstring,_sre,_optimize_unicode
_sre:
re._constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
re._parser:  _PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
----------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
```

31.4 runpy — 파이썬 모듈 찾기와 실행

소스 코드: Lib/runpy.py

---

runpy 모듈은 파이썬 모듈을 먼저 임포트 하지 않고 찾아서 실행하는 데 사용합니다. 주요 용도는 파일 시스템이 아닌 파이썬 모듈 이름 공간을 사용하여 스크립트를 찾을 수 있는 `-m` 명령 줄 스위치를 구현하는 것입니다.

이것은 샌드박스 모듈이 아닙니다 - 모든 코드가 현재 프로세스에서 실행되고, 모든 부작용(가령 다른 모듈의 캐시된 임포트)은 함수가 반환된 후에도 그대로 유지됩니다.

또한, 실행된 코드에서 정의된 모든 함수와 클래스는 *runpy* 함수가 반환된 후 올바르게 작동하지 않을 수 있습니다. 이러한 제한이 주어진 사용 사례에 적합하지 않으면, 이 모듈보다 *importlib*가 더 적합한 선택일 수 있습니다.

runpy 모듈은 두 가지 함수를 제공합니다:

runpy.**run_module**(*mod_name*, *init_globals=None*, *run_name=None*, *alter_sys=False*)

지정된 모듈의 코드를 실행하고 결과 모듈의 전역 딕셔너리를 반환합니다. 모듈의 코드는 먼저 표준 임포트 메커니즘(자세한 내용은 **PEP 302**를 참조하십시오)을 사용하여 찾은 다음 새로운 모듈 이름 공간에서 실행됩니다.

mod_name 인자는 절대 모듈 이름이어야 합니다. 모듈 이름이 일반 모듈이 아닌 패키지를 참조하면, 해당 패키지를 임포트하고 그 패키지 내의 `__main__` 서브 모듈을 실행하고 결과 모듈 전역 딕셔너리를 반환합니다.

선택적 딕셔너리 인자 *init_globals*는 코드가 실행되기 전에 모듈의 전역 딕셔너리를 미리 채우기 위해 사용될 수 있습니다. *init_globals*는 수정되지 않습니다. 아래의 특수 전역 변수가 *init_globals*에 정의되어 있으면, 해당 정의가 `run_module()`에 의해 대체됩니다.

특수 전역 변수 `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` 및 `__package__`는 모듈 코드가 실행되기 전에 전역 딕셔너리에 설정됩니다. (이 변수는 최소한의 변수 집합임에 유의하십시오 - 인터프리터 구현 세부 사항에 따라 다른 변수가 묵시적으로 설정될 수 있습니다.)

`__name__`은 (이 선택적 인자가 `None`이 아니면) *run_name*으로, 명명된 모듈이 패키지면 `mod_name + '.__main__'`으로, 그렇지 않으면 *mod_name* 인자로 설정됩니다.

`__spec__`은 실제로 임포트 된 모듈에 맞게 설정됩니다 (즉, `__spec__.name`은 항상 *mod_name*이나 `mod_name + '.__main__'`이 됩니다, 절대 *run_name*은 아닙니다).

`__file__`, `__cached__`, `__loader__` 및 `__package__`는 모듈 스펙에 따라 표준적으로 설정됩니다.

인자 *alter_sys*가 제공되고 `True`로 평가되면, `sys.argv[0]`은 `__file__` 값으로 갱신되고 `sys.modules[__name__]`은 실행 중인 모듈에 대한 임시 모듈 객체로 갱신됩니다. `sys.argv[0]`과 `sys.modules[__name__]`은 함수가 반환되기 전에 원래 값으로 복원됩니다.

이 `sys` 조작은 스레드-안전하지 않습니다. 다른 스레드가 부분적으로 초기화된 모듈과 변경된 인자 목록을 볼 수 있습니다. 스레드를 사용하는 코드에서 이 함수를 호출할 때 `sys` 모듈을 단독으로 두는 것이 좋습니다.

더 보기

명령 줄에서 동등한 기능을 제공하는 `-m` 옵션.

버전 3.1에서 변경: `__main__` 서브 모듈을 찾아 패키지를 실행할 수 있는 기능 추가.

버전 3.2에서 변경: `__cached__` 전역 변수 추가 (**PEP 3147**을 참조하십시오).

버전 3.4에서 변경: **PEP 451**이 추가한 모듈 스펙 기능을 활용하도록 갱신되었습니다. 이것은 실제 모듈 이름을 항상 `__spec__.name`으로 액세스할 수 있으면서, `__cached__`가 이 방법으로 실행되는 모듈에 대해 올바르게 설정되도록 합니다.

버전 3.12에서 변경: `__cached__`, `__loader__` 및 `__package__`의 설정은 폐지되었습니다. 대안은 `ModuleSpec`을 참조하세요.

`runpy.`**`run_path`**(*path_name*, *init_globals=None*, *run_name=None*)

명명된 파일 시스템 위치에 있는 코드를 실행하고 결과 모듈의 전역 딕셔너리를 반환합니다. CPython 명령 줄에 제공된 스크립트 이름과 마찬가지로, *file_path*는 파이썬 소스 파일, 컴파일된 바이트 코드 파일 또는 `__main__` 모듈이 포함된 유효한 `sys.path` 항목(예를 들어, 최상위 수준 `__main__.py` 파일을 포함하는 zip 파일)을 가리킬 수 있습니다.

간단한 스크립트의 경우, 지정된 코드는 단순히 새로운 모듈 이름 공간에서 실행됩니다. 유효한 `sys.path` 항목(보통 zip 파일이나 디렉터리)의 경우, 항목이 먼저 `sys.path`의 시작 부분에 추가됩니다. 그런 다음 함수는 갱신된 경로를 사용하여 `__main__` 모듈을 찾아 실행합니다. 지정된 위치에 해당 모듈이 없을 때 `sys.path`의 다른 위치에 있는 기존 `__main__` 항목을 호출하는 것을 막는 특별한 보호 장치가 없다는 점에 유의하십시오.

선택적 딕셔너리 인자 *init_globals*는 코드가 실행되기 전에 모듈의 전역 딕셔너리를 미리 채우기 위해 사용될 수 있습니다. *init_globals*는 수정되지 않습니다. 아래의 특수 전역 변수가 *init_globals*에 정의되어 있으면, 해당 정의가 `run_path()`에 의해 대체됩니다.

특수 전역 변수 `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` 및 `__package__`는 모듈 코드가 실행되기 전에 전역 딕셔너리에 설정됩니다. (이 변수는 최소한의 변수 집합임에 유의하십시오 - 인터프리터 구현 세부 사항에 따라 다른 변수가 묵시적으로 설정될 수 있습니다.)

__name__은 (이 선택적 인자가 None이 아니면) *run_name*으로, 그렇지 않으면 '<run_path>'로 설정됩니다.

*file_path*가 스크립트 파일(소스나 사전 컴파일된 바이트 코드)을 직접 참조하면, __file__은 *file_path*로 설정되고 __spec__, __cached__, __loader__ 및 __package__는 모두 None으로 설정됩니다.

*file_path*가 유효한 sys.path 항목에 대한 참조면, __spec__은 임포트 된 __main__ 모듈에 대해 적절하게 설정됩니다 (즉, __spec__.name은 항상 __main__이 됩니다). __file__, __cached__, __loader__ 및 __package__는 모듈 스펙에 따라 표준적으로 설정됩니다.

sys 모듈에도 여러 가지 변경이 적용됩니다. 첫째, sys.path는 위에서 설명한 것처럼 변경될 수 있습니다. sys.argv[0]은 *file_path* 값으로 갱신되고 sys.modules[__name__]은 실행 중인 모듈에 대한 임시 모듈 객체로 갱신됩니다. 함수가 반환되기 전에 sys의 항목에 대한 모든 수정 내용이 되돌려집니다.

run_module()과 달리, sys에 대한 변경은 이 함수에서는 선택 사항이 아닌데, 이 조정이 sys.path 항목의 실행을 허용하는 데 필수적이기 때문입니다. 스레드-안전 제약 사항이 계속 적용되므로, 스레드를 사용하는 코드에서 이 함수를 사용하려면 임포트 잠금을 사용하여 직렬화하거나 별도의 프로세스에 위임해야 합니다.

더 보기

명령 줄에서의 동등한 기능에 대한 using-on-interface-options (python path/to/script).

Added in version 3.2.

버전 3.4에서 변경: **PEP 451**이 추가한 모듈 스펙 기능을 활용하도록 갱신되었습니다. 이것은 __main__이 직접 실행되는 대신 유효한 sys.path 항목에서 임포트 될 때 __cached__가 올바르게 설정되도록 합니다.

버전 3.12에서 변경: __cached__, __loader__ 및 __package__의 설정은 폐지되었습니다.

더 보기

PEP 338 – 모듈을 스크립트로 실행하기
Nick Coghlan이 작성하고 구현한 PEP.

PEP 366 – 메인 모듈 명시적 상대 임포트
Nick Coghlan이 작성하고 구현한 PEP.

PEP 451 – 임포트 시스템의 ModuleSpec 형
Eric Snow가 작성하고 구현한 PEP

using-on-general - CPython 명령 줄 세부 사항

importlib.import_module() 함수

31.5 importlib — import의 구현

Added in version 3.1.

소스 코드: Lib/importlib/__init__.py

31.5.1 소개

importlib 패키지의 목적은 세 가지입니다.

하나는 파이썬 소스 코드에서 import 문(그리고, 확장하면 __import__() 함수)의 구현을 제공하는 것입니다. 이것은 모든 파이썬 인터프리터에 이식할 수 있는 import의 구현을 제공합니다. 또한 파이썬 이외의 프로그래밍 언어로 구현된 것보다 이해하기 쉬운 구현을 제공합니다.

둘째, `import`를 구현하는 구성 요소가 이 패키지에서 노출되어, 사용자가 임포트 프로세스에 참여하기 위해 자신의 사용자 지정 객체(일반적으로 임포터라고 합니다)를 쉽게 만들 수 있도록 합니다.

Three, the package contains modules exposing additional functionality for managing aspects of Python packages:

- *importlib.metadata* presents access to metadata from third-party distributions.
- *importlib.resources* provides routines for accessing non-code "resources" from Python packages.

더 보기

import
`import` 문의 언어 레퍼런스.

패키지 명세
패키지의 원래 명세. 이 문서를 작성한 이후로 일부 의미가 변경되었습니다 (예를 들어 *sys.modules*의 `None`을 기반으로 하는 리디렉션).

***__import__()* 함수**
`import` 문은 이 함수의 편의 문법입니다.

The initialization of the sys.path module search path
The initialization of *sys.path*.

PEP 235
대소 문자를 구분하지 않는 플랫폼에서의 임포트

PEP 263
파이썬 소스 코드 인코딩 정의

PEP 302
새로운 임포트 훅

PEP 328
임포트: 다중 줄과 절대/상대

PEP 366
메인 모듈 명시적 상대 임포트

PEP 420
묵시적 이름 공간 패키지

PEP 451
임포트 시스템을 위한 ModuleSpec 형

PEP 488
PYO 파일 제거

PEP 489
다단계 확장 모듈 초기화

PEP 552
결정론적 pyc

PEP 3120
UTF-8을 기본 소스 인코딩으로 사용하기

PEP 3147
PYC 저장소 디렉터리

31.5.2 함수

`importlib.__import__`(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)

내장 *__import__()* 함수의 구현.

> **참고**
>
> 프로그래밍 방식으로 모듈을 임포트 하려면 이 함수 대신 *import_module()*을 사용해야 합니다.

importlib.**import_module**(*name*, *package=None*)

모듈을 임포트 합니다. *name* 인자는 절대나 상대적인 항으로 임포트 할 모듈을 지정합니다 (예를 들어 pkg.mod나 ..mod). 이름이 상대적인 항으로 지정되면, *package* 인자는 패키지 이름을 결정하기 위한 앵커 역할을 하는 패키지 이름으로 설정해야 합니다 (예를 들어 import_module('..mod', 'pkg.subpkg')는 pkg.mod를 임포트 합니다).

import_module() 함수는 *importlib.__import__()* 주위를 감싸는 단순화 래퍼 역할을 합니다. 이는 함수의 모든 의미가 *importlib.__import__()*에서 파생됨을 뜻합니다. 이 두 함수의 가장 중요한 차이점은 *import_module()*이 지정된 패키지나 모듈(예를 들어 pkg.mod)을 반환하는 반면, *__import__()*는 최상위 패키지나 모듈(예를 들어 pkg)을 반환한다는 것입니다.

인터프리터가 실행을 시작한 이후 만들어진 모듈(예를 들어, 파이썬 소스 파일을 만들면)을 동적으로 임포트 하는 경우, 임포트 시스템에서 새 모듈을 알 수 있도록 *invalidate_caches()*를 호출해야 할 수 있습니다.

버전 3.3에서 변경: 부모 패키지는 자동으로 임포트 됩니다.

importlib.**invalidate_caches**()

*sys.meta_path*에 저장된 파인더의 내부 캐시를 무효로 합니다. 파인더가 invalidate_caches() 를 구현하면 무효화를 수행하기 위해 호출됩니다. 모든 파인더가 새로운 모듈의 존재를 알 수 있도록 프로그램이 실행되는 동안 모듈이 만들어진/설치된 경우 이 함수를 호출해야 합니다.

Added in version 3.3.

버전 3.10에서 변경: Namespace packages created/installed in a different *sys.path* location after the same namespace was already imported are noticed.

importlib.**reload**(*module*)

이전에 임포트 한 *module*을 다시 로드합니다. 인자는 모듈 객체여야 해서, 이전에 성공적으로 임포트 됐어야 합니다. 외부 편집기를 사용하여 모듈 소스 파일을 편집했고 파이썬 인터프리터를 떠나지 않고 새 버전을 시험해보고 싶을 때 유용합니다. 반환 값은 모듈 객체입니다 (재 임포트로 인해 다른 객체가 *sys.modules*에 배치되면 다를 수 있습니다).

*reload()*가 실행될 때:

- 파이썬 모듈의 코드가 다시 컴파일되고 모듈 수준 코드가 다시 실행되어, 원래 모듈을 로드한 로더를 재사용하여 모듈 딕셔너리에 있는 이름에 연결되는 새로운 객체 집합을 정의합니다. 확장 모듈의 init 함수는 두 번째에는 호출되지 않습니다.
- 파이썬의 다른 모든 객체와 마찬가지로 이전 객체는 참조 횟수가 0으로 떨어진 후에만 자원이 회수됩니다.
- 모듈 이름 공간의 이름은 새로운 객체나 변경된 객체를 가리키도록 갱신됩니다.
- 이전 객체에 대한 다른 참조(가령 모듈 외부의 이름)는 새 객체를 참조하기 위해 다시 연결되지 않으며 필요하다면 그들이 등장하는 각 이름 공간에서 갱신되어야 합니다.

다른 여러 가지 경고가 있습니다:

모듈을 다시 로드할 때, 그것의 (모듈의 전역 변수를 포함하는) 딕셔너리가 유지됩니다. 이름을 재정의하면 이전 정의를 대체해서, 일반적으로 문제가 되지 않습니다. 새 버전의 모듈이 이전 버전이 정의한 이름을 정의하지 않으면, 이전 정의가 그대로 남습니다. 이 기능은 객체의 전역 테이블이나 캐시를 유지한다면 모듈의 이점으로 사용될 수 있습니다 — try 문으로 테이블의 존재를 검사하고 필요하다면 초기화를 건너뛸 수 있습니다:

```
try:
    cache
except NameError:
    cache = {}
```

일반적으로 내장이나 동적으로 로드된 모듈을 다시 로드하는 것은 그리 유용하지 않습니다. *sys*, *__main__*, *builtins* 및 기타 주요 모듈을 다시 로드하지 않는 것이 좋습니다. 많은 경우 확장 모듈은 두 번 이상 초기화되도록 설계되지 않았으며, 다시 로드할 때 임의의 방식으로 실패할 수 있습니다.

모듈이 `from … import …`를 사용하여 다른 모듈에서 객체를 임포트 하면, 다른 모듈에 대해 *reload()*를 호출해도 그것에서 임포트 한 객체를 재정의하지 않습니다 — 이것을 피하는 한 가지 방법은 `from` 문을 다시 실행하는 것입니다, 다른 방법은 대신 `import`와 정규화된 이름(*module.name*)을 사용하는 것입니다.

모듈이 클래스의 인스턴스를 인스턴스 화하면, 클래스를 정의하는 모듈을 다시 로드해도 인스턴스의 메서드 정의에는 영향을 미치지 않습니다 — 이전 클래스 정의를 계속 사용합니다. 파생 클래스의 경우도 마찬가지입니다.

Added in version 3.4.

버전 3.7에서 변경: 다시 로드되는 모듈에 *ModuleSpec*이 없으면 *ModuleNotFoundError* 가 발생합니다.

> **경고**
>
> This function is not thread-safe. Calling it from multiple threads can result in unexpected behavior. It's recommended to use the *threading.Lock* or other synchronization primitives for thread-safe module reloading.

31.5.3 `importlib.abc` – import와 관련된 추상 베이스 클래스

소스 코드: Lib/importlib/abc.py

---

importlib.abc 모듈에는 `import`에서 사용하는 모든 핵심 추상 베이스 클래스가 포함되어 있습니다. 핵심 ABC 구현에 도움이 되도록 핵심 추상 베이스 클래스의 일부 서브 클래스도 제공됩니다.

ABC 계층:

```
object
 +-- MetaPathFinder
 +-- PathEntryFinder
 +-- Loader
      +-- ResourceLoader --------+
      +-- InspectLoader          |
           +-- ExecutionLoader --+
                                 +-- FileLoader
                                 +-- SourceLoader
```

class importlib.abc.**MetaPathFinder**

메타 경로 파인더를 나타내는 추상 베이스 클래스.

Added in version 3.3.

버전 3.10에서 변경: No longer a subclass of `Finder`.

find_spec(*fullname*, *path*, *target=None*)

지정된 모듈의 스펙을 찾는 추상 메서드. 최상위 임포트 인 경우, *path*는 `None` 입니다. 그렇지 않으면, 이것은 서브 패키지나 모듈의 검색이 되고, *path*는 부모 패키지의 `__path__` 값입니다. 스펙을 찾을 수 없으면, `None`이 반환됩니다. 전달될 때, `target`은 파인더가 반환할 스펙에 대해 더 정교하게 추측하기 위해 사용할 수 있는 모듈 객체입니다. *importlib.util.spec_from_loader()*는 구상 `MetaPathFinders`를 구현하는 데 유용할 수 있습니다.

Added in version 3.4.

invalidate_caches()

호출될 때, 파인더가 사용하는 내부 캐시를 무효로 해야 하는 선택적 메서드. *sys.meta_path*에서 모든 파인더의 캐시를 무효로 할 때 *importlib.invalidate_caches()*에서 사용합니다.

버전 3.4에서 변경: 호출될 때 *NotImplemented* 대신 `None`을 반환합니다.

class importlib.abc.**PathEntryFinder**

경로 엔트리 파인더를 나타내는 추상 베이스 클래스. *MetaPathFinder*와 일부 유사하지만, `PathEntryFinder`는 *importlib.machinery.PathFinder*가 제공하는 경로 기반 임포트 서브 시스템 내에서만 사용하려는 것입니다.

Added in version 3.3.

버전 3.10에서 변경: No longer a subclass of `Finder`.

find_spec(*fullname*, *target=None*)

지정된 모듈의 스펙을 찾는 추상 메서드. 파인더는 할당된 경로 엔트리 내에서만 모듈을 검색합니다. 스펙을 찾을 수 없으면, `None`이 반환됩니다. 전달될 때, `target`은 파인더가 반환할 스펙에 대해 더 정교하게 추측하기 위해 사용할 수 있는 모듈 객체입니다. *importlib.util.spec_from_loader()*는 구상 `PathEntryFinders`를 구현하는 데 유용할 수 있습니다.

Added in version 3.4.

invalidate_caches()

호출될 때, 파인더가 사용하는 내부 캐시를 무효로 해야 하는 선택적 메서드. 모든 캐시 된 파인더의 캐시를 무효화 할 때 *importlib.machinery.PathFinder.invalidate_caches()*에서 사용합니다.

class importlib.abc.**Loader**

로더의 추상 베이스 클래스. 로더에 대한 정확한 정의는 **PEP 302**를 참조하십시오.

리소스 읽기를 지원하려는 로더는 *importlib.resources.abc.ResourceReader*에 지정된 대로 `get_resource_reader()` 메서드를 구현해야 합니다.

버전 3.7에서 변경: 선택적 `get_resource_reader()` 메서드를 도입했습니다.

create_module(*spec*)

모듈을 임포트 할 때 사용할 모듈 객체를 반환하는 메서드. 이 메서드는 `None`을 반환해서 기본 모듈 생성 시맨틱이 적용되어야 함을 나타낼 수 있습니다.

Added in version 3.4.

버전 3.6에서 변경: *exec_module()*이 정의될 때 이 메서드는 더는 선택 사항이 아닙니다.

exec_module(*module*)

모듈을 임포트 하거나 다시 로드할 때 자체 이름 공간에서 모듈을 실행하는 추상 메서드. *exec_module()*이 호출될 때 모듈이 이미 초기화되어 있어야 합니다. 이 메서드가 존재하면, *create_module()*을 정의해야 합니다.

Added in version 3.4.

버전 3.6에서 변경: *create_module()*도 정의해야 합니다.

load_module(*fullname*)

모듈을 로드하는 레거시 메서드. 모듈을 로드할 수 없으면, *ImportError*가 발생하고, 그렇지 않으면 로드된 모듈이 반환됩니다.

요청된 모듈이 *sys.modules*에 이미 존재하면, 해당 모듈이 사용되고 다시 로드되어야 합니다. 그렇지 않으면 로더는 임포트에서 재귀를 방지하기 위해 로드를 시작하기 전에 새 모듈을 만들어 *sys.modules*에 삽입해야 합니다. 로더가 모듈을 삽입했는데 로드에 실패하면, 로더가 *sys.modules*에서 모듈을 제거해야 합니다; 로더가 실행을 시작하기 전에 이미 *sys.modules*에 있었던 모듈은 그대로 두어야 합니다.

로더는 모듈에서 여러 어트리뷰트를 설정해야 합니다 (이러한 어트리뷰트 중 일부는 모듈을 다시 로드할 때 변경될 수 있습니다):

- module.__name__
- module.__file__
- module.__cached__ (폐지됨)
- module.__path__
- module.__package__ (폐지됨)
- module.__loader__ (폐지됨)

exec_module()을 사용할 수 있으면 이전 버전과 호환되는 기능이 제공됩니다.

버전 3.4에서 변경: 호출될 때 NotImplementedError 대신 ImportError를 발생시킵니다. exec_module()을 사용할 수 있을 때 제공되는 기능.

Deprecated since version 3.4, will be removed in version 3.15: 모듈 로드에 권장되는 API는 exec_module()(및 create_module()) 입니다. 로더는 load_module() 대신 이것을 구현해야 합니다. 임포트 절차는 exec_module()이 구현될 때 load_module()의 다른 모든 책임을 처리합니다.

class importlib.abc.**ResourceLoader**

TraversableResources 로 대체되었습니다

스토리지 백 엔드에서 임의의 리소스를 로드하기 위한 선택적 **PEP 302** 프로토콜을 구현하는 로더의 추상 베이스 클래스.

버전 3.7부터 폐지됨: This ABC is deprecated in favour of supporting resource loading through importlib.resources.abc.TraversableResources. This class exists for backwards compatibility only with other ABCs in this module.

abstractmethod get_data(*path*)

*path*에 있는 데이터를 바이트열로 반환하는 추상 메서드. 임의의 데이터를 저장할 수 있는 파일류 스토리지 백 엔드가 있는 로더는 이 추상 메서드를 구현하여 저장된 데이터에 직접 액세스하도록 할 수 있습니다. *path*를 찾을 수 없으면 OSError가 발생합니다. *path*는 모듈의 __file__ 어트리뷰트나 패키지의 __path__에서 온 항목을 사용하여 구성될 것으로 기대됩니다.

버전 3.4에서 변경: NotImplementedError 대신 OSError를 발생시킵니다.

class importlib.abc.**InspectLoader**

모듈을 검사(inspect)하는 로더를 위한 선택적 **PEP 302** 프로토콜을 구현하는 로더의 추상 베이스 클래스.

get_code(*fullname*)

모듈에 대한 코드 객체나, 모듈에 코드 객체가 없으면 (예를 들어, 내장 모듈이 이런 경우입니다) None을 반환합니다. 로더가 요청한 모듈을 찾을 수 없으면 ImportError가 발생합니다.

> **참고**
>
> 이 메서드에는 기본 구현이 있지만, 가능하다면 성능을 위해 재정의하는 것이 좋습니다.

버전 3.4에서 변경: 더는 추상적이지 않고 구상 구현이 제공됩니다.

abstractmethod get_source(*fullname*)

모듈의 소스를 반환하는 추상 메서드. 인식된 모든 줄 구분자를 '\n' 문자로 변환하는 유니버설 줄 넘김을 사용하여 텍스트 문자열로 반환됩니다. 사용 가능한 소스가 없으면 (예를 들어, 내장 모듈) None을 반환합니다. 로더가 지정된 모듈을 찾을 수 없으면 ImportError를 발생시킵니다.

버전 3.4에서 변경: NotImplementedError 대신 ImportError를 발생시킵니다.

is_package(*fullname*)

모듈이 패키지이면 참값을 반환하고, 그렇지 않으면 거짓 값을 반환하는 선택적 메서드. 로더가 모듈을 찾을 수 없으면 `ImportError`가 발생합니다.

버전 3.4에서 변경: `NotImplementedError` 대신 `ImportError`를 발생시킵니다.

static source_to_code(*data*, *path*='<*string*>')

파이썬 소스에서 코드 객체를 만듭니다.

data 인자는 `compile()` 함수가 지원하는 것은 무엇이든 될 수 있습니다 (즉 문자열이나 바이트열). *path* 인자는 소스 코드가 온 곳의 "경로" 여야 하며, 추상 개념(예를 들어 zip 파일에서의 위치) 일 수 있습니다.

후속 코드 객체를 사용하면 `exec(code, module.__dict__)` 를 실행하여 그 코드를 모듈에서 실행할 수 있습니다.

Added in version 3.4.

버전 3.5에서 변경: 메서드를 정적(static)으로 만들었습니다.

exec_module(*module*)

`Loader.exec_module()`의 구현.

Added in version 3.4.

load_module(*fullname*)

`Loader.load_module()`의 구현.

Deprecated since version 3.4, will be removed in version 3.15: 대신 `exec_module()`을 사용하십시오.

class importlib.abc.**ExecutionLoader**

구현될 때, 모듈이 스크립트로 실행되도록 돕는 `InspectLoader`에서 상속되는 추상 베이스 클래스. ABC는 선택적 **PEP 302** 프로토콜을 표현합니다.

abstractmethod get_filename(*fullname*)

지정된 모듈의 `__file__` 값을 반환하는 추상 메서드. 사용 가능한 경로가 없으면 `ImportError`가 발생합니다.

소스 코드를 사용할 수 있으면, 메서드는 모듈을 로드하는 데 바이트 코드를 사용했는지와 관계없이 소스 파일의 경로를 반환해야 합니다.

버전 3.4에서 변경: `NotImplementedError` 대신 `ImportError`를 발생시킵니다.

class importlib.abc.**FileLoader**(*fullname*, *path*)

`ResourceLoader`와 `ExecutionLoader`를 상속하고 `ResourceLoader.get_data()`와 `ExecutionLoader.get_filename()`의 구상 구현을 제공하는 추상 베이스 클래스.

fullname 인자는 로더가 처리해야 하는 모듈의 완전히 결정된(resolved) 이름입니다. *path* 인자는 모듈의 파일 경로입니다.

Added in version 3.3.

name

로더가 처리할 수 있는 모듈의 이름.

path

모듈 파일의 경로.

load_module(*fullname*)

super의 `load_module()` 을 호출합니다.

Deprecated since version 3.4, will be removed in version 3.15: 대신 `Loader.exec_module()`을 사용하십시오.

abstractmethod get_filename(*fullname*)

`path`를 반환합니다.

abstractmethod get_data(*path*)

*path*를 바이너리 파일로 읽고 그것의 바이트열을 반환합니다.

class importlib.abc.**SourceLoader**

소스 (및 선택적으로 바이트 코드) 파일 로드를 구현하기 위한 추상 베이스 클래스. 이 클래스는 `ResourceLoader`와 `ExecutionLoader`를 모두 상속하며, 다음을 구현해야 합니다:

- `ResourceLoader.get_data()`
- ***ExecutionLoader.get_filename()***
  소스 파일의 경로만 반환해야 합니다; 소스 없는 로딩은 지원되지 않습니다.

이 클래스에 의해 정의된 추상 메서드는 선택적 바이트 코드 파일 지원을 추가하는 것입니다. 이러한 선택적 메서드를 구현하지 않으면 (또는 그들이 `NotImplementedError`를 발생시키도록 하면) 로더가 소스 코드에 대해서만 작동하도록 만듭니다. 메서드를 구현하면 로더가 소스*와* 바이트 코드 파일 모두에 대해 작동하게 할 수 있습니다; 바이트 코드만 제공되는 소스 없는 로드는 허용하지 않습니다. 바이트 코드 파일은 파이썬 컴파일러의 구문 분석 단계를 제거하여 로딩 속도를 높이기 위한 최적화라서, 바이트 코드 전용 API는 노출되지 않습니다.

path_stats(*path*)

지정된 경로에 대한 메타 데이터를 포함하는 `dict`를 반환하는 선택적 추상 메서드. 지원되는 딕셔너리 키는 다음과 같습니다:

- `'mtime'` (필수): 소스 코드의 수정 시간을 나타내는 정수나 부동 소수점 숫자;
- `'size'` (선택): 바이트 단위의 소스 코드의 크기.

향후 확장을 위해, 딕셔너리의 다른 키는 무시됩니다. 경로를 처리할 수 없으면, `OSError`가 발생합니다.

Added in version 3.3.

버전 3.4에서 변경: `NotImplementedError` 대신 `OSError`를 발생시킵니다.

path_mtime(*path*)

지정된 경로의 수정 시간을 반환하는 선택적 추상 메서드.

버전 3.3부터 폐지됨: 이 메서드는 폐지되었고 `path_stats()`로 대체되었습니다. 구현할 필요는 없지만, 호환성을 위해 여전히 제공됩니다. 경로를 처리할 수 없으면 `OSError`를 발생시킵니다.

버전 3.4에서 변경: `NotImplementedError` 대신 `OSError`를 발생시킵니다.

set_data(*path*, *data*)

지정된 바이트열을 파일 경로에 쓰는 선택적 추상 메서드. 존재하지 않는 중간 디렉터리는 자동으로 만들어집니다.

경로가 읽기 전용(`errno.EACCES`/`PermissionError`)이라서 경로에 쓰지 못할 때 예외를 전파하지 않습니다.

버전 3.4에서 변경: 호출할 때 더는 `NotImplementedError`를 발생시키지 않습니다.

get_code(*fullname*)

`InspectLoader.get_code()`의 구상 구현.

exec_module(*module*)

`Loader.exec_module()`의 구상 구현.

Added in version 3.4.

load_module(*fullname*)

`Loader.load_module()`의 구상 구현.

Deprecated since version 3.4, will be removed in version 3.15: 대신 `exec_module()`을 사용하십시오.

get_source(*fullname*)

`InspectLoader.get_source()`의 구상 구현.

is_package(*fullname*)

`InspectLoader.is_package()`의 구상 구현. (`ExecutionLoader.get_filename()`에서 제공되는) 파일 경로가 파일 확장자를 제거했을 때 `__init__`라는 이름의 파일이고 동시에 모듈 이름 자체가 `__init__`로 끝나지 않으면 모듈은 패키지로 결정됩니다.

class importlib.abc.**ResourceReader**

TraversableResources 로 대체되었습니다

리소스(*resources*)를 읽을 수 있는 기능을 제공하는 추상 베이스 클래스.

이 ABC의 관점에서, 리소스(*resource*)는 패키지 내에 제공되는 바이너리 아티팩트(artifact)입니다. 일반적으로 이것은 패키지의 `__init__.py` 파일 옆에 있는 데이터 파일 같은 것입니다. 이 클래스의 목적은 이러한 데이터 파일에 대한 액세스를 추상화하여 패키지와 해당 데이터 파일이 예를 들어 zip 파일에 있는지 파일 시스템에 저장되어 있는지가 중요하지 않도록 만드는 것입니다.

이 클래스의 모든 메서드에서, *resource* 인자는 개념적으로 단지 파일 이름을 나타내는 경로류 객체가 될 것으로 기대됩니다. 이는 *resource* 인자에 서브 디렉터리 경로가 포함되지 않아야 함을 의미합니다. 판독기(reader)가 읽으려는 패키지의 위치가 "디렉터리"의 역할을 하기 때문입니다. 따라서 디렉터리와 파일 이름에 대한 은유는 각각 패키지와 리소스입니다. 이것은 또한 이 클래스의 인스턴스가 (잠재적으로 여러 패키지나 모듈을 나타내는 대신) 특정 패키지와 직접적으로 연관될 것으로 기대되는 이유입니다.

리소스 읽기를 지원하려는 로더는 이 ABC의 인터페이스를 구현하는 객체를 반환하는 `get_resource_reader(fullname)`이라는 메서드를 제공해야 합니다. fullname으로 지정된 모듈이 패키지가 아니면, 이 메서드는 `None`을 반환해야 합니다. 이 ABC와 호환되는 객체는 지정된 모듈이 패키지일 때만 반환해야 합니다.

Added in version 3.7.

Deprecated since version 3.12, removed in version 3.14: 대신 `importlib.resources.abc.TraversableResources`을 사용하십시오.

abstractmethod open_resource(*resource*)

*resource*의 바이너리 읽기를 위해 열린 파일류 객체를 반환합니다.

리소스를 찾을 수 없으면, `FileNotFoundError` 가 발생합니다.

abstractmethod resource_path(*resource*)

*resource*에 대한 파일 시스템 경로를 반환합니다.

리소스가 파일 시스템에 구체적으로 존재하지 않으면, `FileNotFoundError` 가 발생합니다.

abstractmethod is_resource(*name*)

명명된 *name*을 리소스로 간주하면 `True`를 반환합니다. *name*이 없으면, `FileNotFoundError` 가 발생합니다.

abstractmethod contents()

패키지 내용에 대한 문자열의 이터러블을 반환합니다. 이터레이터가 반환한 모든 이름이 실제 리소스일 필요는 없음에 유의하십시오, 예를 들어 `is_resource()`가 거짓인 이름을 반환하는 것이 허용됩니다.

리소스가 아닌 이름이 반환되도록 하는 것은 패키지와 그것의 리소스가 저장되는 방법이 사전에 알려졌고 리소스가 아닌 이름이 유용한 상황을 허용하기 위함입니다. 예를

들어, 패키지와 리소스가 파일 시스템에 저장되어있는 것으로 알려졌을 때 해당 서브 디렉터리 이름을 직접 사용할 수 있도록 서브 디렉터리 이름 반환이 허용됩니다.

추상 메서드는 항목이 없는 이터러블을 반환합니다.

class importlib.abc.**Traversable**

디렉터리를 탐색하고 파일을 여는 데 적합한 *pathlib.Path* 메서드의 부분집합이 있는 객체.

For a representation of the object on the file-system, use *importlib.resources.as_file()*.

Added in version 3.9.

Deprecated since version 3.12, removed in version 3.14: 대신 *importlib.resources.abc.Traversable*을 사용하십시오.

name

Abstract. The base name of this object without any parent references.

abstractmethod iterdir()

Yield `Traversable` objects in `self`.

abstractmethod is_dir()

Return `True` if `self` is a directory.

abstractmethod is_file()

Return `True` if `self` is a file.

abstractmethod joinpath(*child*)

Return Traversable child in `self`.

abstractmethod __truediv__(*child*)

Return `Traversable` child in `self`.

abstractmethod open(*mode='r', *args, **kwargs*)

mode may be 'r' or 'rb' to open as text or binary. Return a handle suitable for reading (same as *pathlib.Path.open*).

When opening as text, accepts encoding parameters such as those accepted by *io.TextIOWrapper*.

read_bytes()

Read contents of `self` as bytes.

read_text(*encoding=None*)

Read contents of `self` as text.

class importlib.abc.**TraversableResources**

importlib.resources.files() 인터페이스를 제공할 수 있는 리소스 리더를 위한 추상 베이스 클래스. *importlib.resources.abc.ResourceReader*를 서브 클래싱하고 *importlib.resources.abc.ResourceReader*의 추상 메서드의 구상 구현을 제공합니다. 따라서, *importlib.abc.TraversableResources*를 제공하는 모든 로더는 ResourceReader도 제공합니다.

Loaders that wish to support resource reading are expected to implement this interface.

Added in version 3.9.

Deprecated since version 3.12, removed in version 3.14: 대신 *importlib.resources.abc.TraversableResources*을 사용하십시오.

abstractmethod files()

Returns a *importlib.resources.abc.Traversable* object for the loaded package.

31.5.4 importlib.machinery – 임포터와 경로 훅

소스 코드: Lib/importlib/machinery.py

이 모듈에는 import 가 모듈을 찾고 로드하는 데 도움이 되는 다양한 객체가 포함되어 있습니다.

importlib.machinery.**SOURCE_SUFFIXES**

소스 모듈로 인식되는 파일 접미사를 나타내는 문자열 리스트.

Added in version 3.3.

importlib.machinery.**DEBUG_BYTECODE_SUFFIXES**

최적화되지 않은 바이트 코드 모듈의 파일 접미사를 나타내는 문자열 리스트.

Added in version 3.3.

버전 3.5부터 폐지됨: 대신 *BYTECODE_SUFFIXES*를 사용하십시오.

importlib.machinery.**OPTIMIZED_BYTECODE_SUFFIXES**

최적화된 바이트 코드 모듈의 파일 접미사를 나타내는 문자열 리스트.

Added in version 3.3.

버전 3.5부터 폐지됨: 대신 *BYTECODE_SUFFIXES*를 사용하십시오.

importlib.machinery.**BYTECODE_SUFFIXES**

바이트 코드 모듈로 인식되는 파일 접미사를 나타내는 문자열 리스트 (앞의 점을 포함합니다).

Added in version 3.3.

버전 3.5에서 변경: 이 값은 더는 __debug__에 의존하지 않습니다.

importlib.machinery.**EXTENSION_SUFFIXES**

확장 모듈로 인식되는 파일 접미사를 나타내는 문자열 리스트.

Added in version 3.3.

importlib.machinery.**all_suffixes**()

표준 임포트 절차가 인식하는 모듈의 모든 파일 접미사를 나타내는 문자열의 결합한 리스트를 반환합니다. 이것은 모듈 종류에 대한 세부 정보 없이 파일 시스템 경로가 잠재적으로 모듈을 참조하는지를 알아야 하는 코드(예를 들어, *inspect.getmodulename()*)를 위한 도우미입니다.

Added in version 3.3.

class importlib.machinery.**BuiltinImporter**

내장 모듈용 임포터. 알려진 모든 내장 모듈은 *sys.builtin_module_names*에 나열되어 있습니다. 이 클래스는 *importlib.abc.MetaPathFinder*와 *importlib.abc.InspectLoader* ABC를 구현합니다.

이 클래스는 인스턴스 화의 필요성을 완화하기 위해 클래스 메서드만 정의합니다.

버전 3.5에서 변경: **PEP 489**의 일부로, 내장 임포터는 이제 Loader.create_module() 과 Loader.exec_module() 을 구현합니다.

class importlib.machinery.**FrozenImporter**

프로즌(frozen) 모듈용 임포터. 이 클래스는 *importlib.abc.MetaPathFinder*와 *importlib.abc.InspectLoader* ABC를 구현합니다.

이 클래스는 인스턴스 화의 필요성을 완화하기 위해 클래스 메서드만 정의합니다.

버전 3.4에서 변경: Loader.create_module() 과 Loader.exec_module() 메서드를 얻었습니다.

class importlib.machinery.**WindowsRegistryFinder**

윈도우 레지스트리에 선언된 모듈용 파인더. 이 클래스는 *importlib.abc.MetaPathFinder* ABC를 구현합니다.

이 클래스는 인스턴스 화의 필요성을 완화하기 위해 클래스 메서드만 정의합니다.

Added in version 3.3.

버전 3.6부터 폐지됨: 대신 *site* 구성을 사용하십시오. 이후 버전의 파이썬은 기본적으로 이 파인더를 활성화하지 않을 수 있습니다.

class importlib.machinery.**PathFinder**

*sys.path*와 패키지 `__path__` 어트리뷰트용 파인더. 이 클래스는 *importlib.abc.MetaPathFinder* ABC를 구현합니다.

이 클래스는 인스턴스 화의 필요성을 완화하기 위해 클래스 메서드만 정의합니다.

classmethod find_spec(*fullname*, *path=None*, *target=None*)

sys.path 또는, 정의되었다면, *path*에서 *fullname*에 의해 지정된 모듈에 대한 스펙을 찾으려고 시도하는 클래스 메서드. 검색된 각 경로 엔트리에 대해, *sys.path_importer_cache*가 확인됩니다. 거짓이 아닌 객체를 찾으면 검색 중인 모듈을 찾기 위한 경로 엔트리 파인더로 사용됩니다. *sys.path_importer_cache*에 엔트리가 없으면, *sys.path_hooks*에서 경로 엔트리를 위한 파인더를 검색하고, 발견되면, 모듈에 대해 조회되는 것과 동시에 *sys.path_importer_cache*에 저장됩니다. 파인더가 아예 발견되지 않으면 `None`이 캐시에 저장되고 반환됩니다.

Added in version 3.4.

버전 3.5에서 변경: 현재 작업 디렉터리 – 빈 문자열로 표현됩니다 – 가 더는 유효하지 않으면 `None`이 반환되지만 *sys.path_importer_cache*에 값이 캐시 되지는 않습니다.

classmethod invalidate_caches()

메서드를 정의하는 *sys.path_importer_cache*에 저장된 모든 파인더에 대해 *importlib.abc.PathEntryFinder.invalidate_caches()*를 호출합니다. *sys.path_importer_cache*에 `None`으로 설정된 엔트리가 삭제됩니다.

버전 3.7에서 변경: *sys.path_importer_cache*에서 `None`의 엔트리가 삭제됩니다.

버전 3.4에서 변경: `''`(즉 빈 문자열)에 대해서는 현재 작업 디렉터리로 *sys.path_hooks*의 객체를 호출합니다.

class importlib.machinery.**FileFinder**(*path*, **loader_details*)

파일 시스템에서의 결과를 캐시 하는 *importlib.abc.PathEntryFinder*의 구상 구현.

path 인자는 파인더가 검색을 담당하는 디렉터리입니다.

loader_details 인자는 각각 로더와 로더가 인식하는 파일 접미사의 시퀀스를 포함하는 가변 개수의 2개 항목 튜플입니다. 로더는 모듈 이름과 찾은 파일의 경로로 구성되는 두 인자를 받아들이는 콜러블일 것으로 기대됩니다.

파인더는 필요에 따라 디렉터리 내용을 캐시 하여, 각 모듈 검색에서 stat 호출을 수행하여 캐시가 시효가 지나지 않았는지 확인합니다. 캐시 만료는 파일 시스템의 운영 체제 상태 정보의 세분성에 의존하기 때문에, 모듈 검색, 새 파일 생성 및 새 파일이 나타내는 모듈 검색의 잠재적 경쟁 조건이 있습니다. stat 호출의 세분성 이하로 연산이 아주 빠르게 수행되면, 모듈 검색이 실패합니다. 이를 방지하려면, 모듈을 동적으로 만들 때, *importlib.invalidate_caches()*를 호출해야 합니다.

Added in version 3.3.

path

파인더가 검색할 경로.

find_spec(*fullname*, *target=None*)

path 내에서 *fullname*을 처리할 스펙을 찾으려고 합니다.

Added in version 3.4.

invalidate_caches()

내부 캐시를 지웁니다.

classmethod path_hook(**loader_details*)

*sys.path_hooks*에 사용할 클로저를 반환하는 클래스 메서드. *FileFinder*의 인스턴스는 클로저에 직접 제공된 경로 인자와 *loader_details*를 간접적으로 사용하여 클로저에 의해 반환됩니다.

클로저에 대한 인자가 기존 디렉터리가 아니면, *ImportError*가 발생합니다.

class importlib.machinery.**SourceFileLoader**(*fullname*, *path*)

*importlib.abc.FileLoader*를 서브 클래싱하고 다른 메서드의 구상 구현을 제공하는 *importlib.abc.SourceLoader*의 구상 구현.

Added in version 3.3.

name

이 로더가 처리할 모듈의 이름.

path

소스 파일의 경로.

is_package(*fullname*)

*path*가 패키지에 대한 것으로 드러나면 `True`를 반환합니다.

path_stats(*path*)

*importlib.abc.SourceLoader.path_stats()*의 구상 구현.

set_data(*path*, *data*)

*importlib.abc.SourceLoader.set_data()*의 구상 구현.

load_module(*name=None*)

로드할 모듈 이름을 지정하는 것이 선택적인 *importlib.abc.Loader.load_module()*의 구상 구현.

Deprecated since version 3.6, will be removed in version 3.15: 대신 *importlib.abc.Loader.exec_module()*을 사용하십시오.

class importlib.machinery.**SourcelessFileLoader**(*fullname*, *path*)

바이트 코드 파일을 (즉, 소스 코드 파일 없이) 임포트 할 수 있는 *importlib.abc.FileLoader*의 구상 구현.

바이트 코드 파일(그래서 소스 코드 파일이 아닌)을 직접 사용하면 모든 파이썬 구현이나 바이트 코드 형식을 변경하는 새 버전의 파이썬에서 모듈을 사용할 수 없게 됨에 유의하십시오.

Added in version 3.3.

name

로더가 처리할 모듈의 이름.

path

바이트 코드 파일의 경로.

is_package(*fullname*)

*path*를 기반으로 모듈이 패키지인지 판단합니다.

get_code(*fullname*)

*path*에서 만들어진 *name*의 코드 객체를 반환합니다.

get_source(*fullname*)

이 로더가 사용될 때는 바이트 코드 파일에 소스가 없어서 `None`을 반환합니다.

load_module(*name=None*)

로드할 모듈 이름을 지정하는 것이 선택적인 `importlib.abc.Loader.load_module()`의 구상 구현.

Deprecated since version 3.6, will be removed in version 3.15: 대신 `importlib.abc.Loader.exec_module()`을 사용하십시오.

class importlib.machinery.**ExtensionFileLoader**(*fullname, path*)

확장 모듈을 위한 `importlib.abc.ExecutionLoader`의 구상 구현.

fullname 인자는 로더가 지원할 모듈의 이름을 지정합니다. *path* 인자는 확장 모듈 파일의 경로입니다.

Note that, by default, importing an extension module will fail in subinterpreters if it doesn' t implement multi-phase init (see **PEP 489**), even if it would otherwise import successfully.

Added in version 3.3.

버전 3.12에서 변경: Multi-phase init is now required for use in subinterpreters.

name

로더가 지원하는 모듈의 이름.

path

확장 모듈의 경로.

create_module(*spec*)

PEP 489에 따라 지정된 명세에서 모듈 객체를 만듭니다.

Added in version 3.5.

exec_module(*module*)

PEP 489에 따라 주어진 모듈 객체를 초기화합니다.

Added in version 3.5.

is_package(*fullname*)

`EXTENSION_SUFFIXES`에 기반해서 파일 경로가 패키지의 `__init__` 모듈을 가리키면 `True`를 반환합니다.

get_code(*fullname*)

확장 모듈에는 코드 객체가 없어서 `None`을 반환합니다.

get_source(*fullname*)

확장 모듈에는 소스 코드가 없어서 `None`을 반환합니다.

get_filename(*fullname*)

`path`를 반환합니다.

Added in version 3.4.

class importlib.machinery.**NamespaceLoader**(*name, path, path_finder*)

A concrete implementation of `importlib.abc.InspectLoader` for namespace packages. This is an alias for a private class and is only made public for introspecting the `__loader__` attribute on namespace packages:

```
>>> from importlib.machinery import NamespaceLoader
>>> import my_namespace
>>> isinstance(my_namespace.__loader__, NamespaceLoader)
True
>>> import importlib.abc
>>> isinstance(my_namespace.__loader__, importlib.abc.Loader)
True
```

Added in version 3.11.

class importlib.machinery.**ModuleSpec**(*name, loader, *, origin=None, loader_state=None, is_package=None*)

모듈의 임포트 시스템 관련 상태에 대한 명세. 이것은 일반적으로 모듈의 `__spec__` 어트리뷰트로 노출됩니다. 이 중 많은 어트리뷰트는 모듈에서도 직접 제공됩니다: 예를 들어, `module.__spec__.origin == module.__file__`. 그러나 *values*는 일반적으로 동등하지만, 두 객체 간에 동기화가 없기 때문에 다를 수 있음에 유의하십시오. 예를 들어, 실행 시간에 모듈의 `__file__`을 갱신할 수 있으며, 이는 모듈의 *`__spec__.origin`*에 자동으로 반영되지 않고, 그 반대도 마찬가지 입니다.

Added in version 3.4.

name

The module' s fully qualified name (see `module.__name__`). The *finder* should always set this attribute to a non-empty string.

loader

모듈을 로드하는 데 사용되는 로더 (`module.__loader__`를 참조하세요). 파인더는 항상 이 어트리뷰트를 설정해야 합니다.

origin

The location the *loader* should use to load the module (see `module.__file__`). For example, for modules loaded from a `.py` file this is the filename. The *finder* should always set this attribute to a meaningful value for the *loader* to use. In the uncommon case that there is not one (like for namespace packages), it should be set to `None`.

submodule_search_locations

A (possibly empty) *sequence* of strings enumerating the locations in which a package' s submodules will be found (see `module.__path__`). Most of the time there will only be a single directory in this list.

The *finder* should set this attribute to a sequence, even an empty one, to indicate to the import system that the module is a package. It should be set to `None` for non-package modules. It is set automatically later to a special object for namespace packages.

loader_state

The *finder* may set this attribute to an object containing additional, module-specific data to use when loading the module. Otherwise it should be set to `None`.

cached

The filename of a compiled version of the module' s code (see `module.__cached__`). The *finder* should always set this attribute but it may be `None` for modules that do not need compiled code stored.

parent

(읽기 전용) 모듈이 속한 패키지의 패키지의 완전히 정규화된 이름 (또는 최상위 수준 모듈의 경우 빈 문자열). `module.__package__`를 참조하세요. 모듈이 패키지면, *`name`*과 같습니다.

has_location

`True` if the spec' s *`origin`* refers to a loadable location, `False` otherwise. This value impacts how `origin` is interpreted and how the module' s `__file__` is populated.

class importlib.machinery.**AppleFrameworkLoader**(*name, path*)

프레임워크 형식의 확장 모듈을 로드할 수 있는 *`importlib.machinery.ExtensionFileLoader`*의 특수화.

For compatibility with the iOS App Store, *all* binary modules in an iOS app must be dynamic libraries, contained in a framework with appropriate metadata, stored in the `Frameworks` folder of the packaged app. There can be only a single binary per framework, and there can be no executable binary material outside the Frameworks folder.

To accommodate this requirement, when running on iOS, extension module binaries are *not* packaged as `.so` files on `sys.path`, but as individual standalone frameworks. To discover those frameworks, this loader is be registered against the `.fwork` file extension, with a `.fwork` file acting as a placeholder in the original location of the binary on `sys.path`. The `.fwork` file contains the path of the actual binary in the `Frameworks` folder, relative to the app bundle. To allow for resolving a framework-packaged binary back to the original location,

the framework is expected to contain a `.origin` file that contains the location of the `.fwork` file, relative to the app bundle.

For example, consider the case of an import `from foo.bar import _whiz`, where `_whiz` is implemented with the binary module `sources/foo/bar/_whiz.abi3.so`, with `sources` being the location registered on `sys.path`, relative to the application bundle. This module *must* be distributed as `Frameworks/foo.bar._whiz.framework/foo.bar._whiz` (creating the framework name from the full import path of the module), with an `Info.plist` file in the `.framework` directory identifying the binary as a framework. The `foo.bar._whiz` module would be represented in the original location with a `sources/foo/bar/_whiz.abi3.fwork` marker file, containing the path `Frameworks/foo.bar._whiz/foo.bar._whiz`. The framework would also contain `Frameworks/foo.bar._whiz.framework/foo.bar._whiz.origin`, containing the path to the `.fwork` file.

When a module is loaded with this loader, the `__file__` for the module will report as the location of the `.fwork` file. This allows code to use the `__file__` of a module as an anchor for file system traversal. However, the spec origin will reference the location of the *actual* binary in the `.framework` folder.

The Xcode project building the app is responsible for converting any `.so` files from wherever they exist in the `PYTHONPATH` into frameworks in the `Frameworks` folder (including stripping extensions from the module file, the addition of framework metadata, and signing the resulting framework), and creating the `.fwork` and `.origin` files. This will usually be done with a build step in the Xcode project; see the iOS documentation for details on how to construct this build step.

Added in version 3.13.

가용성: iOS.

name

로더가 지원하는 모듈의 이름.

path

확장 모듈의 `.fwork` 파일 경로.

31.5.5 `importlib.util` – 임포터를 위한 유틸리티 코드

소스 코드: Lib/importlib/util.py

이 모듈에는 임포터 구성에 도움이 되는 다양한 객체가 포함되어 있습니다.

`importlib.util.`**`MAGIC_NUMBER`**

바이트 코드 버전 번호를 나타내는 바이트열. 바이트 코드의 로드/쓰기에 도움이 필요하면 *`importlib.abc.SourceLoader`*를 고려하십시오.

Added in version 3.4.

`importlib.util.`**`cache_from_source`**(*path*, *debug_override=None*, ***, *optimization=None*)

소스 *path*와 연관된 바이트 컴파일된 파일의 **PEP 3147/PEP 488** 경로를 반환합니다. 예를 들어, *path*가 `/foo/bar/baz.py`이면 반환값은 파이썬 3.2의 경우 `/foo/bar/__pycache__/baz.cpython-32.pyc`입니다. `cpython-32` 문자열은 현재 매직 태그에서 온 것입니다 (`get_tag()`를 참조하십시오; `sys.implementation.cache_tag`가 정의되지 않으면 *`NotImplementedError`*가 발생합니다).

optimization 매개 변수는 바이트 코드 파일의 최적화 수준을 지정하는 데 사용됩니다. 빈 문자열은 최적화하지 않음을 나타내므로, *optimization*이 `''` 인 `/foo/bar/baz.py`는 바이트 코드 경로가 `/foo/bar/__pycache__/baz.cpython-32.pyc`가 됩니다. `None`은 인터프리터의 최적화 수준이 사용되도록 합니다. 다른 값의 문자열 표현은 사용되므로, *optimization*이 `2`인 `/foo/bar/baz.py`는 바이트 코드 경로가 `/foo/bar/__pycache__/baz.cpython-32.opt-2.pyc`가 됩니다. *optimization*의 문자열 표현은 영숫자만 가능하며, 그렇지 않으면 *`ValueError`*가 발생합니다.

debug_override 매개 변수는 폐지되었고 `__debug__`의 시스템값을 대체하는 데 사용할 수 있습니다. `True` 값은 *optimization*을 빈 문자열로 설정하는 것과 등등합니다. `False` 값은 *optimization*을 `1`로 설정하는 것과 같습니다. *debug_override*와 *optimization*이 모두 `None`이 아니면 *`TypeError`*가 발생합니다.

Added in version 3.4.

버전 3.5에서 변경: *optimization* 매개 변수가 추가되었고 *debug_override* 매개 변수는 폐지되었습니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

importlib.util.**source_from_cache**(*path*)

*path*에 **PEP 3147** 파일 이름이 주어지면, 연관된 소스 코드 파일 경로를 반환합니다. 예를 들어, *path*가 `/foo/bar/__pycache__/baz.cpython-32.pyc`이면 반환된 경로는 `/foo/bar/baz.py`입니다. *path*는 존재할 필요는 없지만, **PEP 3147**이나 **PEP 488** 형식을 준수하지 않으면, `ValueError`가 발생합니다. `sys.implementation.cache_tag`가 정의되지 않으면, `NotImplementedError`가 발생합니다.

Added in version 3.4.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

importlib.util.**decode_source**(*source_bytes*)

소스 코드를 나타내는 주어진 바이트열을 디코딩하고 유니버설 줄 넘김이 적용된 문자열로 반환합니다 (`importlib.abc.InspectLoader.get_source()`에 필요한 대로).

Added in version 3.4.

importlib.util.**resolve_name**(*name*, *package*)

상대 모듈 이름을 절대 이름으로 결정합니다.

name 선두에 점이 없으면, **name**이 단순히 반환됩니다. 이를 통해 **package** 인자가 필요한지 확인하지 않고 `importlib.util.resolve_name('sys', __spec__.parent)`와 같은 사용이 가능합니다.

name이 상대 모듈 이름이지만 **package**가 거짓 값(예를 들어 `None`이나 빈 문자열)이면 `ImportError`가 발생합니다. `ImportError`는 또한 상대 이름이 그것을 포함하는 패키지를 벗어날 것 같으면 발생합니다 (예를 들어 `spam` 패키지 내에서 `..bacon`을 요청하는 것).

Added in version 3.3.

버전 3.9에서 변경: import 문과의 일관성을 개선하기 위해, 잘못된 상대 임포트 시도에 대해 `ValueError` 대신 `ImportError`를 발생시킵니다.

importlib.util.**find_spec**(*name*, *package=None*)

선택적으로 지정된 **package** 이름에 상대적으로, 모듈의 스펙을 찾습니다. 모듈이 `sys.modules`에 있으면, `sys.modules[name].__spec__`이 반환됩니다 (스펙이 `None`이 되거나 설정되지 않지 않은 한, 그럴 경우는 `ValueError`가 발생합니다). 그렇지 않으면 `sys.meta_path`를 사용한 검색이 수행됩니다. 스펙을 찾지 못하면 `None`이 반환됩니다.

name이 서브 모듈에 관한 것이면 (점을 포함하면), 부모 모듈은 자동으로 임포트 됩니다.

name과 **package**는 `import_module()`과 같게 작동합니다.

Added in version 3.4.

버전 3.7에서 변경: **package**가 실제로 패키지가 아니면 (즉 `__path__` 어트리뷰트가 없으면) `AttributeError` 대신 `ModuleNotFoundError`를 발생시킵니다.

importlib.util.**module_from_spec**(*spec*)

spec과 `spec.loader.create_module`을 기반으로 새 모듈을 만듭니다.

`spec.loader.create_module`이 `None`을 반환하지 않으면, 어떤 기존 어트리뷰트도 재설정되지 않습니다. 또한 **spec**에 액세스하거나 모듈에서 어트리뷰트를 설정하는 동안 트리거 되면 `AttributeError`가 발생하지 않습니다.

spec은 모듈에서 가능한 많은 임포트 제어 어트리뷰트를 설정하는 데 사용되므로 새 모듈을 작성하는 데 `types.ModuleType`을 사용하는 것보다 이 함수가 선호됩니다.

Added in version 3.5.

importlib.util.**spec_from_loader**(*name*, *loader*, **, origin=None*, *is_package=None*)

로더(loader)를 기반으로 `ModuleSpec` 인스턴스를 만들기 위한 팩토리 함수. 매개 변수는 ModuleSpec에서와 같은 의미입니다. 이 함수는 `InspectLoader.is_package()`와 같은 사용 가능한 로더 API를 사용하여 스펙에 빠진 정보를 채웁니다.

Added in version 3.4.

importlib.util.**spec_from_file_location**(*name, location, *, loader=None, submodule_search_locations=None*)

파일 경로를 기반으로 *ModuleSpec* 인스턴스를 만드는 팩토리 함수. 로더 API를 사용하고 모듈이 파일 기반일 것이라는 것이 뜻하는 것으로 누락된 정보가 스펙에 채워집니다.

Added in version 3.4.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

importlib.util.**source_hash**(*source_bytes*)

*source_bytes*의 해시를 바이트열로 반환합니다. 해시 기반 .pyc 파일은 해당 소스 파일 내용의 *source_hash()*를 헤더에 포함합니다.

Added in version 3.7.

importlib.util.**_incompatible_extension_module_restrictions**(**, disable_check*)

A context manager that can temporarily skip the compatibility check for extension modules. By default the check is enabled and will fail when a single-phase init module is imported in a subinterpreter. It will also fail for a multi-phase init module that doesn' t explicitly support a per-interpreter GIL, when imported in an interpreter with its own GIL.

Note that this function is meant to accommodate an unusual case; one which is likely to eventually go away. There' s is a pretty good chance this is not what you were looking for.

You can get the same effect as this function by implementing the basic interface of multi-phase init (**PEP 489**) and lying about support for multiple interpreters (or per-interpreter GIL).

> **경고**
>
> Using this function to disable the check can lead to unexpected behavior and even crashes. It should only be used during extension module development.

Added in version 3.12.

class importlib.util.**LazyLoader**(*loader*)

모듈이 어트리뷰트에 액세스할 때까지 모듈 로더의 실행을 연기하는 클래스.

이 클래스는 필요한 모듈 형에 대한 제어로 *exec_module()*을 정의하는 로더**에서만** 작동합니다. 같은 이유로, 로더의 *create_module()* 메서드는 None을 반환하거나, 슬롯을 사용하지 않고 __class__ 어트리뷰트가 변경될 수 있는 형을 반환해야 합니다. 마지막으로, *sys.modules*에 배치된 객체를 치환하는 모듈은 인터프리터 전체에서 모듈 참조를 안전하게 대체할 방법이 없어서 작동하지 않습니다; 이러한 치환이 감지되면 *ValueError*가 발생합니다.

> **참고**
>
> 시작 시간이 중요한 프로젝트의 경우, 이 클래스를 사용하면 사용하지 않을 모듈을 로드하는 데 드는 비용을 최소화할 수 있습니다. 시작 시간이 핵심이 아닌 프로젝트의 경우 로딩이 지연되는 동안 만들어진, 따라서 문맥을 벗어난 에러 메시지 때문에, 이 클래스를 사용하지 말 것을 **강하게** 권고합니다.

Added in version 3.5.

버전 3.6에서 변경: *importlib.machinery.BuiltinImporter*와 *importlib.machinery.ExtensionFileLoader*에 대한 호환성 경고를 제거하고, *create_module()*을 호출하기 시작했습니다.

classmethod factory(*loader*)

지연된 로더(lazy loader)를 만드는 콜러블을 반환하는 클래스 메서드. 이것은 로더가 인스턴스가 아닌 클래스로 전달되는 상황에서 사용하려는 것입니다.

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path, (lazy_loader, suffixes))
```

31.5.6 예

프로그래밍 방식으로 임포트 하기

프로그래밍 방식으로 모듈을 임포트 하려면, *importlib.import_module()*을 사용하십시오.

```
import importlib

itertools = importlib.import_module('itertools')
```

모듈을 임포트 할 수 있는지 확인하기

실제로 임포트를 수행하지 않고 모듈을 임포트 할 수 있는지 확인해야 하면, *importlib.util.find_spec()*을 사용해야 합니다.

Note that if `name` is a submodule (contains a dot), *importlib.util.find_spec()* will import the parent module.

```
import importlib.util
import sys

# 설명을 위해.
name = 'itertools'

if name in sys.modules:
    print(f"{name!r} already in sys.modules")
elif (spec := importlib.util.find_spec(name)) is not None:
    # 실제 임포트를 수행하기로 선택하면 ...
    module = importlib.util.module_from_spec(spec)
    sys.modules[name] = module
    spec.loader.exec_module(module)
    print(f"{name!r} has been imported")
else:
    print(f"can't find the {name!r} module")
```

소스 파일을 직접 임포트 하기

This recipe should be used with caution: it is an approximation of an import statement where the file path is specified directly, rather than *sys.path* being searched. Alternatives should first be considered first, such as modifying *sys.path* when a proper module is required, or using *runpy.run_path()* when the global namespace resulting from running a Python file is appropriate.

파이썬 소스 파일을 경로에서 직접 임포트 하려면, 다음 조리법을 사용하십시오:

```
import importlib.util
import sys


def import_from_path(module_name, file_path):
    spec = importlib.util.spec_from_file_location(module_name, file_path)
    module = importlib.util.module_from_spec(spec)
    sys.modules[module_name] = module
    spec.loader.exec_module(module)
    return module
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# 오직 설명을 위해 (`json` 의 사용은 임의적입니다).
import json
file_path = json.__file__
module_name = json.__name__

# `import json` 과 유사한 결과.
json = import_from_path(module_name, file_path)
```

Implementing lazy imports

The example below shows how to implement lazy imports:

```
>>> import importlib.util
>>> import sys
>>> def lazy_import(name):
...     spec = importlib.util.find_spec(name)
...     loader = importlib.util.LazyLoader(spec.loader)
...     spec.loader = loader
...     module = importlib.util.module_from_spec(spec)
...     sys.modules[name] = module
...     loader.exec_module(module)
...     return module
...
>>> lazy_typing = lazy_import("typing")
>>> #lazy_typing 은 실제 모듈 객체입니다,
>>> #하지만 아직 메모리에 로드되지 않았습니다.
>>> lazy_typing.TYPE_CHECKING
False
```

임포터 설정하기

임포트의 심층 사용자 정의를 위해서는, 일반적으로 임포터를 구현하려고 합니다. 이는 파인더와 로더 측면을 모두 관리한다는 의미입니다. 파인더에는 필요에 따라 두 가지 종류가 있습니다: 메타 경로 파인더나 경로 엔트리 파인더. 전자는 *sys.meta_path*에 배치하는 것이고 후자는 *sys.path_hooks*에서 경로 엔트리 훅을 사용하여 만드는 것으로 *sys.path* 항목과 함께 작동하여 파인더를 만듭니다. 이 예제는 임포트가 임포터를 사용할 수 있도록 임포터를 등록하는 방법을 보여줍니다 (임포터를 직접 만들려면, 이 패키지에 정의된 적절한 클래스의 설명서를 읽으십시오):

```
import importlib.machinery
import sys

# 오직 설명을 위해.
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                  importlib.machinery.SOURCE_SUFFIXES)

# 메타 경로 파인더 설정하기.
# 파인더를 우선순위 측면에서 리스트의 올바른 위치에
# 넣으십시오.
sys.meta_path.append(SpamMetaPathFinder)

# 경로 엔트리 파인더 설정하기.
# 경로 훅을 우선순위 측면에서 리스트의 올바른 위치에
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
# 넣으십시오.
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_details))
```

importlib.import_module() 근사하기

임포트 자체는 파이썬 코드로 구현되므로, importlib를 통해 대부분의 임포트 절차를 노출할 수 있습니다. 다음은 *importlib.import_module()*의 근사적인(approximate) 구현을 제공하여 importlib가 노출하는 다양한 API를 설명하는 데 도움을 줍니다:

```
import importlib.util
import sys

def import_module(name, package=None):
    """import의 대략적인 구현."""
    absolute_name = importlib.util.resolve_name(name, package)
    try:
        return sys.modules[absolute_name]
    except KeyError:
        pass

    path = None
    if '.' in absolute_name:
        parent_name, _, child_name = absolute_name.rpartition('.')
        parent_module = import_module(parent_name)
        path = parent_module.__spec__.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
    else:
        msg = f'No module named {absolute_name!r}'
        raise ModuleNotFoundError(msg, name=absolute_name)
    module = importlib.util.module_from_spec(spec)
    sys.modules[absolute_name] = module
    spec.loader.exec_module(module)
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
```

31.6 importlib.resources – Package resource reading, opening and access

Source code: Lib/importlib/resources/__init__.py

Added in version 3.7.

This module leverages Python' s import system to provide access to *resources* within *packages*.

"Resources" are file-like resources associated with a module or package in Python. The resources may be contained directly in a package, within a subdirectory contained in that package, or adjacent to modules outside a package. Resources may be text or binary. As a result, Python module sources (.py) of a package and compilation artifacts (pycache) are technically de-facto resources of that package. In practice, however, resources are primarily those non-Python artifacts exposed specifically by the package author.

Resources can be opened or read in either binary or text mode.

Resources are roughly akin to files inside directories, though it' s important to keep in mind that this is just a metaphor. Resources and packages **do not** have to exist as physical files and directories on the file system: for example, a package and its resources can be imported from a zip file using *zipimport*.

> **참고**
>
> This module provides functionality similar to pkg_resources Basic Resource Access without the performance overhead of that package. This makes reading resources included in packages easier, with more stable and consistent semantics.
>
> The standalone backport of this module provides more information on using importlib.resources and migrating from pkg_resources to importlib.resources.

Loaders that wish to support resource reading should implement a `get_resource_reader(fullname)` method as specified by *importlib.resources.abc.ResourceReader*.

class `importlib.resources.`**`Anchor`**

Represents an anchor for resources, either a *module object* or a module name as a string. Defined as `Union[str, ModuleType]`.

`importlib.resources.`**`files`**(*anchor:* Anchor | None = *None*)

Returns a *Traversable* object representing the resource container (think directory) and its resources (think files). A Traversable may contain other containers (think subdirectories).

anchor is an optional *Anchor*. If the anchor is a package, resources are resolved from that package. If a module, resources are resolved adjacent to that module (in the same package or the package root). If the anchor is omitted, the caller' s module is used.

Added in version 3.9.

버전 3.12에서 변경: *package* parameter was renamed to *anchor*. *anchor* can now be a non-package module and if omitted will default to the caller' s module. *package* is still accepted for compatibility but will raise a *DeprecationWarning*. Consider passing the anchor positionally or using `importlib_resources >= 5.10` for a compatible interface on older Pythons.

`importlib.resources.`**`as_file`**(*traversable*)

Given a *Traversable* object representing a file or directory, typically from *importlib.resources.files()*, return a context manager for use in a `with` statement. The context manager provides a *pathlib.Path* object.

Exiting the context manager cleans up any temporary file or directory created when the resource was extracted from e.g. a zip file.

Use `as_file` when the Traversable methods (`read_text`, etc) are insufficient and an actual file or directory on the file system is required.

Added in version 3.9.

버전 3.12에서 변경: Added support for *traversable* representing a directory.

31.6.1 Functional API

A set of simplified, backwards-compatible helpers is available. These allow common operations in a single function call.

For all the following functions:

- *anchor* is an *Anchor*, as in *files()*. Unlike in `files`, it may not be omitted.
- *path_names* are components of a resource' s path name, relative to the anchor. For example, to get the text of resource named `info.txt`, use:

```
importlib.resources.read_text(my_module, "info.txt")
```

Like `Traversable.joinpath`, The individual components should use forward slashes (`/`) as path separators. For example, the following are equivalent:

```
importlib.resources.read_binary(my_module, "pics/painting.png")
importlib.resources.read_binary(my_module, "pics", "painting.png")
```

For backward compatibility reasons, functions that read text require an explicit *encoding* argument if multiple *path_names* are given. For example, to get the text of `info/chapter1.txt`, use:

```
importlib.resources.read_text(my_module, "info", "chapter1.txt",
                              encoding='utf-8')
```

`importlib.resources.`**`open_binary`**(*anchor*, **path_names*)

Open the named resource for binary reading.

See *the introduction* for details on *anchor* and *path_names*.

This function returns a `BinaryIO` object, that is, a binary stream open for reading.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).open('rb')
```

버전 3.13에서 변경: Multiple *path_names* are accepted.

`importlib.resources.`**`open_text`**(*anchor*, **path_names*, *encoding='utf-8'*, *errors='strict'*)

Open the named resource for text reading. By default, the contents are read as strict UTF-8.

See *the introduction* for details on *anchor* and *path_names*. *encoding* and *errors* have the same meaning as in built-in `open()`.

For backward compatibility reasons, the *encoding* argument must be given explicitly if there are multiple *path_names*. This limitation is scheduled to be removed in Python 3.15.

This function returns a `TextIO` object, that is, a text stream open for reading.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).open('r', encoding=encoding)
```

버전 3.13에서 변경: Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

`importlib.resources.`**`read_binary`**(*anchor*, **path_names*)

Read and return the contents of the named resource as `bytes`.

See *the introduction* for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).read_bytes()
```

버전 3.13에서 변경: Multiple *path_names* are accepted.

`importlib.resources.`**`read_text`**(*anchor*, **path_names*, *encoding='utf-8'*, *errors='strict'*)

Read and return the contents of the named resource as `str`. By default, the contents are read as strict UTF-8.

See *the introduction* for details on *anchor* and *path_names*. *encoding* and *errors* have the same meaning as in built-in `open()`.

For backward compatibility reasons, the *encoding* argument must be given explicitly if there are multiple *path_names*. This limitation is scheduled to be removed in Python 3.15.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).read_text(encoding=encoding)
```

버전 3.13에서 변경: Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

`importlib.resources.path`(*anchor*, **path_names*)

Provides the path to the *resource* as an actual file system path. This function returns a context manager for use in a `with` statement. The context manager provides a `pathlib.Path` object.

Exiting the context manager cleans up any temporary files created, e.g. when the resource needs to be extracted from a zip file.

For example, the `stat()` method requires an actual file system path; it can be used like this:

```
with importlib.resources.path(anchor, "resource.txt") as fspath:
    result = fspath.stat()
```

See *the introduction* for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
as_file(files(anchor).joinpath(*path_names))
```

버전 3.13에서 변경: Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

`importlib.resources.is_resource`(*anchor*, **path_names*)

Return `True` if the named resource exists, otherwise `False`. This function does not consider directories to be resources.

See *the introduction* for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).is_file()
```

버전 3.13에서 변경: Multiple *path_names* are accepted.

`importlib.resources.contents`(*anchor*, **path_names*)

Return an iterable over the named items within the package or path. The iterable returns names of resources (e.g. files) and non-resources (e.g. directories) as `str`. The iterable does not recurse into subdirectories.

See *the introduction* for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
for resource in files(anchor).joinpath(*path_names).iterdir():
    yield resource.name
```

버전 3.11부터 폐지됨: Prefer `iterdir()` as above, which offers more control over the results and richer functionality.

31.7 `importlib.resources.abc` – Abstract base classes for resources

Source code: Lib/importlib/resources/abc.py

Added in version 3.11.

class importlib.resources.abc.**ResourceReader**

Superseded by TraversableResources

An *abstract base class* to provide the ability to read *resources*.

From the perspective of this ABC, a *resource* is a binary artifact that is shipped within a package. Typically this is something like a data file that lives next to the `__init__.py` file of the package. The purpose of this class is to help abstract out the accessing of such data files so that it does not matter if the package and its data file(s) are stored e.g. in a zip file versus on the file system.

For any of methods of this class, a *resource* argument is expected to be a *path-like object* which represents conceptually just a file name. This means that no subdirectory paths should be included in the *resource* argument. This is because the location of the package the reader is for, acts as the "directory". Hence the metaphor for directories and file names is packages and resources, respectively. This is also why instances of this class are expected to directly correlate to a specific package (instead of potentially representing multiple packages or a module).

Loaders that wish to support resource reading are expected to provide a method called `get_resource_reader(fullname)` which returns an object implementing this ABC's interface. If the module specified by fullname is not a package, this method should return `None`. An object compatible with this ABC should only be returned when the specified module is a package.

버전 3.12부터 폐지됨: Use `importlib.resources.abc.TraversableResources` instead.

abstractmethod open_resource(*resource*)

Returns an opened, *file-like object* for binary reading of the *resource*.

If the resource cannot be found, `FileNotFoundError` is raised.

abstractmethod resource_path(*resource*)

Returns the file system path to the *resource*.

If the resource does not concretely exist on the file system, raise `FileNotFoundError`.

abstractmethod is_resource(*name*)

Returns `True` if the named *name* is considered a resource. `FileNotFoundError` is raised if *name* does not exist.

abstractmethod contents()

Returns an *iterable* of strings over the contents of the package. Do note that it is not required that all names returned by the iterator be actual resources, e.g. it is acceptable to return names for which `is_resource()` would be false.

Allowing non-resource names to be returned is to allow for situations where how a package and its resources are stored are known a priori and the non-resource names would be useful. For instance, returning subdirectory names is allowed so that when it is known that the package and resources are stored on the file system then those subdirectory names can be used directly.

The abstract method returns an iterable of no items.

class importlib.resources.abc.**Traversable**

An object with a subset of `pathlib.Path` methods suitable for traversing directories and opening files.

For a representation of the object on the file-system, use `importlib.resources.as_file()`.

name

Abstract. The base name of this object without any parent references.

abstractmethod iterdir()

Yield Traversable objects in self.

abstractmethod is_dir()

Return `True` if self is a directory.

abstractmethod is_file()

Return `True` if self is a file.

abstractmethod joinpath(**pathsegments*)

Traverse directories according to *pathsegments* and return the result as `Traversable`.

Each *pathsegments* argument may contain multiple names separated by forward slashes (`/`, `posixpath.sep`). For example, the following are equivalent:

```
files.joinpath('subdir', 'subsuddir', 'file.txt')
files.joinpath('subdir/subsuddir/file.txt')
```

Note that some `Traversable` implementations might not be updated to the latest version of the protocol. For compatibility with such implementations, provide a single argument without path separators to each call to `joinpath`. For example:

```
files.joinpath('subdir').joinpath('subsubdir').joinpath('file.txt')
```

버전 3.11에서 변경: `joinpath` accepts multiple *pathsegments*, and these segments may contain forward slashes as path separators. Previously, only a single *child* argument was accepted.

abstractmethod __truediv__(*child*)

Return Traversable child in self. Equivalent to `joinpath(child)`.

abstractmethod open(*mode='r', *args, **kwargs*)

mode may be 'r' or 'rb' to open as text or binary. Return a handle suitable for reading (same as `pathlib.Path.open`).

When opening as text, accepts encoding parameters such as those accepted by `io.TextIOWrapper`.

read_bytes()

Read contents of self as bytes.

read_text(*encoding=None*)

Read contents of self as text.

class importlib.resources.abc.**TraversableResources**

An abstract base class for resource readers capable of serving the `importlib.resources.files()` interface. Subclasses `ResourceReader` and provides concrete implementations of the `ResourceReader`'s abstract methods. Therefore, any loader supplying `TraversableResources` also supplies `ResourceReader`.

Loaders that wish to support resource reading are expected to implement this interface.

abstractmethod files()

Returns a `importlib.resources.abc.Traversable` object for the loaded package.

31.8 importlib.metadata – 패키지 메타데이터 액세스

Added in version 3.8.

버전 3.10에서 변경: `importlib.metadata`는 더는 잠정적이지 않습니다.

Source code: Lib/importlib/metadata/__init__.py

`importlib.metadata` is a library that provides access to the metadata of an installed Distribution Package, such as its entry points or its top-level names (Import Packages, modules, if any). Built in part on Python's import system, this library intends to replace similar functionality in the entry point API and metadata API of `pkg_resources`. Along with `importlib.resources`, this package can eliminate the need to use the older and less efficient `pkg_resources` package.

`importlib.metadata` operates on third-party *distribution packages* installed into Python's `site-packages` directory via tools such as pip. Specifically, it works with distributions with discoverable `dist-info` or `egg-info` directories, and metadata defined by the Core metadata specifications.

중요

These are *not* necessarily equivalent to or correspond 1:1 with the top-level *import package* names that can be imported inside Python code. One *distribution package* can contain multiple *import packages* (and single modules), and one top-level *import package* may map to multiple *distribution packages* if it is a namespace package. You can use *packages_distributions*() to get a mapping between them.

By default, distribution metadata can live on the file system or in zip archives on `sys.path`. Through an extension mechanism, the metadata can live almost anywhere.

더 보기

https://importlib-metadata.readthedocs.io/
The documentation for `importlib_metadata`, which supplies a backport of `importlib.metadata`. This includes an API reference for this module' s classes and functions, as well as a migration guide for existing users of `pkg_resources`.

31.8.1 개요

Let' s say you wanted to get the version string for a Distribution Package you' ve installed using `pip`. We start by creating a virtual environment and installing something into it:

```
$ python -m venv example
$ source example/bin/activate
(example) $ python -m pip install wheel
```

다음을 실행하여 `wheel`에 대한 버전 문자열을 얻을 수 있습니다:

```
(example) $ python
>>> from importlib.metadata import version
>>> version('wheel')
'0.32.3'
```

You can also get a collection of entry points selectable by properties of the EntryPoint (typically 'group' or 'name'), such as `console_scripts`, `distutils.commands` and others. Each group contains a collection of *EntryPoint* objects.

여러분은 배포 메타데이터를 얻을 수 있습니다:

```
>>> list(metadata('wheel'))
['Metadata-Version', 'Name', 'Version', 'Summary', 'Home-page', 'Author', 'Author-
↪email', 'Maintainer', 'Maintainer-email', 'License', 'Project-URL', 'Project-URL
↪', 'Project-URL', 'Keywords', 'Platform', 'Classifier', 'Classifier', 'Classifier
↪', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier',
↪'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Requires-Python',
↪'Provides-Extra', 'Requires-Dist', 'Requires-Dist']
```

또한 배포의 버전 번호를 가져오고, 구성 파일을 나열하고, 배포의 요구사항 리스트를 얻을 수 있습니다.

exception `importlib.metadata.`**`PackageNotFoundError`**

Subclass of `ModuleNotFoundError` raised by several functions in this module when queried for a distribution package which is not installed in the current Python environment.

31.8.2 기능적 API

이 패키지는 공용 API를 통해 다음과 같은 기능을 제공합니다.

진입 지점

importlib.metadata.**entry_points**(***select_params*)

Returns a *EntryPoints* instance describing entry points for the current environment. Any given keyword parameters are passed to the `select()` method for comparison to the attributes of the individual entry point definitions.

Note: it is not currently possible to query for entry points based on their `EntryPoint.dist` attribute (as different `Distribution` instances do not currently compare equal, even if they have the same attributes)

class importlib.metadata.**EntryPoints**

Details of a collection of installed entry points.

Also provides a `.groups` attribute that reports all identified entry point groups, and a `.names` attribute that reports all identified entry point names.

class importlib.metadata.**EntryPoint**

Details of an installed entry point.

각 `EntryPoint` 인스턴스에는 `.name`, `.group` 및 `.value` 어트리뷰트가 있고 값을 결정하는 `.load()` 메서드가 있습니다. `.value` 어트리뷰트의 구성 요소를 가져오기 위한 `.module`, `.attr` 및 `.extras` 어트리뷰트도 있으며, `.dist`는 진입 지점을 제공하는 배포 패키지에 대한 정보를 얻는데 사용됩니다.

모든 진입 지점을 조회합니다:

```
>>> eps = entry_points()
```

The `entry_points()` function returns a `EntryPoints` object, a collection of all `EntryPoint` objects with `names` and `groups` attributes for convenience:

```
>>> sorted(eps.groups)
['console_scripts', 'distutils.commands', 'distutils.setup_keywords', 'egg_info.
↪writers', 'setuptools.installation']
```

`EntryPoints` has a `select()` method to select entry points matching specific properties. Select entry points in the `console_scripts` group:

```
>>> scripts = eps.select(group='console_scripts')
```

Equivalently, since `entry_points()` passes keyword arguments through to select:

```
>>> scripts = entry_points(group='console_scripts')
```

Pick out a specific script named "wheel" (found in the wheel project):

```
>>> 'wheel' in scripts.names
True
>>> wheel = scripts['wheel']
```

Equivalently, query for that entry point during selection:

```
>>> (wheel,) = entry_points(group='console_scripts', name='wheel')
>>> (wheel,) = entry_points().select(group='console_scripts', name='wheel')
```

Inspect the resolved entry point:

```
>>> wheel
EntryPoint(name='wheel', value='wheel.cli:main', group='console_scripts')
>>> wheel.module
'wheel.cli'
>>> wheel.attr
'main'
>>> wheel.extras
[]
>>> main = wheel.load()
>>> main
<function main at 0x103528488>
```

The `group` and `name` are arbitrary values defined by the package author and usually a client will wish to resolve all entry points for a particular group. Read the setuptools docs for more information on entry points, their definition, and usage.

버전 3.12에서 변경: The "selectable" entry points were introduced in `importlib_metadata` 3.6 and Python 3.10. Prior to those changes, `entry_points` accepted no parameters and always returned a dictionary of entry points, keyed by group. With `importlib_metadata` 5.0 and Python 3.12, `entry_points` always returns an `EntryPoints` object. See backports.entry_points_selectable for compatibility options.

버전 3.13에서 변경: `EntryPoint` objects no longer present a tuple-like interface (`__getitem__()`).

배포 메타데이터

`importlib.metadata.`**`metadata`**(*distribution_name*)

Return the distribution metadata corresponding to the named distribution package as a *`PackageMetadata`* instance.

Raises *`PackageNotFoundError`* if the named distribution package is not installed in the current Python environment.

class `importlib.metadata.`**`PackageMetadata`**

A concrete implementation of the PackageMetadata protocol.

In addition to providing the defined protocol methods and attributes, subscripting the instance is equivalent to calling the `get()` method.

모든 배포 패키지는 `metadata()` 함수를 사용하여 추출할 수 있는 몇 가지 메타 데이터가 포함되어 있습니다:

```
>>> wheel_metadata = metadata('wheel')
```

반환된 데이터 구조의 키는 메타데이터 키워드의 이름을 지정하고, 값은 배포 메타데이터에서 구문 분석하지 않은 채로 반환됩니다:

```
>>> wheel_metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

`PackageMetadata` also presents a `json` attribute that returns all the metadata in a JSON-compatible form per **PEP 566**:

```
>>> wheel_metadata.json['requires_python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

The full set of available metadata is not described here. See the PyPA Core metadata specification for additional details.

버전 3.10에서 변경: The `Description` is now included in the metadata when presented through the payload. Line continuation characters have been removed.

The `json` attribute was added.

배포 버전

importlib.metadata.**version**(*distribution_name*)

Return the installed distribution package version for the named distribution package.

Raises *PackageNotFoundError* if the named distribution package is not installed in the current Python environment.

version() 함수는 배포 패키지의 버전 번호를 문자열로 가져오는 가장 빠른 방법입니다:

```
>>> version('wheel')
'0.32.3'
```

배포 파일

importlib.metadata.**files**(*distribution_name*)

Return the full set of files contained within the named distribution package.

Raises *PackageNotFoundError* if the named distribution package is not installed in the current Python environment.

Returns *None* if the distribution is found but the installation database records reporting the files associated with the distribution package are missing.

class importlib.metadata.**PackagePath**

A *pathlib.PurePath* derived object with additional dist, size, and hash properties corresponding to the distribution package' s installation metadata for that file.

The files() function takes a Distribution Package name and returns all of the files installed by this distribution. Each file is reported as a *PackagePath* instance. For example:

```
>>> util = [p for p in files('wheel') if 'util.py' in str(p)][0]
>>> util
PackagePath('wheel/util.py')
>>> util.size
859
>>> util.dist
<importlib.metadata._hooks.PathDistribution object at 0x101e0cef0>
>>> util.hash
<FileHash mode: sha256 value: bYkw5oMccfazVCoYQwKkkemoVyMAFoR34mmKBx8R1NI>
```

일단 파일을 얻으면, 내용을 읽을 수도 있습니다:

```
>>> print(util.read_text())
import base64
import sys
...
def as_bytes(s):
    if isinstance(s, text_type):
        return s.encode('utf-8')
    return s
```

You can also use the locate() method to get the absolute path to the file:

```
>>> util.locate()
PosixPath('/home/gustav/example/lib/site-packages/wheel/util.py')
```

메타 데이터 파일 목록 파일(RECORD 나 SOURCES.txt)이 누락된 경우, files() 는 *None*을 반환합니다. 대상 배포에 메타 데이터가 있음이 알려지지 않았을 때, 이 조건에 대한 보호로 호출자는 files() 에 대한 호출을 always_iterable이나 다른 것으로 감쌀 수 있습니다.

배포 요구 사항

importlib.metadata.**requires**(*distribution_name*)

Return the declared dependency specifiers for the named distribution package.

Raises *PackageNotFoundError* if the named distribution package is not installed in the current Python environment.

배포 패키지의 전체 요구 사항을 얻으려면, requires() 함수를 사용하십시오:

```
>>> requires('wheel')
["pytest (>=3.0.0) ; extra == 'test'", "pytest-cov ; extra == 'test'"]
```

Mapping import to distribution packages

importlib.metadata.**packages_distributions**()

Return a mapping from the top level module and import package names found via *sys.meta_path* to the names of the distribution packages (if any) that provide the corresponding files.

To allow for namespace packages (which may have members provided by multiple distribution packages), each top level import name maps to a list of distribution names rather than mapping directly to a single name.

A convenience method to resolve the Distribution Package name (or names, in the case of a namespace package) that provide each importable top-level Python module or Import Package:

```
>>> packages_distributions()
{'importlib_metadata': ['importlib-metadata'], 'yaml': ['PyYAML'], 'jaraco': [
↪'jaraco.classes', 'jaraco.functools'], ...}
```

Some editable installs, do not supply top-level names, and thus this function is not reliable with such installs.

Added in version 3.10.

31.8.3 배포

importlib.metadata.**distribution**(*distribution_name*)

Return a *Distribution* instance describing the named distribution package.

Raises *PackageNotFoundError* if the named distribution package is not installed in the current Python environment.

class importlib.metadata.**Distribution**

Details of an installed distribution package.

Note: different Distribution instances do not currently compare equal, even if they relate to the same installed distribution and accordingly have the same attributes.

위에 기술된 모듈 수준 API가 가장 일반적이며 편리한 사용법이지만, Distribution 클래스에서 모든 정보를 얻을 수 있습니다. Distribution은 파이썬 배포 패키지의 메타 데이터를 나타내는 추상 객체입니다. *distribution()* 함수를 호출해서, 설치된 배포 패키지의 구상 Distribution 서브클래스 인스턴스를 얻을 수 있습니다:

```
>>> from importlib.metadata import distribution
>>> dist = distribution('wheel')
>>> type(dist)
<class 'importlib.metadata.PathDistribution'>
```

따라서, 버전 번호를 얻는 다른 방법은 Distribution 인스턴스를 사용하는 것입니다:

```
>>> dist.version
'0.32.3'
```

Distribution 인스턴스에서 사용할 수 있는 모든 종류의 추가 메타 데이터가 있습니다:

```
>>> dist.metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
>>> dist.metadata['License']
'MIT'
```

For editable packages, an `origin` property may present **PEP 610** metadata:

```
>>> dist.origin.url
'file:///path/to/wheel-0.32.3.editable-py3-none-any.whl'
```

The full set of available metadata is not described here. See the PyPA Core metadata specification for additional details.

Added in version 3.13: The `.origin` property was added.

31.8.4 배포 발견

By default, this package provides built-in support for discovery of metadata for file system and zip file Distribution Packages. This metadata finder search defaults to `sys.path`, but varies slightly in how it interprets those values from how other import machinery does. In particular:

- `importlib.metadata` does not honor *`bytes`* objects on `sys.path`.
- `importlib.metadata` will incidentally honor *`pathlib.Path`* objects on `sys.path` even though such values will be ignored for imports.

31.8.5 Implementing Custom Providers

`importlib.metadata` address two API surfaces, one for *consumers* and another for *providers*. Most users are consumers, consuming metadata provided by the packages. There are other use-cases, however, where users wish to expose metadata through some other mechanism, such as alongside a custom importer. Such a use case calls for a *custom provider*.

Because Distribution Package metadata is not available through *`sys.path`* searches, or package loaders directly, the metadata for a distribution is found through import system finders. To find a distribution package' s metadata, `importlib.metadata` queries the list of *meta path finders* on *`sys.meta_path`*.

The implementation has hooks integrated into the `PathFinder`, serving metadata for distribution packages found on the file system.

추상 클래스 *`importlib.abc.MetaPathFinder`*는 파이썬의 임포트 시스템에 의해 파인더가 기대하는 인터페이스를 정의합니다. `importlib.metadata`는 *`sys.meta_path`*의 파인더에서 선택적인 `find_distributions` 콜러블을 조회함으로써 이 프로토콜을 확장하고 이 확장된 인터페이스를 다음과 같은 추상 메서드를 정의하는 `DistributionFinder` 추상 베이스 클래스로 제공합니다:

```
@abc.abstractmethod
def find_distributions(context=DistributionFinder.Context()) ->␣
↪Iterable[Distribution]:
    """Return an iterable of all Distribution instances capable of
    loading the metadata for packages for the indicated ``context``.
    """
```

The `DistributionFinder.Context` object provides `.path` and `.name` properties indicating the path to search and name to match and may supply other relevant context sought by the consumer.

In practice, to support finding distribution package metadata in locations other than the file system, subclass `Distribution` and implement the abstract methods. Then from a custom finder, return instances of this derived `Distribution` in the `find_distributions()` method.

예제

Imagine a custom finder that loads Python modules from a database:

```
class DatabaseImporter(importlib.abc.MetaPathFinder):
    def __init__(self, db):
        self.db = db

    def find_spec(self, fullname, target=None) -> ModuleSpec:
        return self.db.spec_from_name(fullname)

sys.meta_path.append(DatabaseImporter(connect_db(...)))
```

That importer now presumably provides importable modules from a database, but it provides no metadata or entry points. For this custom importer to provide metadata, it would also need to implement `DistributionFinder`:

```
from importlib.metadata import DistributionFinder

class DatabaseImporter(DistributionFinder):
    ...

    def find_distributions(self, context=DistributionFinder.Context()):
        query = dict(name=context.name) if context.name else {}
        for dist_record in self.db.query_distributions(query):
            yield DatabaseDistribution(dist_record)
```

In this way, `query_distributions` would return records for each distribution served by the database matching the query. For example, if `requests-1.0` is in the database, `find_distributions` would yield a `DatabaseDistribution` for `Context(name='requests')` or `Context(name=None)`.

For the sake of simplicity, this example ignores `context.path`. The `path` attribute defaults to `sys.path` and is the set of import paths to be considered in the search. A `DatabaseImporter` could potentially function without any concern for a search path. Assuming the importer does no partitioning, the "path" would be irrelevant. In order to illustrate the purpose of `path`, the example would need to illustrate a more complex `DatabaseImporter` whose behavior varied depending on `sys.path`/`PYTHONPATH`. In that case, the `find_distributions` should honor the `context.path` and only yield `Distributions` pertinent to that path.

`DatabaseDistribution`, then, would look something like:

```
class DatabaseDistribution(importlib.metadata.Distribution):
    def __init__(self, record):
        self.record = record

    def read_text(self, filename):
        """
        Read a file like "METADATA" for the current distribution.
        """
        if filename == "METADATA":
            return f"""Name: {self.record.name}
Version: {self.record.version}
"""
        if filename == "entry_points.txt":
            return "\n".join(
              f"""[{ep.group}]\n{ep.name}={ep.value}"""
              for ep in self.record.entry_points)

    def locate_file(self, path):
        raise RuntimeError("This distribution has no file system")
```

This basic implementation should provide metadata and entry points for packages served by the

DatabaseImporter, assuming that the record supplies suitable .name, .version, and .entry_points attributes.

The DatabaseDistribution may also provide other metadata files, like RECORD (required for Distribution.files) or override the implementation of Distribution.files. See the source for more inspiration.

31.9 The initialization of the sys.path module search path

A module search path is initialized when Python starts. This module search path may be accessed at *sys.path*.

The first entry in the module search path is the directory that contains the input script, if there is one. Otherwise, the first entry is the current directory, which is the case when executing the interactive shell, a -c command, or -m module.

The PYTHONPATH environment variable is often used to add directories to the search path. If this environment variable is found then the contents are added to the module search path.

> **참고**
>
> PYTHONPATH will affect all installed Python versions/environments. Be wary of setting this in your shell profile or global environment variables. The *site* module offers more nuanced techniques as mentioned below.

The next items added are the directories containing standard Python modules as well as any *extension modules* that these modules depend on. Extension modules are .pyd files on Windows and .so files on other platforms. The directory with the platform-independent Python modules is called prefix. The directory with the extension modules is called exec_prefix.

The PYTHONHOME environment variable may be used to set the prefix and exec_prefix locations. Otherwise these directories are found by using the Python executable as a starting point and then looking for various 'landmark' files and directories. Note that any symbolic links are followed so the real Python executable location is used as the search starting point. The Python executable location is called home.

Once home is determined, the prefix directory is found by first looking for python*majorversionminorversion*.zip (python311.zip). On Windows the zip archive is searched for in home and on Unix the archive is expected to be in lib. Note that the expected zip archive location is added to the module search path even if the archive does not exist. If no archive was found, Python on Windows will continue the search for prefix by looking for Lib\os.py. Python on Unix will look for lib/python*majorversion*.*minorversion*/os.py (lib/python3.11/os.py). On Windows prefix and exec_prefix are the same, however on other platforms lib/python*majorversion*.*minorversion*/lib-dynload (lib/python3.11/lib-dynload) is searched for and used as an anchor for exec_prefix. On some platforms lib may be lib64 or another value, see *sys.platlibdir* and PYTHONPLATLIBDIR.

Once found, prefix and exec_prefix are available at *sys.base_prefix* and *sys.base_exec_prefix* respectively.

If PYTHONHOME is not set, and a pyvenv.cfg file is found alongside the main executable, or in its parent directory, *sys.prefix* and *sys.exec_prefix* get set to the directory containing pyvenv.cfg, otherwise they are set to the same value as *sys.base_prefix* and *sys.base_exec_prefix*, respectively. This is used by *Virtual Environments*.

Finally, the *site* module is processed and site-packages directories are added to the module search path. A common way to customize the search path is to create *sitecustomize* or *usercustomize* modules as described in the *site* module documentation.

> **참고**
>
> Certain command line options may further affect path calculations. See -E, -I, -s and -S for further details.

버전 3.14에서 변경: *sys.prefix* and *sys.exec_prefix* are now set to the `pyvenv.cfg` directory during the path initialization. This was previously done by *site*, therefore affected by `-S`.

31.9.1 Virtual Environments

Virtual environments place a `pyvenv.cfg` file in their prefix, which causes *sys.prefix* and *sys.exec_prefix* to point to them, instead of the base installation.

The `prefix` and `exec_prefix` values of the base installation are available at *sys.base_prefix* and *sys.base_exec_prefix*.

As well as being used as a marker to identify virtual environments, `pyvenv.cfg` may also be used to configure the *site* initialization. Please refer to *site*' s *virtual environments documentation*.

> **참고**
>
> `PYTHONHOME` overrides the `pyvenv.cfg` detection.

> **참고**
>
> There are other ways how "virtual environments" could be implemented, this documentation refers implementations based on the `pyvenv.cfg` mechanism, such as *venv*. Most virtual environment implementations follow the model set by *venv*, but there may be exotic implementations that diverge from it.

31.9.2 _pth files

To completely override *sys.path* create a `._pth` file with the same name as the shared library or executable (`python._pth` or `python311._pth`). The shared library path is always known on Windows, however it may not be available on other platforms. In the `._pth` file specify one line for each path to add to *sys.path*. The file based on the shared library name overrides the one based on the executable, which allows paths to be restricted for any program loading the runtime if desired.

When the file exists, all registry and environment variables are ignored, isolated mode is enabled, and *site* is not imported unless one line in the file specifies `import site`. Blank paths and lines starting with `#` are ignored. Each path may be absolute or relative to the location of the file. Import statements other than to `site` are not permitted, and arbitrary code cannot be specified.

Note that `.pth` files (without leading underscore) will be processed normally by the *site* module when `import site` has been specified.

31.9.3 Embedded Python

If Python is embedded within another application `Py_InitializeFromConfig()` and the `PyConfig` structure can be used to initialize Python. The path specific details are described at init-path-config.

> **더 보기**
>
> - windows_finding_modules for detailed Windows notes.
> - using-on-unix for Unix details.

CHAPTER 32

파이썬 언어 서비스

파이썬은 파이썬 언어로 작업하는 데 도움이 되는 여러 모듈을 제공합니다. 이 모듈들은 토큰화, 구문 분석, 문법 분석, 바이트 코드 역 어셈블리 및 기타 다양한 기능을 지원합니다.

이 모듈들은 다음과 같습니다:

32.1 ast — Abstract syntax trees

소스 코드: Lib/ast.py

---

ast 모듈은 파이썬 응용 프로그램이 파이썬 추상 구문 문법의 트리를 처리하는 데 도움을 줍니다. 추상 구문 자체는 각 파이썬 릴리스마다 바뀔 수 있습니다; 이 모듈은 프로그래밍 방식으로 현재 문법의 모양을 찾는 데 도움이 됩니다.

*ast.PyCF_ONLY_AST*를 플래그로 *compile()* 내장 함수에 전달하거나, 이 모듈에서 제공된 *parse()* 도우미를 사용하여 추상 구문 트리를 생성할 수 있습니다. 결과는 클래스가 모두 *ast.AST*에서 상속되는 객체들의 트리가 됩니다. 내장 *compile()* 함수를 사용하여 추상 구문 트리를 파이썬 코드 객체로 컴파일할 수 있습니다.

32.1.1 Abstract grammar

추상 문법은 현재 다음과 같이 정의됩니다:

```
-- ASDL's 4 builtin types are:
-- identifier, int, string, constant

module Python
{
    mod = Module(stmt* body, type_ignore* type_ignores)
        | Interactive(stmt* body)
        | Expression(expr body)
        | FunctionType(expr* argtypes, expr returns)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns,
                       string? type_comment, type_param* type_params)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
          | AsyncFunctionDef(identifier name, arguments args,
                            stmt* body, expr* decorator_list, expr? returns,
                            string? type_comment, type_param* type_params)

          | ClassDef(identifier name,
             expr* bases,
             keyword* keywords,
             stmt* body,
             expr* decorator_list,
             type_param* type_params)
          | Return(expr? value)

          | Delete(expr* targets)
          | Assign(expr* targets, expr value, string? type_comment)
          | TypeAlias(expr name, type_param* type_params, expr value)
          | AugAssign(expr target, operator op, expr value)
          -- 'simple' indicates that we annotate simple name without parens
          | AnnAssign(expr target, expr annotation, expr? value, int simple)

          -- use 'orelse' because else is a keyword in target languages
          | For(expr target, expr iter, stmt* body, stmt* orelse, string? type_
↪comment)
          | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, string?␣
↪type_comment)
          | While(expr test, stmt* body, stmt* orelse)
          | If(expr test, stmt* body, stmt* orelse)
          | With(withitem* items, stmt* body, string? type_comment)
          | AsyncWith(withitem* items, stmt* body, string? type_comment)

          | Match(expr subject, match_case* cases)

          | Raise(expr? exc, expr? cause)
          | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
          | TryStar(stmt* body, excepthandler* handlers, stmt* orelse, stmt*␣
↪finalbody)
          | Assert(expr test, expr? msg)

          | Import(alias* names)
          | ImportFrom(identifier? module, alias* names, int? level)

          | Global(identifier* names)
          | Nonlocal(identifier* names)
          | Expr(expr value)
          | Pass | Break | Continue

          -- col_offset is the byte offset in the utf8 string the parser uses
          attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

          -- BoolOp() can use left & right?
    expr = BoolOp(boolop op, expr* values)
         | NamedExpr(expr target, expr value)
         | BinOp(expr left, operator op, expr right)
         | UnaryOp(unaryop op, expr operand)
         | Lambda(arguments args, expr body)
         | IfExp(expr test, expr body, expr orelse)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        | Dict(expr?* keys, expr* values)
        | Set(expr* elts)
        | ListComp(expr elt, comprehension* generators)
        | SetComp(expr elt, comprehension* generators)
        | DictComp(expr key, expr value, comprehension* generators)
        | GeneratorExp(expr elt, comprehension* generators)
        -- the grammar constrains where yield expressions can occur
        | Await(expr value)
        | Yield(expr? value)
        | YieldFrom(expr value)
        -- need sequences for compare to distinguish between
        -- x < 4 < 3 and (x < 4) < 3
        | Compare(expr left, cmpop* ops, expr* comparators)
        | Call(expr func, expr* args, keyword* keywords)
        | FormattedValue(expr value, int conversion, expr? format_spec)
        | Interpolation(expr value, constant str, int conversion, expr? format_
↪spec)
        | JoinedStr(expr* values)
        | TemplateStr(expr* values)
        | Constant(constant value, string? kind)

        -- the following expression can appear in assignment context
        | Attribute(expr value, identifier attr, expr_context ctx)
        | Subscript(expr value, expr slice, expr_context ctx)
        | Starred(expr value, expr_context ctx)
        | Name(identifier id, expr_context ctx)
        | List(expr* elts, expr_context ctx)
        | Tuple(expr* elts, expr_context ctx)

        -- can appear only in Subscript
        | Slice(expr? lower, expr? upper, expr? step)

         -- col_offset is the byte offset in the utf8 string the parser uses
         attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

   expr_context = Load | Store | Del

   boolop = And | Or

   operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
                | RShift | BitOr | BitXor | BitAnd | FloorDiv

   unaryop = Invert | Not | UAdd | USub

   cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

   comprehension = (expr target, expr iter, expr* ifs, int is_async)

   excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                   attributes (int lineno, int col_offset, int? end_lineno, int?␣
↪end_col_offset)

   arguments = (arg* posonlyargs, arg* args, arg? vararg, arg* kwonlyargs,
                expr* kw_defaults, arg? kwarg, expr* defaults)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    arg = (identifier arg, expr? annotation, string? type_comment)
           attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)
               attributes (int lineno, int col_offset, int? end_lineno, int? end_
↪col_offset)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)
             attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

    withitem = (expr context_expr, expr? optional_vars)

    match_case = (pattern pattern, expr? guard, stmt* body)

    pattern = MatchValue(expr value)
            | MatchSingleton(constant value)
            | MatchSequence(pattern* patterns)
            | MatchMapping(expr* keys, pattern* patterns, identifier? rest)
            | MatchClass(expr cls, pattern* patterns, identifier* kwd_attrs,␣
↪pattern* kwd_patterns)

            | MatchStar(identifier? name)
            -- The optional "rest" MatchMapping parameter handles capturing extra␣
↪mapping keys

            | MatchAs(pattern? pattern, identifier? name)
            | MatchOr(pattern* patterns)

             attributes (int lineno, int col_offset, int end_lineno, int end_col_
↪offset)

    type_ignore = TypeIgnore(int lineno, string tag)

    type_param = TypeVar(identifier name, expr? bound, expr? default_value)
               | ParamSpec(identifier name, expr? default_value)
               | TypeVarTuple(identifier name, expr? default_value)
               attributes (int lineno, int col_offset, int end_lineno, int end_col_
↪offset)
}
```

32.1.2 노드 클래스

class ast.**AST**

This is the base of all AST node classes. The actual node classes are derived from the `Parser/Python.asdl` file, which is reproduced *above*. They are defined in the `_ast` C module and re-exported in *ast*.

추상 문법의 각 좌변 심볼마다 하나의 클래스가 정의되어 있습니다 (예를 들어, `ast.stmt` 나 `ast.expr`). 또한, 우변의 생성자마다 하나의 클래스가 정의되어 있습니다; 이 클래스는 좌변 트리의 클래스에서 상속됩니다. 예를 들어, *ast.BinOp*는 `ast.expr`에서 상속됩니다. 대안을 갖는 생성 규칙 (일명 "합")의 경우, 좌변 클래스는 추상입니다: 특정 생성자 노드의 인스턴스만 만들어집니다.

_fields

Each concrete class has an attribute `_fields` which gives the names of all child nodes.

구상 클래스의 각 인스턴스에는 각 자식 노드마다 문법에 정의된 형의 어트리뷰트가 하나씩 있습니다. 예를 들어, *ast.BinOp* 인스턴스는 `ast.expr` 형의 어트리뷰트 `left`를 갖습니다.

문법에서 이러한 어트리뷰트가 선택적으로 표시되면 (물음표를 사용해서), 값은 `None`일 수 있습니다. 어트리뷰트가 0개 이상의 값을 가질 수 있으면 (애스터리스크로 표시됩니다), 값은 파이썬 리스트로 표현됩니다. *compile()*로 AST를 컴파일할 때 가능한 모든 어트리뷰트가 존재하고 유효한 값을 가져야 합니다.

_field_types

The `_field_types` attribute on each concrete class is a dictionary mapping field names (as also listed in *_fields*) to their types.

```
>>> ast.TypeVar._field_types
{'name': <class 'str'>, 'bound': ast.expr | None, 'default_value': ast.
↪expr | None}
```

Added in version 3.13.

lineno

col_offset

end_lineno

end_col_offset

Instances of `ast.expr` and `ast.stmt` subclasses have *lineno*, *col_offset*, *end_lineno*, and *end_col_offset* attributes. The *lineno* and *end_lineno* are the first and last line numbers of source text span (1-indexed so the first line is line 1) and the *col_offset* and *end_col_offset* are the corresponding UTF-8 byte offsets of the first and last tokens that generated the node. The UTF-8 offset is recorded because the parser uses UTF-8 internally.

종료 위치는 컴파일러에 필요하지 않아서 선택 사항입니다. 종료 오프셋은 마지막 심볼 뒤입니다. 예를 들어 `source_line[node.col_offset : node.end_col_offset]`를 사용하여 한 줄 표현식 노드의 소스 세그먼트를 가져올 수 있습니다.

`ast.T` 클래스의 생성자는 다음과 같이 인자를 구문 분석합니다:

- 위치 인자가 있으면, `T._fields`에 있는 항목 수만큼 있어야 합니다; 이러한 이름의 어트리뷰트로 대입될 것입니다.
- 키워드 인자가 있으면, 같은 이름의 어트리뷰트를 지정된 값으로 설정합니다.

예를 들어, *ast.UnaryOp* 노드를 만들고 채우려면, 다음과 같이 할 수 있습니다

```
node = ast.UnaryOp(ast.USub(), ast.Constant(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

If a field that is optional in the grammar is omitted from the constructor, it defaults to `None`. If a list field is omitted, it defaults to the empty list. If a field of type `ast.expr_context` is omitted, it defaults to *Load()*. If any other field is omitted, a *DeprecationWarning* is raised and the AST node will not have this field. In Python 3.15, this condition will raise an error.

버전 3.8에서 변경: *ast.Constant* 클래스는 이제 모든 상수에 사용됩니다.

버전 3.9에서 변경: 단순 인덱스는 값으로 표현되고, 확장 슬라이스는 튜플로 표현됩니다.

버전 3.14에서 변경: The `__repr__()` output of *AST* nodes includes the values of the node fields.

버전 3.8부터 폐지됨: Old classes `ast.Num`, `ast.Str`, `ast.Bytes`, `ast.NameConstant` and `ast.Ellipsis` are still available, but they will be removed in future Python releases. In the meantime, instantiating them will return an instance of a different class.

버전 3.9부터 폐지됨: Old classes `ast.Index` and `ast.ExtSlice` are still available, but they will be removed in future Python releases. In the meantime, instantiating them will return an instance of a different class.

Deprecated since version 3.13, will be removed in version 3.15: Previous versions of Python allowed the creation of AST nodes that were missing required fields. Similarly, AST node constructors allowed arbitrary keyword arguments

that were set as attributes of the AST node, even if they did not match any of the fields of the AST node. This behavior is deprecated and will be removed in Python 3.15.

> **참고**
>
> 여기에 표시된 특정 노드 클래스에 대한 설명은 처음에는 환상적인 Green Tree Snakes 프로젝트와 모든 기여자로부터 차용했습니다.

Root nodes

class ast.**Module**(*body*, *type_ignores*)

A Python module, as with file input. Node type generated by `ast.parse()` in the default `"exec"` *mode*.

`body` is a `list` of the module' s 문장.

`type_ignores` is a `list` of the module' s type ignore comments; see `ast.parse()` for more details.

```
>>> print(ast.dump(ast.parse('x = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1))])
```

class ast.**Expression**(*body*)

A single Python expression input. Node type generated by `ast.parse()` when *mode* is `"eval"`.

`body` is a single node, one of the *expression types*.

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class ast.**Interactive**(*body*)

A single interactive input, like in tut-interac. Node type generated by `ast.parse()` when *mode* is `"single"`.

`body` is a `list` of *statement nodes*.

```
>>> print(ast.dump(ast.parse('x = 1; y = 2', mode='single'), indent=4))
Interactive(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1)),
        Assign(
            targets=[
                Name(id='y', ctx=Store())],
            value=Constant(value=2))])
```

class ast.**FunctionType**(*argtypes*, *returns*)

A representation of an old-style type comments for functions, as Python versions prior to 3.5 didn' t support **PEP 484** annotations. Node type generated by `ast.parse()` when *mode* is `"func_type"`.

Such type comments would look like this:

```
def sum_two_number(a, b):
    # type: (int, int) -> int
    return a + b
```

argtypes is a *list* of *expression nodes*.

returns is a single *expression node*.

```
>>> print(ast.dump(ast.parse('(int, str) -> List[int]', mode='func_type'),␣
↪indent=4))
FunctionType(
    argtypes=[
        Name(id='int', ctx=Load()),
        Name(id='str', ctx=Load())],
    returns=Subscript(
        value=Name(id='List', ctx=Load()),
        slice=Name(id='int', ctx=Load()),
        ctx=Load()))
```

Added in version 3.8.

리터럴

class ast.**Constant**(*value*)

A constant value. The value attribute of the Constant literal contains the Python object it represents. The values represented can be instances of *str*, *bytes*, *int*, *float*, *complex*, and *bool*, and the constants *None* and *Ellipsis*.

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class ast.**FormattedValue**(*value*, *conversion*, *format_spec*)

f-문자열에서 단일 포매팅 필드를 나타내는 노드. 문자열에 단일 포매팅 필드가 포함되어 있고 다른 것이 없으면 노드를 분리 할 수 있습니다, 그렇지 않으면 *JoinedStr*에 나타납니다.

- value는 모든 표현식 노드(가령 리터럴, 변수 또는 함수 호출)입니다.
- conversion은 정수입니다:
  - -1: 포매팅 없음
  - 97 (ord('a')): !a *ASCII* formatting
  - 114 (ord('r')): !r *repr()* formatting
  - 115 (ord('s')): !s *string* formatting
- format_spec은 값의 포매팅을 나타내는 *JoinedStr* 노드이거나, 표맷이 지정되지 않았으면 None입니다. conversion과 format_spec을 동시에 설정할 수 있습니다.

class ast.**JoinedStr**(*values*)

일련의 *FormattedValue*와 *Constant* 노드로 구성된 f-문자열.

```
>>> print(ast.dump(ast.parse('f"sin({a}) is {sin(a):.3}"', mode='eval'),␣
↪indent=4))
Expression(
    body=JoinedStr(
        values=[
            Constant(value='sin('),
            FormattedValue(
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                value=Name(id='a', ctx=Load()),
                conversion=-1),
            Constant(value=') is '),
            FormattedValue(
                value=Call(
                    func=Name(id='sin', ctx=Load()),
                    args=[
                        Name(id='a', ctx=Load())]),
                conversion=-1,
                format_spec=JoinedStr(
                    values=[
                        Constant(value='.3')]))]))
```

class ast.**TemplateStr**(*values*, /)

Added in version 3.14.

Node representing a template string literal, comprising a series of `Interpolation` and `Constant` nodes. These nodes may be any order, and do not need to be interleaved.

```
>>> expr = ast.parse('t"{name} finished {place:ordinal}"', mode='eval')
>>> print(ast.dump(expr, indent=4))
Expression(
    body=TemplateStr(
        values=[
            Interpolation(
                value=Name(id='name', ctx=Load()),
                str='name',
                conversion=-1),
            Constant(value=' finished '),
            Interpolation(
                value=Name(id='place', ctx=Load()),
                str='place',
                conversion=-1,
                format_spec=JoinedStr(
                    values=[
                        Constant(value='ordinal')]))]))
```

class ast.**Interpolation**(*value*, *str*, *conversion*, *format_spec=None*)

Added in version 3.14.

Node representing a single interpolation field in a template string literal.

- `value` is any expression node (such as a literal, a variable, or a function call). This has the same meaning as `FormattedValue.value`.
- `str` is a constant containing the text of the interpolation expression.
- `conversion`은 정수입니다:
  - -1: no conversion
  - 97 (`ord('a')`): `!a` *ASCII* conversion
  - 114 (`ord('r')`): `!r` *repr()* conversion
  - 115 (`ord('s')`): `!s` *string* conversion

  This has the same meaning as `FormattedValue.conversion`.
- `format_spec` is a *JoinedStr* node representing the formatting of the value, or `None` if no format was specified. Both `conversion` and `format_spec` can be set at the same time. This has the same meaning as `FormattedValue.format_spec`.

class ast.**List**(*elts*, *ctx*)

class ast.**Tuple**(*elts*, *ctx*)

리스트나 튜플. `elts`는 요소를 나타내는 노드의 리스트를 보유합니다. `ctx`는 컨테이너가 대입 대상이면 (가령 `(x,y)=something`) *Store*이고, 그렇지 않으면 *Load*입니다.

```
>>> print(ast.dump(ast.parse('[1, 2, 3]', mode='eval'), indent=4))
Expression(
    body=List(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
>>> print(ast.dump(ast.parse('(1, 2, 3)', mode='eval'), indent=4))
Expression(
    body=Tuple(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
```

class ast.**Set**(*elts*)

집합. `elts`는 집합의 요소를 나타내는 노드의 리스트를 보유합니다.

```
>>> print(ast.dump(ast.parse('{1, 2, 3}', mode='eval'), indent=4))
Expression(
    body=Set(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)]))
```

class ast.**Dict**(*keys*, *values*)

딕셔너리. `keys`와 `values`는 각각 키와 값을 나타내는 노드의 리스트를 일치하는 순서대로 (`dictionary.keys()`와 `dictionary.values()`를 호출할 때 반환되는 순서) 보유합니다.

딕셔너리 리터럴을 사용하여 딕셔너리 언 패킹을 수행할 때 확장될 표현식은 `values` 리스트로 가고, `keys`의 해당 위치에는 `None`이 갑니다.

```
>>> print(ast.dump(ast.parse('{"a":1, **d}', mode='eval'), indent=4))
Expression(
    body=Dict(
        keys=[
            Constant(value='a'),
            None],
        values=[
            Constant(value=1),
            Name(id='d', ctx=Load())]))
```

변수

class ast.**Name**(*id*, *ctx*)

변수 이름. `id`는 이름을 문자열로 보유하며, `ctx`는 다음 형 중 하나입니다.

class ast.**Load**

class ast.**Store**

class ast.**Del**

변수 참조는 변숫값을 로드하거나, 그것에 새 값을 대입하거나, 그것을 삭제하는데 사용될 수 있습니다. 변수 참조에는 이러한 경우를 구별하기 위한 컨텍스트가 제공됩니다.

```
>>> print(ast.dump(ast.parse('a'), indent=4))
Module(
    body=[
        Expr(
            value=Name(id='a', ctx=Load()))])

>>> print(ast.dump(ast.parse('a = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store())],
            value=Constant(value=1))])

>>> print(ast.dump(ast.parse('del a'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='a', ctx=Del())])])
```

class ast.**Starred**(*value*, *ctx*)

`*var` 변수 참조. `value`는 변수(일반적으로 *Name* 노드)를 보유합니다. 이 형은 `*args`로 *Call* 노드를 빌드할 때 사용해야 합니다.

```
>>> print(ast.dump(ast.parse('a, *b = it'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
                        Starred(
                            value=Name(id='b', ctx=Store()),
                            ctx=Store())],
                    ctx=Store())],
            value=Name(id='it', ctx=Load()))])
```

표현식

class ast.**Expr**(*value*)

표현식(가령 함수 호출)이 반환 값이 사용되거나 저장되지 않은 자신만의 문장으로 나타나면, 이 컨테이너에 래핑 됩니다. `value`는 이 섹션의 다른 노드인 *Constant*, *Name*, *Lambda*, *Yield* 또는 *YieldFrom* 노드 중 하나를 보유합니다.

```
>>> print(ast.dump(ast.parse('-a'), indent=4))
Module(
    body=[
        Expr(
            value=UnaryOp(
                op=USub(),
                operand=Name(id='a', ctx=Load()))))])
```

class ast.**UnaryOp**(*op*, *operand*)

단항 연산. op는 연산자이고, operand는 임의의 표현식 노드입니다.

class ast.**UAdd**

class ast.**USub**

class ast.**Not**

class ast.**Invert**

단항 연산자 토큰. *Not*은 not 키워드이고, *Invert*는 ~ 연산자입니다.

```
>>> print(ast.dump(ast.parse('not x', mode='eval'), indent=4))
Expression(
    body=UnaryOp(
        op=Not(),
        operand=Name(id='x', ctx=Load())))
```

class ast.**BinOp**(*left*, *op*, *right*)

이항 연산 (더하기나 나누기 같은). op는 연산자이고, left와 right는 임의의 표현식 노드입니다.

```
>>> print(ast.dump(ast.parse('x + y', mode='eval'), indent=4))
Expression(
    body=BinOp(
        left=Name(id='x', ctx=Load()),
        op=Add(),
        right=Name(id='y', ctx=Load())))
```

class ast.**Add**

class ast.**Sub**

class ast.**Mult**

class ast.**Div**

class ast.**FloorDiv**

class ast.**Mod**

class ast.**Pow**

class ast.**LShift**

class ast.**RShift**

class ast.**BitOr**

class ast.**BitXor**

class ast.**BitAnd**

class ast.**MatMult**

이항 연산자 토큰.

class ast.**BoolOp**(*op*, *values*)

불리언 연산, 'or' 나 'and' . op는 *Or*나 *And*입니다. values는 관련된 값입니다. 같은 연산자를 사용하는 연속 연산(가령 a or b or c)은 여러 값을 가진 하나의 노드로 축소됩니다.

여기에는 *UnaryOp*인 not이 포함되지 않습니다.

```
>>> print(ast.dump(ast.parse('x or y', mode='eval'), indent=4))
Expression(
    body=BoolOp(
        op=Or(),
        values=[
            Name(id='x', ctx=Load()),
            Name(id='y', ctx=Load())]))
```

class ast.**And**

class ast.**Or**

불리언 연산자 토큰.

class ast.**Compare**(*left*, *ops*, *comparators*)

둘 이상의 값의 비교. `left`는 비교의 첫 번째 값이고, `ops`는 연산자의 리스트이며, `comparators`는 비교의 첫 번째 요소 다음의 값 리스트입니다.

```
>>> print(ast.dump(ast.parse('1 <= a < 10', mode='eval'), indent=4))
Expression(
    body=Compare(
        left=Constant(value=1),
        ops=[
            LtE(),
            Lt()],
        comparators=[
            Name(id='a', ctx=Load()),
            Constant(value=10)]))
```

class ast.**Eq**

class ast.**NotEq**

class ast.**Lt**

class ast.**LtE**

class ast.**Gt**

class ast.**GtE**

class ast.**Is**

class ast.**IsNot**

class ast.**In**

class ast.**NotIn**

비교 연산자 토큰.

class ast.**Call**(*func*, *args*, *keywords*)

함수 호출. `func`는 함수이며, 종종 *Name*이나 *Attribute* 객체입니다. 인자 중:

- `args`는 위치로 전달된 인자의 리스트를 보유합니다.
- `keywords` holds a list of *keyword* objects representing arguments passed by keyword.

The `args` and `keywords` arguments are optional and default to empty lists.

```
>>> print(ast.dump(ast.parse('func(a, b=c, *d, **e)', mode='eval'), indent=4))
Expression(
    body=Call(
        func=Name(id='func', ctx=Load()),
        args=[
            Name(id='a', ctx=Load()),
            Starred(
                value=Name(id='d', ctx=Load()),
                ctx=Load())],
        keywords=[
            keyword(
                arg='b',
                value=Name(id='c', ctx=Load())),
            keyword(
                value=Name(id='e', ctx=Load()))]))
```

class ast.**keyword**(*arg*, *value*)

함수 호출이나 클래스 정의에 대한 키워드 인자. `arg`는 매개 변수 이름의 원시 문자열이고, `value`는 전달할 노드입니다.

class ast.**IfExp**(*test*, *body*, *orelse*)

a if b else c와 같은 표현식. 각 필드는 단일 노드를 보유해서, 다음 예에서, 세 개 모두 *Name* 노드입니다.

```
>>> print(ast.dump(ast.parse('a if b else c', mode='eval'), indent=4))
Expression(
    body=IfExp(
        test=Name(id='b', ctx=Load()),
        body=Name(id='a', ctx=Load()),
        orelse=Name(id='c', ctx=Load())))
```

class ast.**Attribute**(*value*, *attr*, *ctx*)

어트리뷰트 액세스, 예를 들어 d.keys. value는 노드(보통 *Name*)입니다. attr은 어트리뷰트의 이름을 제공하는 문자열이며, ctx는 어트리뷰트에 적용되는 방식에 따라 *Load*, *Store* 또는 *Del*입니다.

```
>>> print(ast.dump(ast.parse('snake.colour', mode='eval'), indent=4))
Expression(
    body=Attribute(
        value=Name(id='snake', ctx=Load()),
        attr='colour',
        ctx=Load()))
```

class ast.**NamedExpr**(*target*, *value*)

명명된 표현식. 이 AST 노드는 대입 표현식 연산자(바다코끼리(walrus) 연산자라고도 합니다)에 의해 생성됩니다. 첫 번째 인자가 여러 노드일 수 있는 *Assign* 노드와 달리, 이 경우에는 target과 value는 모두 단일 노드여야 합니다.

```
>>> print(ast.dump(ast.parse('(x := 4)', mode='eval'), indent=4))
Expression(
    body=NamedExpr(
        target=Name(id='x', ctx=Store()),
        value=Constant(value=4)))
```

Added in version 3.8.

서브스크립팅

class ast.**Subscript**(*value*, *slice*, *ctx*)

서브스크립트, 가령 l[1]. value는 서브스크립트되는 객체입니다 (보통 시퀀스나 매핑). slice는 인덱스, 슬라이스 또는 키입니다. *Tuple*일 수 있으며 *Slice*를 포함합니다. ctx는 서브스크립트로 수행되는 동작에 따라 *Load*, *Store* 또는 *Del*입니다.

```
>>> print(ast.dump(ast.parse('l[1:2, 3]', mode='eval'), indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Tuple(
            elts=[
                Slice(
                    lower=Constant(value=1),
                    upper=Constant(value=2)),
                Constant(value=3)],
            ctx=Load()),
        ctx=Load()))
```

class ast.**Slice**(*lower*, *upper*, *step*)

일반 슬라이싱 (lower:upper나 lower:upper:step 형식). *Subscript*의 *slice* 필드 내에서만 직접 또는 *Tuple*의 요소로 등장할 수 있습니다.

```
>>> print(ast.dump(ast.parse('l[1:2]', mode='eval'), indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Slice(
            lower=Constant(value=1),
            upper=Constant(value=2)),
        ctx=Load()))
```

컴프리헨션

class ast.**ListComp**(*elt*, *generators*)

class ast.**SetComp**(*elt*, *generators*)

class ast.**GeneratorExp**(*elt*, *generators*)

class ast.**DictComp**(*key*, *value*, *generators*)

리스트와 집합 컴프리헨션, 제너레이터 표현식 및 딕셔너리 컴프리헨션. `elt`(또는 `key`와 `value`)는 항목마다 평가될 부분을 나타내는 단일 노드입니다.

`generators`는 *comprehension* 노드의 리스트입니다.

```
>>> print(ast.dump(
...     ast.parse('[x for x in numbers]', mode='eval'),
...     indent=4,
... ))
Expression(
    body=ListComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
>>> print(ast.dump(
...     ast.parse('{x: x**2 for x in numbers}', mode='eval'),
...     indent=4,
... ))
Expression(
    body=DictComp(
        key=Name(id='x', ctx=Load()),
        value=BinOp(
            left=Name(id='x', ctx=Load()),
            op=Pow(),
            right=Constant(value=2)),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
>>> print(ast.dump(
...     ast.parse('{x for x in numbers}', mode='eval'),
...     indent=4,
... ))
Expression(
    body=SetComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                is_async=0)]))
```

class ast.**comprehension**(*target*, *iter*, *ifs*, *is_async*)

컴프리헨션에서 하나의 `for` 절. `target`은 각 요소(보통 *Name*이나 *Tuple* 노드)에 사용할 참조입니다. `iter`는 이터레이트 할 객체입니다. `ifs`는 테스트 표현식의 리스트입니다: 각 `for` 절은 여러 `ifs`를 가질 수 있습니다.

`is_async`는 컴프리헨션이 비동기임을 나타냅니다(`for` 대신 `async for`를 사용합니다). 값은 정수(0이나 1)입니다.

```
>>> print(ast.dump(ast.parse('[ord(c) for line in file for c in line]', mode=
↪'eval'),
...                indent=4)) # Multiple comprehensions in one.
Expression(
    body=ListComp(
        elt=Call(
            func=Name(id='ord', ctx=Load()),
            args=[
                Name(id='c', ctx=Load())]),
        generators=[
            comprehension(
                target=Name(id='line', ctx=Store()),
                iter=Name(id='file', ctx=Load()),
                is_async=0),
            comprehension(
                target=Name(id='c', ctx=Store()),
                iter=Name(id='line', ctx=Load()),
                is_async=0)]))

>>> print(ast.dump(ast.parse('(n**2 for n in it if n>5 if n<10)', mode='eval'),
...                indent=4)) # generator comprehension
Expression(
    body=GeneratorExp(
        elt=BinOp(
            left=Name(id='n', ctx=Load()),
            op=Pow(),
            right=Constant(value=2)),
        generators=[
            comprehension(
                target=Name(id='n', ctx=Store()),
                iter=Name(id='it', ctx=Load()),
                ifs=[
                    Compare(
                        left=Name(id='n', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=5)]),
                    Compare(
                        left=Name(id='n', ctx=Load()),
                        ops=[
                            Lt()],
                        comparators=[
                            Constant(value=10)])],
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                is_async=0)]))

>>> print(ast.dump(ast.parse('[i async for i in soc]', mode='eval'),
...                indent=4)) # Async comprehension
Expression(
    body=ListComp(
        elt=Name(id='i', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='i', ctx=Store()),
                iter=Name(id='soc', ctx=Load()),
                is_async=1)]))
```

문장

class ast.**Assign**(*targets*, *value*, *type_comment*)

대입. `targets`는 노드의 리스트이고, `value`는 단일 노드입니다.

`targets`의 여러 노드는 각각 같은 값을 할당하는 것을 나타냅니다. 언 패킹은 `targets` 내에 *Tuple*이나 *List*를 넣어 표현됩니다.

type_comment

`type_comment`는 형 어노테이션이 주석으로 포함된 선택적 문자열입니다.

```
>>> print(ast.dump(ast.parse('a = b = 1'), indent=4)) # Multiple assignment
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store()),
                Name(id='b', ctx=Store())],
            value=Constant(value=1))])

>>> print(ast.dump(ast.parse('a,b = c'), indent=4)) # Unpacking
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
                        Name(id='b', ctx=Store())],
                    ctx=Store())],
            value=Name(id='c', ctx=Load()))])
```

class ast.**AnnAssign**(*target*, *annotation*, *value*, *simple*)

An assignment with a type annotation. `target` is a single node and can be a *Name*, an *Attribute* or a *Subscript*. `annotation` is the annotation, such as a *Constant* or *Name* node. `value` is a single optional node.

`simple` is always either 0 (indicating a "complex" target) or 1 (indicating a "simple" target). A "simple" target consists solely of a *Name* node that does not appear between parentheses; all other targets are considered complex. Only simple targets appear in the `__annotations__` dictionary of modules and classes.

```
>>> print(ast.dump(ast.parse('c: int'), indent=4))
Module(
    body=[
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        AnnAssign(
            target=Name(id='c', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=1)])

>>> print(ast.dump(ast.parse('(a): int = 1'), indent=4)) # Annotation with␣
↪parenthesis
Module(
    body=[
        AnnAssign(
            target=Name(id='a', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            value=Constant(value=1),
            simple=0)])

>>> print(ast.dump(ast.parse('a.b: int'), indent=4)) # Attribute annotation
Module(
    body=[
        AnnAssign(
            target=Attribute(
                value=Name(id='a', ctx=Load()),
                attr='b',
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)])

>>> print(ast.dump(ast.parse('a[1]: int'), indent=4)) # Subscript annotation
Module(
    body=[
        AnnAssign(
            target=Subscript(
                value=Name(id='a', ctx=Load()),
                slice=Constant(value=1),
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)])
```

class ast.**AugAssign**(*target*, *op*, *value*)

증분 대입, 가령 `a += 1`. 다음 예에서, `target`은 `x`(*Store* 컨텍스트로)를 위한 *Name* 노드이고, `op`는 *Add*이며, `value`는 값이 1인 *Constant*입니다.

The `target` attribute cannot be of class *Tuple* or *List*, unlike the targets of *Assign*.

```
>>> print(ast.dump(ast.parse('x += 2'), indent=4))
Module(
    body=[
        AugAssign(
            target=Name(id='x', ctx=Store()),
            op=Add(),
            value=Constant(value=2))])
```

class ast.**Raise**(*exc*, *cause*)

`raise` 문. `exc`는 발생시킬 예외 객체로 일반적으로 *Call*이나 *Name*이거나, 독립 `raise`의 경우 `None` 입니다. `cause`는 `raise x from y`에서 `y`에 해당하는 선택적 부분입니다.

```
>>> print(ast.dump(ast.parse('raise x from y'), indent=4))
Module(
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    body=[
        Raise(
            exc=Name(id='x', ctx=Load()),
            cause=Name(id='y', ctx=Load()))])
```

class ast.**Assert**(*test*, *msg*)

어서션. `test`는 (*Compare* 노드와 같은) 조건을 보유합니다. `msg`는 실패 메시지를 보유합니다.

```
>>> print(ast.dump(ast.parse('assert x,y'), indent=4))
Module(
    body=[
        Assert(
            test=Name(id='x', ctx=Load()),
            msg=Name(id='y', ctx=Load()))])
```

class ast.**Delete**(*targets*)

`del` 문을 나타냅니다. `targets`는 *Name*, *Attribute* 또는 *Subscript* 같은 노드들의 리스트입니다.

```
>>> print(ast.dump(ast.parse('del x,y,z'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='x', ctx=Del()),
                Name(id='y', ctx=Del()),
                Name(id='z', ctx=Del())])])
```

class ast.**Pass**

`pass` 문.

```
>>> print(ast.dump(ast.parse('pass'), indent=4))
Module(
    body=[
        Pass()])
```

class ast.**TypeAlias**(*name*, *type_params*, *value*)

A *type alias* created through the `type` statement. `name` is the name of the alias, `type_params` is a list of *type parameters*, and `value` is the value of the type alias.

```
>>> print(ast.dump(ast.parse('type Alias = int'), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            value=Name(id='int', ctx=Load()))])
```

Added in version 3.12.

함수나 루프 내부에만 적용할 수 있는 다른 문장들은 다른 섹션에 설명되어 있습니다.

임포트

class ast.**Import**(*names*)

import 문. `names`는 *alias* 노드의 리스트입니다.

```
>>> print(ast.dump(ast.parse('import x,y,z'), indent=4))
Module(
    body=[
        Import(
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')])])
```

class ast.**ImportFrom**(*module*, *names*, *level*)

`from x import y`를 나타냅니다. `module`은 선행 점이 없는 'from' 이름의 원시 문자열이며, `from . import foo`와 같은 문장의 경우 `None`입니다. `level`은 상대 임포트 수준을 보유하는 정수입니다 (0은 절대 임포트를 의미합니다).

```
>>> print(ast.dump(ast.parse('from y import x,y,z'), indent=4))
Module(
    body=[
        ImportFrom(
            module='y',
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')],
            level=0)])
```

class ast.**alias**(*name*, *asname*)

두 매개 변수 모두 이름의 원시 문자열입니다. 정규 이름을 사용하면 `asname`은 `None`이 될 수 있습니다.

```
>>> print(ast.dump(ast.parse('from ..foo.bar import a as b, c'), indent=4))
Module(
    body=[
        ImportFrom(
            module='foo.bar',
            names=[
                alias(name='a', asname='b'),
                alias(name='c')],
            level=2)])
```

제어 흐름

> **참고**
>
> `else`와 같은 선택적 절은 존재하지 않으면 빈 목록으로 저장됩니다.

class ast.**If**(*test*, *body*, *orelse*)

`if` 문. `test`는 (*Compare* 노드와 같은) 단일 노드를 보유합니다. `body`와 `orelse`는 각각 노드 리스트를 보유합니다.

`elif` 절은 AST에서 특별한 표현이 없지만, 앞의 `orelse` 섹션 안에서 추가 *If* 노드로 나타납니다.

```
>>> print(ast.dump(ast.parse("""
... if x:
...     ...
... elif y:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        If(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                If(
                    test=Name(id='y', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))],
                    orelse=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

class ast.**For**(*target, iter, body, orelse, type_comment*)

A `for` loop. `target` holds the variable(s) the loop assigns to, as a single *Name*, *Tuple*, *List*, *Attribute* or *Subscript* node. `iter` holds the item to be looped over, again as a single node. `body` and `orelse` contain lists of nodes to execute. Those in `orelse` are executed if the loop finishes normally, rather than via a `break` statement.

type_comment

`type_comment`는 형 어노테이션이 주석으로 포함된 선택적 문자열입니다.

```
>>> print(ast.dump(ast.parse("""
... for x in y:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='x', ctx=Store()),
            iter=Name(id='y', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**While**(*test, body, orelse*)

`while` 루프. `test`는 (*Compare* 노드와 같은) 조건을 보유합니다.

```
>>> print(ast.dump(ast.parse("""
... while x:
...     ...
... else:
...     ...
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
... """), indent=4))
Module(
    body=[
        While(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**Break**

class ast.**Continue**

break와 continue 문.

```
>>> print(ast.dump(ast.parse("""\
... for a in b:
...     if a > 5:
...         break
...     else:
...         continue
...
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='a', ctx=Store()),
            iter=Name(id='b', ctx=Load()),
            body=[
                If(
                    test=Compare(
                        left=Name(id='a', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=5)]),
                    body=[
                        Break()],
                    orelse=[
                        Continue()])])])
```

class ast.**Try**(*body*, *handlers*, *orelse*, *finalbody*)

try 블록. *ExceptHandler* 노드의 리스트인 handlers를 제외한, 모든 어트리뷰트는 실행할 노드의 리스트입니다.

```
>>> print(ast.dump(ast.parse("""
... try:
...     ...
... except Exception:
...     ...
... except OtherException as e:
...     ...
... else:
...     ...
... finally:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...     ...
... """), indent=4))
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            handlers=[
                ExceptHandler(
                    type=Name(id='Exception', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                ExceptHandler(
                    type=Name(id='OtherException', ctx=Load()),
                    name='e',
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))],
            finalbody=[
                Expr(
                    value=Constant(value=Ellipsis))])])
```

class ast.**TryStar**(*body*, *handlers*, *orelse*, *finalbody*)

`try` blocks which are followed by `except*` clauses. The attributes are the same as for *Try* but the *ExceptHandler* nodes in `handlers` are interpreted as `except*` blocks rather then `except`.

```
>>> print(ast.dump(ast.parse("""
... try:
...     ...
... except* Exception:
...     ...
... """), indent=4))
Module(
    body=[
        TryStar(
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            handlers=[
                ExceptHandler(
                    type=Name(id='Exception', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.11.

class ast.**ExceptHandler**(*type*, *name*, *body*)

단일 `except` 절. `type`은 일치할 예외 형이며, 일반적으로 *Name* 노드(또는 모두 잡는 `except:` 절의 경우는 `None`)입니다. `name`은 예외를 담을 이름을 위한 원시 문자열이거나, 절에 `as foo`가 없으면 `None`입니다. `body`는 노드의 리스트입니다.

```
>>> print(ast.dump(ast.parse("""\
... try:
...     a + 1
... except TypeError:
...     pass
... """), indent=4))
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=BinOp(
                        left=Name(id='a', ctx=Load()),
                        op=Add(),
                        right=Constant(value=1)))],
            handlers=[
                ExceptHandler(
                    type=Name(id='TypeError', ctx=Load()),
                    body=[
                        Pass()])])])
```

class ast.**With**(*items*, *body*, *type_comment*)

with 블록. items는 컨텍스트 관리자를 나타내는 *withitem* 노드의 리스트이며, body는 컨텍스트 내에서 들여쓰기 된 블록입니다.

type_comment

type_comment는 형 어노테이션이 주석으로 포함된 선택적 문자열입니다.

class ast.**withitem**(*context_expr*, *optional_vars*)

with 블록의 단일 컨텍스트 관리자. context_expr은 컨텍스트 관리자이며, 종종 *Call* 노드입니다. optional_vars는 as foo 부분의 경우 *Name*, *Tuple* 또는 *List*이거나, 사용하지 않으면 None입니다.

```
>>> print(ast.dump(ast.parse("""\
... with a as b, c as d:
...     something(b, d)
... """), indent=4))
Module(
    body=[
        With(
            items=[
                withitem(
                    context_expr=Name(id='a', ctx=Load()),
                    optional_vars=Name(id='b', ctx=Store())),
                withitem(
                    context_expr=Name(id='c', ctx=Load()),
                    optional_vars=Name(id='d', ctx=Store()))],
            body=[
                Expr(
                    value=Call(
                        func=Name(id='something', ctx=Load()),
                        args=[
                            Name(id='b', ctx=Load()),
                            Name(id='d', ctx=Load())]))])])
```

Pattern matching

class ast.**Match**(*subject*, *cases*)

A `match` statement. `subject` holds the subject of the match (the object that is being matched against the cases) and `cases` contains an iterable of *match_case* nodes with the different cases.

Added in version 3.10.

class ast.**match_case**(*pattern*, *guard*, *body*)

A single case pattern in a `match` statement. `pattern` contains the match pattern that the subject will be matched against. Note that the *AST* nodes produced for patterns differ from those produced for expressions, even when they share the same syntax.

The `guard` attribute contains an expression that will be evaluated if the pattern matches the subject.

`body` contains a list of nodes to execute if the pattern matches and the result of evaluating the guard expression is true.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] if x>0:
...         ...
...     case tuple():
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchAs(name='x')]),
                    guard=Compare(
                        left=Name(id='x', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=0)]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='tuple', ctx=Load())),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchValue**(*value*)

A match literal or value pattern that compares by equality. `value` is an expression node. Permitted value nodes are restricted as described in the match statement documentation. This pattern succeeds if the match subject is equal to the evaluated value.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case "Relevant":
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchValue(
                        value=Constant(value='Relevant')),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchSingleton**(*value*)

A match literal pattern that compares by identity. `value` is the singleton to be compared against: `None`, `True`, or `False`. This pattern succeeds if the match subject is the given constant.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case None:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSingleton(value=None),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchSequence**(*patterns*)

A match sequence pattern. `patterns` contains the patterns to be matched against the subject elements if the subject is a sequence. Matches a variable length sequence if one of the subpatterns is a `MatchStar` node, otherwise matches a fixed length sequence.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [1, 2]:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchValue(
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                                value=Constant(value=1)),
                            MatchValue(
                                value=Constant(value=2))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchStar**(*name*)

Matches the rest of the sequence in a variable length match sequence pattern. If `name` is not `None`, a list containing the remaining sequence elements is bound to that name if the overall sequence pattern is successful.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [1, 2, *rest]:
...         ...
...     case [*_]:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchValue(
                                value=Constant(value=1)),
                            MatchValue(
                                value=Constant(value=2)),
                            MatchStar(name='rest')]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchSequence(
                        patterns=[
                            MatchStar()]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchMapping**(*keys*, *patterns*, *rest*)

A match mapping pattern. `keys` is a sequence of expression nodes. `patterns` is a corresponding sequence of pattern nodes. `rest` is an optional name that can be specified to capture the remaining mapping elements. Permitted key expressions are restricted as described in the match statement documentation.

This pattern succeeds if the subject is a mapping, all evaluated key expressions are present in the mapping, and the value corresponding to each key matches the corresponding subpattern. If `rest` is not `None`, a dict containing the remaining mapping elements is bound to that name if the overall mapping pattern is successful.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case {1: _, 2: _}:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
...         ...
...     case {**rest}:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchMapping(
                        keys=[
                            Constant(value=1),
                            Constant(value=2)],
                        patterns=[
                            MatchAs(),
                            MatchAs()]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchMapping(rest='rest'),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchClass**(*cls*, *patterns*, *kwd_attrs*, *kwd_patterns*)

A match class pattern. `cls` is an expression giving the nominal class to be matched. `patterns` is a sequence of pattern nodes to be matched against the class defined sequence of pattern matching attributes. `kwd_attrs` is a sequence of additional attributes to be matched (specified as keyword arguments in the class pattern), `kwd_patterns` are the corresponding patterns (specified as keyword values in the class pattern).

This pattern succeeds if the subject is an instance of the nominated class, all positional patterns match the corresponding class-defined attributes, and any specified keyword attributes match their corresponding pattern.

Note: classes may define a property that returns self in order to match a pattern node against the instance being matched. Several builtin types are also matched that way, as described in the match statement documentation.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case Point2D(0, 0):
...         ...
...     case Point3D(x=0, y=0, z=0):
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point2D', ctx=Load()),
                        patterns=[
                            MatchValue(
                                value=Constant(value=0)),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                            MatchValue(
                                value=Constant(value=0))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchClass(
                        cls=Name(id='Point3D', ctx=Load()),
                        kwd_attrs=[
                            'x',
                            'y',
                            'z'],
                        kwd_patterns=[
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0)),
                            MatchValue(
                                value=Constant(value=0))]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])
```

Added in version 3.10.

class ast.**MatchAs**(*pattern*, *name*)

A match "as-pattern", capture pattern or wildcard pattern. `pattern` contains the match pattern that the subject will be matched against. If the pattern is `None`, the node represents a capture pattern (i.e a bare name) and will always succeed.

The `name` attribute contains the name that will be bound if the pattern is successful. If `name` is `None`, `pattern` must also be `None` and the node represents the wildcard pattern.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] as y:
...         ...
...     case _:
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchAs(
                        pattern=MatchSequence(
                            patterns=[
                                MatchAs(name='x')]),
                        name='y'),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                match_case(
                    pattern=MatchAs(),
                    body=[
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                        Expr(
                            value=Constant(value=Ellipsis))])])])])
```

Added in version 3.10.

class ast.**MatchOr**(*patterns*)

A match "or-pattern". An or-pattern matches each of its subpatterns in turn to the subject, until one succeeds. The or-pattern is then deemed to succeed. If none of the subpatterns succeed the or-pattern fails. The `patterns` attribute contains a list of match pattern nodes that will be matched against the subject.

```
>>> print(ast.dump(ast.parse("""
... match x:
...     case [x] | (y):
...         ...
... """), indent=4))
Module(
    body=[
        Match(
            subject=Name(id='x', ctx=Load()),
            cases=[
                match_case(
                    pattern=MatchOr(
                        patterns=[
                            MatchSequence(
                                patterns=[
                                    MatchAs(name='x')]),
                            MatchAs(name='y')]),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])])])])
```

Added in version 3.10.

Type annotations

class ast.**TypeIgnore**(*lineno*, *tag*)

A `# type: ignore` comment located at *lineno*. *tag* is the optional tag specified by the form `# type: ignore <tag>`.

```
>>> print(ast.dump(ast.parse('x = 1 # type: ignore', type_comments=True),␣
↪indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='x', ctx=Store())],
            value=Constant(value=1))],
    type_ignores=[
        TypeIgnore(lineno=1, tag='')])
>>> print(ast.dump(ast.parse('x: bool = 1 # type: ignore[assignment]', type_
↪comments=True), indent=4))
Module(
    body=[
        AnnAssign(
            target=Name(id='x', ctx=Store()),
            annotation=Name(id='bool', ctx=Load()),
            value=Constant(value=1),
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
            simple=1)],
    type_ignores=[
        TypeIgnore(lineno=1, tag='[assignment]')])
```

> **참고**
>
> `TypeIgnore` nodes are not generated when the *type_comments* parameter is set to `False` (default). See *ast.parse()* for more details.

Added in version 3.8.

Type parameters

Type parameters can exist on classes, functions, and type aliases.

class `ast.`**`TypeVar`**(*name*, *bound*, *default_value*)

A *typing.TypeVar*. `name` is the name of the type variable. `bound` is the bound or constraints, if any. If `bound` is a *Tuple*, it represents constraints; otherwise it represents the bound. `default_value` is the default value; if the `TypeVar` has no default, this attribute will be set to `None`.

```
>>> print(ast.dump(ast.parse("type Alias[T: int = bool] = list[T]"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                TypeVar(
                    name='T',
                    bound=Name(id='int', ctx=Load()),
                    default_value=Name(id='bool', ctx=Load()))],
            value=Subscript(
                value=Name(id='list', ctx=Load()),
                slice=Name(id='T', ctx=Load()),
                ctx=Load()))])
```

Added in version 3.12.

버전 3.13에서 변경: Added the *default_value* parameter.

class `ast.`**`ParamSpec`**(*name*, *default_value*)

A *typing.ParamSpec*. `name` is the name of the parameter specification. `default_value` is the default value; if the `ParamSpec` has no default, this attribute will be set to `None`.

```
>>> print(ast.dump(ast.parse("type Alias[**P = [int, str]] = Callable[P, int]
↪"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                ParamSpec(
                    name='P',
                    default_value=List(
                        elts=[
                            Name(id='int', ctx=Load()),
                            Name(id='str', ctx=Load())],
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                        ctx=Load())),
            value=Subscript(
                value=Name(id='Callable', ctx=Load()),
                slice=Tuple(
                    elts=[
                        Name(id='P', ctx=Load()),
                        Name(id='int', ctx=Load())],
                    ctx=Load()),
                ctx=Load()))])
```

Added in version 3.12.

버전 3.13에서 변경: Added the *default_value* parameter.

class ast.**TypeVarTuple**(*name*, *default_value*)

A *typing.TypeVarTuple*. `name` is the name of the type variable tuple. `default_value` is the default value; if the `TypeVarTuple` has no default, this attribute will be set to `None`.

```
>>> print(ast.dump(ast.parse("type Alias[*Ts = ()] = tuple[*Ts]"), indent=4))
Module(
    body=[
        TypeAlias(
            name=Name(id='Alias', ctx=Store()),
            type_params=[
                TypeVarTuple(
                    name='Ts',
                    default_value=Tuple(ctx=Load()))],
            value=Subscript(
                value=Name(id='tuple', ctx=Load()),
                slice=Tuple(
                    elts=[
                        Starred(
                            value=Name(id='Ts', ctx=Load()),
                            ctx=Load())],
                    ctx=Load()),
                ctx=Load()))])
```

Added in version 3.12.

버전 3.13에서 변경: Added the *default_value* parameter.

함수와 클래스 정의

class ast.**FunctionDef**(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*, *type_params*)

함수 정의.

- `name`은 함수 이름의 원시 문자열입니다.
- `args` is an *arguments* node.
- `body`는 함수 내부의 노드 리스트입니다.
- `decorator_list`는 적용할 데코레이터 리스트이며, 가장 바깥쪽에 먼저 저장됩니다 (즉, 리스트의 첫 번째가 마지막에 적용됩니다).
- `returns`는 반환 어노테이션입니다.
- `type_params` is a list of *type parameters*.

type_comment

`type_comment`는 형 어노테이션이 주석으로 포함된 선택적 문자열입니다.

버전 3.12에서 변경: Added `type_params`.

class `ast.`**`Lambda`**(*args*, *body*)

`lambda`는 표현식 내에서 사용할 수 있는 최소 함수 정의입니다. *`FunctionDef`*와 달리, `body`는 단일 노드를 보유합니다.

```
>>> print(ast.dump(ast.parse('lambda x,y: ...'), indent=4))
Module(
    body=[
        Expr(
            value=Lambda(
                args=arguments(
                    args=[
                        arg(arg='x'),
                        arg(arg='y')]),
                body=Constant(value=Ellipsis)))])
```

class `ast.`**`arguments`**(*posonlyargs*, *args*, *vararg*, *kwonlyargs*, *kw_defaults*, *kwarg*, *defaults*)

함수의 인자.

- `posonlyargs`, `args` 및 `kwonlyargs`는 *`arg`* 노드의 리스트입니다.
- `vararg`와 `kwarg`는 `*args`, `**kwargs` 매개 변수를 참조하는 단일 *`arg`* 노드입니다.
- `kw_defaults`는 키워드 전용 인자의 기본값 리스트입니다. 어떤 것이 `None`이면, 해당 인자는 필수입니다.
- `defaults`는 위치적으로 전달될 수 있는 인자의 기본값 리스트입니다. 기본값 수가 더 적으면, 마지막 n개의 인자에 해당합니다.

class `ast.`**`arg`**(*arg*, *annotation*, *type_comment*)

A single argument in a list. `arg` is a raw string of the argument name; `annotation` is its annotation, such as a *`Name`* node.

`type_comment`

`type_comment`는 주석으로 제공된 형 어노테이션이 있는 선택적 문자열입니다.

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... def f(a: 'annotation', b=1, c=2, *d, e, f=3, **g) -> 'return annotation':
...     pass
... """), indent=4))
Module(
    body=[
        FunctionDef(
            name='f',
            args=arguments(
                args=[
                    arg(
                        arg='a',
                        annotation=Constant(value='annotation')),
                    arg(arg='b'),
                    arg(arg='c')],
                vararg=arg(arg='d'),
                kwonlyargs=[
                    arg(arg='e'),
                    arg(arg='f')],
                kw_defaults=[
                    None,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
                    Constant(value=3)],
                kwarg=arg(arg='g'),
                defaults=[
                    Constant(value=1),
                    Constant(value=2)]),
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())],
            returns=Constant(value='return annotation'))])
```

class ast.**Return**(*value*)

return 문.

```
>>> print(ast.dump(ast.parse('return 4'), indent=4))
Module(
    body=[
        Return(
            value=Constant(value=4))])
```

class ast.**Yield**(*value*)

class ast.**YieldFrom**(*value*)

A yield or yield from expression. Because these are expressions, they must be wrapped in an *Expr* node if the value sent back is not used.

```
>>> print(ast.dump(ast.parse('yield x'), indent=4))
Module(
    body=[
        Expr(
            value=Yield(
                value=Name(id='x', ctx=Load())))])

>>> print(ast.dump(ast.parse('yield from x'), indent=4))
Module(
    body=[
        Expr(
            value=YieldFrom(
                value=Name(id='x', ctx=Load())))])
```

class ast.**Global**(*names*)

class ast.**Nonlocal**(*names*)

global과 nonlocal 문. names는 원시 문자열 리스트입니다.

```
>>> print(ast.dump(ast.parse('global x,y,z'), indent=4))
Module(
    body=[
        Global(
            names=[
                'x',
                'y',
                'z'])])

>>> print(ast.dump(ast.parse('nonlocal x,y,z'), indent=4))
Module(
    body=[
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
        Nonlocal(
            names=[
                'x',
                'y',
                'z'])])
```

class ast.**ClassDef**(*name*, *bases*, *keywords*, *body*, *decorator_list*, *type_params*)

클래스 정의.

- `name`은 클래스 이름의 원시 문자열입니다.
- `bases`는 명시적으로 지정된 베이스 클래스의 노드 리스트입니다.
- `keywords` is a list of *`keyword`* nodes, principally for 'metaclass'. Other keywords will be passed to the metaclass, as per **PEP 3115**.
- `body`는 클래스 정의 내에서 코드를 나타내는 노드 리스트입니다.
- `decorator_list`는 *`FunctionDef`*에서와 같이 노드 리스트입니다.
- `type_params` is a list of *type parameters*.

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... class Foo(base1, base2, metaclass=meta):
...     pass
... """), indent=4))
Module(
    body=[
        ClassDef(
            name='Foo',
            bases=[
                Name(id='base1', ctx=Load()),
                Name(id='base2', ctx=Load())],
            keywords=[
                keyword(
                    arg='metaclass',
                    value=Name(id='meta', ctx=Load()))],
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())])])
```

버전 3.12에서 변경: Added `type_params`.

Async와 await

class ast.**AsyncFunctionDef**(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*, *type_params*)

`async def` 함수 정의. *`FunctionDef`*와 같은 필드를 갖습니다.

버전 3.12에서 변경: Added `type_params`.

class ast.**Await**(*value*)

`await` 표현식. `value`는 기다릴 대상입니다. *`AsyncFunctionDef`*의 본문에서만 유효합니다.

```
>>> print(ast.dump(ast.parse("""\
... async def f():
...     await other_func()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
... """), indent=4))
Module(
    body=[
        AsyncFunctionDef(
            name='f',
            args=arguments(),
            body=[
                Expr(
                    value=Await(
                        value=Call(
                            func=Name(id='other_func', ctx=Load()))))])])
```

class ast.**AsyncFor**(*target*, *iter*, *body*, *orelse*, *type_comment*)

class ast.**AsyncWith**(*items*, *body*, *type_comment*)

`async for` 루프와 `async with` 컨텍스트 관리자. 이들은 각각 `For`와 `With`와 같은 필드를 갖습니다. `AsyncFunctionDef`의 본문에서만 유효합니다.

> **참고**
>
> When a string is parsed by `ast.parse()`, operator nodes (subclasses of `ast.operator`, `ast.unaryop`, `ast.cmpop`, `ast.boolop` and `ast.expr_context`) on the returned tree will be singletons. Changes to one will be reflected in all other occurrences of the same value (for example, `ast.Add`).

32.1.3 ast helpers

노드 클래스 외에도, `ast` 모듈은 추상 구문 트리를 탐색하기 위해 다음 유틸리티 함수와 클래스를 정의합니다:

ast.**parse**(*source*, *filename='<unknown>'*, *mode='exec'*, **, type_comments=False*, *feature_version=None*, *optimize=-1*)

Parse the source into an AST node. Equivalent to `compile(source, filename, mode, flags=FLAGS_VALUE, optimize=optimize)`, where `FLAGS_VALUE` is `ast.PyCF_ONLY_AST` if `optimize <= 0` and `ast.PyCF_OPTIMIZED_AST` otherwise.

If `type_comments=True` is given, the parser is modified to check and return type comments as specified by **PEP 484** and **PEP 526**. This is equivalent to adding `ast.PyCF_TYPE_COMMENTS` to the flags passed to `compile()`. This will report syntax errors for misplaced type comments. Without this flag, type comments will be ignored, and the `type_comment` field on selected AST nodes will always be `None`. In addition, the locations of `# type: ignore` comments will be returned as the `type_ignores` attribute of `Module` (otherwise it is always an empty list).

또한, `mode`가 `'func_type'`이면, 입력 문법은 **PEP 484** "서명 형 주석"에 따라 수정됩니다, 예를 들어 `(str, int) -> List[str]`.

Setting `feature_version` to a tuple `(major, minor)` will result in a "best-effort" attempt to parse using that Python version' s grammar. For example, setting `feature_version=(3, 9)` will attempt to disallow parsing of `match` statements. Currently `major` must equal to `3`. The lowest supported version is `(3, 7)` (and this may increase in future Python versions); the highest is `sys.version_info[0:2]`. "Best-effort" attempt means there is no guarantee that the parse (or success of the parse) is the same as when run on the Python version corresponding to `feature_version`.

If source contains a null character (`\0`), `ValueError` is raised.

> **경고**
>
> Note that successfully parsing source code into an AST object doesn' t guarantee that the source code provided is valid Python code that can be executed as the compilation step can raise further `SyntaxError`

exceptions. For instance, the source `return 42` generates a valid AST node for a return statement, but it cannot be compiled alone (it needs to be inside a function node).

In particular, `ast.parse()` won' t do any scoping checks, which the compilation step does.

> **경고**
>
> 파이썬 AST 컴파일러의 스택 깊이 제한으로 인해 충분히 크고/복잡한 문자열로 파이썬 인터프리터가 충돌하도록 만들 수 있습니다.

버전 3.8에서 변경: `type_comments`, `mode='func_type'` 및 `feature_version`추가했습니다.

버전 3.13에서 변경: The minimum supported version for `feature_version` is now `(3, 7)`. The `optimize` argument was added.

`ast.`**`unparse`**(*ast_obj*)

`ast.AST` 객체를 역 구문 분석하고 `ast.parse()`로 다시 구문 분석할 경우 동등한 `ast.AST` 객체를 생성하는 코드가 포함된 문자열을 생성합니다.

> **경고**
>
> 생성된 코드 문자열은 `ast.AST` 객체를 생성한 원래 코드와 반드시 같을 필요는 없습니다 (상수 튜플/frozenset과 같은 컴파일러 최적화 없이).

> **경고**
>
> 매우 복잡한 표현식을 역 구분 분석하려고 하면 `RecursionError`가 발생할 수 있습니다.

Added in version 3.9.

`ast.`**`literal_eval`**(*node_or_string*)

Evaluate an expression node or a string containing only a Python literal or container display. The string or node provided may only consist of the following Python literal structures: strings, bytes, numbers, tuples, lists, dicts, sets, booleans, `None` and `Ellipsis`.

This can be used for evaluating strings containing Python values without the need to parse the values oneself. It is not capable of evaluating arbitrarily complex expressions, for example involving operators or indexing.

This function had been documented as "safe" in the past without defining what that meant. That was misleading. This is specifically designed not to execute Python code, unlike the more general `eval()`. There is no namespace, no name lookups, or ability to call out. But it is not free from attack: A relatively small input can lead to memory exhaustion or to C stack exhaustion, crashing the process. There is also the possibility for excessive CPU consumption denial of service on some inputs. Calling it on untrusted data is thus not recommended.

> **경고**
>
> It is possible to crash the Python interpreter due to stack depth limitations in Python' s AST compiler.
>
> It can raise `ValueError`, `TypeError`, `SyntaxError`, `MemoryError` and `RecursionError` depending on the malformed input.

버전 3.2에서 변경: 이제 바이트열과 집합 리터럴을 허용합니다.

버전 3.9에서 변경: 이제 `'set()'`으로 빈 집합을 만드는 것을 지원합니다.

버전 3.10에서 변경: For string inputs, leading spaces and tabs are now stripped.

ast.**get_docstring**(*node*, *clean=True*)

주어진 *node*(`FunctionDef`, `AsyncFunctionDef`, `ClassDef` 또는 `Module` 노드이어야 합니다)의 독스트링이나, 독스트링이 없으면 `None`을 반환합니다. *clean*이 참이면, `inspect.cleandoc()`으로 독스트링의 들여쓰기를 정리합니다.

버전 3.5에서 변경: `AsyncFunctionDef` 가 이제 지원됩니다.

ast.**get_source_segment**(*source*, *node*, **, padded=False*)

Get source code segment of the *source* that generated *node*. If some location information (`lineno`, `end_lineno`, `col_offset`, or `end_col_offset`) is missing, return `None`.

*padded*가 `True`이면, 여러 줄 문장의 첫 번째 줄은 원래 위치와 일치하도록 스페이스로 채워집니다.

Added in version 3.8.

ast.**fix_missing_locations**(*node*)

When you compile a node tree with `compile()`, the compiler expects `lineno` and `col_offset` attributes for every node that supports them. This is rather tedious to fill in for generated nodes, so this helper adds these attributes recursively where not already set, by setting them to the values of the parent node. It works recursively starting at *node*.

ast.**increment_lineno**(*node*, *n=1*)

*node*에서 시작하는 트리에서 각 노드의 줄 번호와 끝 줄 번호를 *n*만큼 증가시킵니다. 파일의 다른 위치로 "코드를 이동" 하는 데 유용합니다.

ast.**copy_location**(*new_node*, *old_node*)

Copy source location (`lineno`, `col_offset`, `end_lineno`, and `end_col_offset`) from *old_node* to *new_node* if possible, and return *new_node*.

ast.**iter_fields**(*node*)

*node*에 존재하는 `node._fields`의 각 필드에 대해 `(fieldname, value)` 튜플을 산출합니다.

ast.**iter_child_nodes**(*node*)

*node*의 모든 직접 자식 노드, 즉 노드인 모든 필드와 노드 리스트인 필드의 모든 항목을 산출합니다.

ast.**walk**(*node*)

*node*로 시작하는 트리(*node* 자체를 포함합니다)의 모든 자손 노드를 지정된 순서 없이 재귀적으로 산출합니다. 이는 노드를 제자리에서 수정하고 문맥을 신경 쓰지 않을 때 유용합니다.

class ast.**NodeVisitor**

추상 구문 트리를 걷고 발견된 모든 노드에 대해 방문자 함수를 호출하는 노드 방문자 베이스 클래스. 이 함수는 `visit()` 메서드에 의해 전달되는 값을 반환할 수 있습니다.

이 클래스는 서브 클래싱하고자 하는 것이며, 서브 클래스는 방문자 메서드를 추가합니다.

visit(*node*)

노드를 방문합니다. 기본 구현은 `self.visit_`*classname*이라는 메서드를 호출하는데, 여기서 *classname* 은 노드 클래스의 이름입니다. 또는 이 메서드가 없으면 `generic_visit()`를 호출합니다.

generic_visit(*node*)

이 방문자는 노드의 자식에 대해 `visit()`를 호출합니다.

방문자가 `generic_visit()`를 호출하거나 직접 방문하지 않는 한, 사용자 정의 방문자 메서드가 있는 노드의 자식 노드는 방문 되지 않음에 유의하십시오.

visit_Constant(*node*)

Handles all constant nodes.

탐색 중에 노드에 변경 사항을 적용하려면 `NodeVisitor`를 사용하지 마십시오. 이를 위해 수정을 허락하는 특수한 방문자(`NodeTransformer`)가 있습니다.

버전 3.8부터 폐지됨: Methods `visit_Num()`, `visit_Str()`, `visit_Bytes()`, `visit_NameConstant()` and `visit_Ellipsis()` are deprecated now and will not be called in future Python versions. Add the *visit_Constant()* method to handle all constant nodes.

class ast.**NodeTransformer**

추상 구문 트리를 걷고 노드 수정을 허락하는 *NodeVisitor* 서브 클래스.

*NodeTransformer*는 AST를 걷고 방문자 메서드의 반환 값을 사용하여 이전 노드를 바꾸거나 제거합니다. 방문자 메서드의 반환 값이 `None`이면, 노드가 그 위치에서 제거되고, 그렇지 않으면 반환 값으로 치환됩니다. 반환 값은 원래 노드일 수 있으며, 이때는 치환이 일어나지 않습니다.

다음은 모든 이름 조회(`foo`)를 `data['foo']`로 다시 쓰는 변환기 예제입니다:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Constant(value=node.id),
            ctx=node.ctx
        )
```

Keep in mind that if the node you' re operating on has child nodes you must either transform the child nodes yourself or call the *generic_visit()* method for the node first.

문장의 컬렉션의 일부인 노드의 경우 (모든 문장 노드에 적용됩니다), 방문자는 단일 노드가 아닌 노드 리스트를 반환 할 수도 있습니다.

If *NodeTransformer* introduces new nodes (that weren' t part of original tree) without giving them location information (such as *lineno*), *fix_missing_locations()* should be called with the new sub-tree to recalculate the location information:

```
tree = ast.parse('foo', mode='eval')
new_tree = fix_missing_locations(RewriteName().visit(tree))
```

일반적으로 다음과 같이 변환기를 사용합니다:

```
node = YourTransformer().visit(node)
```

ast.**dump**(*node*, *annotate_fields=True*, *include_attributes=False*, **, indent=None*, *show_empty=False*)

*node*에서 포맷된 트리 덤프를 반환합니다. 이것은 주로 디버깅 목적으로 유용합니다. *annotate_fields*가 참이면 (기본값), 반환된 문자열에 필드의 이름과 값이 표시됩니다. *annotate_fields*가 거짓이면, 모호하지 않은 필드 이름을 생략하여 결과 문자열이 더 간결해집니다. 줄 번호와 열 오프셋과 같은 어트리뷰트는 기본적으로 덤프 되지 않습니다. 원한다면, *include_attributes*를 참으로 설정할 수 있습니다.

*indent*가 음이 아닌 정수나 문자열이면, 트리는 그 들여쓰기 수준으로 예쁘게 인쇄됩니다. 들여쓰기 수준 0, 음수 또는 `""`는 줄 넘김 만 삽입합니다. `None`(기본값)은 단일 줄 표현을 선택합니다. 양의 정수 indent를 사용하면 수준마다 그만큼 들여쓰기 됩니다. *indent*가 문자열(가령 `"\t"`)이면, 해당 문자열은 각 수준을 들여 쓰는 데 사용됩니다.

If *show_empty* is false (the default), optional empty lists will be omitted from the output. Optional `None` values are always omitted.

버전 3.9에서 변경: *indent* 옵션을 추가했습니다.

버전 3.13에서 변경: Added the *show_empty* option.

```
>>> print(ast.dump(ast.parse("""\
... async def f():
...     await other_func()
... """), indent=4, show_empty=True))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Module(
    body=[
        AsyncFunctionDef(
            name='f',
            args=arguments(
                posonlyargs=[],
                args=[],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
            body=[
                Expr(
                    value=Await(
                        value=Call(
                            func=Name(id='other_func', ctx=Load()),
                            args=[],
                            keywords=[]))))],
            decorator_list=[],
            type_params=[])],
    type_ignores=[])
```

32.1.4 Compiler flags

프로그램 컴파일에 대한 효과를 변경하기 위해 다음 플래그를 `compile()`에 전달할 수 있습니다:

`ast.`**`PyCF_ALLOW_TOP_LEVEL_AWAIT`**

최상위 수준 `await`, `async for`, `async with` 및 비동기 컴프리헨션에 대한 지원을 활성화합니다.

Added in version 3.8.

`ast.`**`PyCF_ONLY_AST`**

컴파일된 코드 객체를 반환하는 대신 추상 구문 트리를 생성하고 반환합니다.

`ast.`**`PyCF_OPTIMIZED_AST`**

The returned AST is optimized according to the *optimize* argument in `compile()` or `ast.parse()`.

Added in version 3.13.

`ast.`**`PyCF_TYPE_COMMENTS`**

PEP 484와 **PEP 526** 스타일 형 주석(`# type: <type>`, `# type: ignore <stuff>`)에 대한 지원을 활성화합니다.

Added in version 3.8.

`ast.`**`compare`**(*a*, *b*, */*, ***, *compare_attributes=False*)

Recursively compares two ASTs.

compare_attributes affects whether AST attributes are considered in the comparison. If *compare_attributes* is `False` (default), then attributes are ignored. Otherwise they must all be equal. This option is useful to check whether the ASTs are structurally equal but differ in whitespace or similar details. Attributes include line numbers and column offsets.

Added in version 3.14.

32.1.5 Command-line usage

Added in version 3.9.

`ast` 모듈은 명령 줄에서 스크립트로 실행될 수 있습니다. 다음과 같이 간단합니다:

```
python -m ast [-m <mode>] [-a] [infile]
```

다음과 같은 옵션이 허용됩니다:

-h, --help
도움말 메시지를 표시하고 종료합니다.

-m <mode>
--mode <mode>
`parse()`의 *mode* 인자와 같이, 컴파일해야 하는 코드 종류를 지정합니다.

--no-type-comments
형 주석을 구문 분석하지 않습니다.

-a, --include-attributes
줄 번호와 열 오프셋과 같은 어트리뷰트를 포함합니다.

-i <indent>
--indent <indent>
AST에서 노드 들여쓰기(스페이스 수).

--feature-version <version>
Python version in the format 3.x (for example, 3.10). Defaults to the current version of the interpreter.

Added in version 3.14.

-O <level>
--optimize <level>
Optimization level for parser. Defaults to no optimization.

Added in version 3.14.

--show-empty
Show empty lists and fields that are `None`. Defaults to not showing empty objects.

Added in version 3.14.

`infile`이 지정되면 그 내용이 AST로 구문 분석되고 stdout에 덤프 됩니다. 그렇지 않으면, stdin에서 내용을 읽습니다.

더 보기

Green Tree Snakes, 파이썬 AST로 작업하는 것에 대한 자세한 내용이 있는 외부 문서 자원.

ASTTokens는 토큰의 위치와 토큰을 생성한 소스 코드의 텍스트로 파이썬 AST에 주석을 추가합니다. 이는 소스 코드 변환을 수행하는 도구에 유용합니다.

leoAst.py unifies the token-based and parse-tree-based views of python programs by inserting two-way links between tokens and ast nodes.

LibCST는 코드를 ast 트리처럼 보이고 모든 포매팅 세부 정보를 유지하는 구상 구문 트리(Concrete Syntax Tree)로 구문 분석합니다. 자동화된 리팩토링 (codemod) 응용 프로그램과 린터(linter)를 구축하는 데 유용합니다.

Parso is a Python parser that supports error recovery and round-trip parsing for different Python versions (in multiple Python versions). Parso is also able to list multiple syntax errors in your Python file.

32.2 `symtable` — 컴파일러 심볼 테이블 액세스

소스 코드: Lib/symtable.py

심볼 테이블은 바이트 코드가 생성되기 바로 전에 AST에서 컴파일러에 의해 생성됩니다. 심볼 테이블은 코드에서 모든 식별자의 스코프를 계산합니다. `symtable`은 이러한 테이블을 검사하는 인터페이스를 제공합니다.

32.2.1 심볼 테이블 생성하기

`symtable.`**`symtable`**(*code, filename, compile_type*)

파이썬 소스 *code*에 대한 최상위 `SymbolTable`을 반환합니다. *filename*은 코드가 들어있는 파일의 이름입니다. *compile_type*은 `compile()`에 대한 *mode* 인자와 같습니다.

32.2.2 심볼 테이블 검사하기

class `symtable.`**`SymbolTableType`**

An enumeration indicating the type of a `SymbolTable` object.

`MODULE = "module"`

Used for the symbol table of a module.

`FUNCTION = "function"`

Used for the symbol table of a function.

`CLASS = "class"`

Used for the symbol table of a class.

The following members refer to different flavors of annotation scopes.

`ANNOTATION = "annotation"`

Used for annotations if `from __future__ import annotations` is active.

`TYPE_ALIAS = "type alias"`

Used for the symbol table of `type` constructions.

`TYPE_PARAMETERS = "type parameters"`

Used for the symbol table of generic functions or generic classes.

`TYPE_VARIABLE = "type variable"`

Used for the symbol table of the bound, the constraint tuple or the default value of a single type variable in the formal sense, i.e., a TypeVar, a TypeVarTuple or a ParamSpec object (the latter two do not support a bound or a constraint tuple).

Added in version 3.13.

class `symtable.`**`SymbolTable`**

블록에 대한 이름 공간 테이블. 생성자는 공개되지 않습니다.

`get_type`()

심볼 테이블의 형을 돌려줍니다. 가능한 값은 `SymbolTableType` 열거형의 멤버들입니다.

버전 3.12에서 변경: Added `'annotation'`, `'TypeVar bound'`, `'type alias'`, and `'type parameter'` as possible return values.

버전 3.13에서 변경: Return values are members of the `SymbolTableType` enumeration.

The exact values of the returned string may change in the future, and thus, it is recommended to use `SymbolTableType` members instead of hard-coded strings.

`get_id`()

테이블의 식별자를 돌려줍니다.

get_name()

Return the table' s name. This is the name of the class if the table is for a class, the name of the function if the table is for a function, or `'top'` if the table is global (*get_type()* returns `'module'`). For type parameter scopes (which are used for generic classes, functions, and type aliases), it is the name of the underlying class, function, or type alias. For type alias scopes, it is the name of the type alias. For *TypeVar* bound scopes, it is the name of the `TypeVar`.

get_lineno()

이 테이블이 나타내는 블록의 첫 번째 줄 번호를 반환합니다.

is_optimized()

이 테이블의 지역(locals)을 최적화할 수 있으면 `True`를 반환합니다.

is_nested()

블록이 중첩된 클래스나 함수면 `True`를 반환합니다.

has_children()

블록에 중첩된 이름 공간이 있으면 `True`를 반환합니다. 이것들은 *get_children()*으로 얻을 수 있습니다.

get_identifiers()

Return a view object containing the names of symbols in the table. See the *documentation of view objects*.

lookup(*name*)

테이블에서 *name*을 찾아서 *Symbol* 인스턴스를 반환합니다.

get_symbols()

테이블에 있는 이름에 대한 *Symbol* 인스턴스 리스트를 반환합니다.

get_children()

중첩된 심볼 테이블의 리스트를 반환합니다.

class symtable.**Function**

함수나 메서드의 이름 공간. 이 클래스는 *SymbolTable*을 상속합니다.

get_parameters()

이 함수의 매개 변수 이름을 포함하는 튜플을 반환합니다.

get_locals()

이 함수의 지역 이름을 포함하는 튜플을 반환합니다.

get_globals()

이 함수의 전역 이름을 포함하는 튜플을 반환합니다.

get_nonlocals()

이 함수에서 명시적으로 선언된 nonlocal 이름을 포함하는 튜플을 반환합니다.

get_frees()

이 함수의 자유 (클로저) 변수 이름을 포함하는 튜플을 반환합니다.

class symtable.**Class**

클래스의 이름 공간. 이 클래스는 *SymbolTable*을 상속합니다.

get_methods()

클래스에서 선언된 메서드 같은 함수의 이름을 포함하는 튜플을 반환합니다.

Here, the term 'method' designates *any* function defined in the class body via `def` or `async def`.

Functions defined in a deeper scope (e.g., in an inner class) are not picked up by *get_methods()*.

예를 들면:

```
>>> import symtable
>>> st = symtable.symtable('''
... def outer(): pass
...
... class A:
...     def f():
...         def w(): pass
...
...     def g(self): pass
...
...     @classmethod
...     async def h(cls): pass
...
...     global outer
...     def outer(self): pass
... ''', 'test', 'exec')
>>> class_A = st.get_children()[2]
>>> class_A.get_methods()
('f', 'g', 'h')
```

Although `A().f()` raises *TypeError* at runtime, `A.f` is still considered as a method-like function.

Deprecated since version 3.14, will be removed in version 3.16.

class symtable.**Symbol**

소스의 식별자에 해당하는 *SymbolTable*의 항목. 생성자는 공개되지 않습니다.

get_name()

심볼의 이름을 돌려줍니다.

is_referenced()

심볼이 블록에서 사용되면 `True`를 반환합니다.

is_imported()

심볼이 import 문에서 만들어지면 `True`를 반환합니다.

is_parameter()

심볼이 매개 변수면 `True`를 반환합니다.

is_type_parameter()

Return `True` if the symbol is a type parameter.

Added in version 3.14.

is_global()

심볼이 전역이면 `True`를 반환합니다.

is_nonlocal()

심볼이 nonlocal이면 `True`를 반환합니다.

is_declared_global()

심볼이 global 문으로 전역으로 선언되면 `True`를 반환합니다.

is_local()

심볼이 블록의 지역이면 `True`를 반환합니다.

is_annotated()

심볼이 어노테이트 되었으면 `True`를 반환합니다.

Added in version 3.6.

is_free()

심볼이 블록에서 참조되지만 대입되지 않으면 `True`를 반환합니다.

is_free_class()

Return *True* if a class-scoped symbol is free from the perspective of a method.

Consider the following example:

```
def f():
    x = 1  # function-scoped
    class C:
        x = 2  # class-scoped
        def method(self):
            return x
```

In this example, the class-scoped symbol `x` is considered to be free from the perspective of `C.method`, thereby allowing the latter to return *1* at runtime and not *2*.

Added in version 3.14.

is_assigned()

심볼이 블록에 대입되면 `True`를 반환합니다.

is_comp_iter()

Return `True` if the symbol is a comprehension iteration variable.

Added in version 3.14.

is_comp_cell()

Return `True` if the symbol is a cell in an inlined comprehension.

Added in version 3.14.

is_namespace()

이름 연결(name binding)이 새로운 이름 공간을 도입하면 `True`를 반환합니다.

이름이 함수나 클래스 문의 대상으로 사용되면 참입니다.

예를 들면:

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

하나의 이름을 여러 객체에 연결할 수 있음에 유의하십시오. 결과가 `True` 이면, 이름은 새 이름 공간을 도입하지 않는 int 나 list와 같은 다른 객체에도 연결되어있을 수 있습니다.

get_namespaces()

이 이름에 연결된 이름 공간의 리스트를 돌려줍니다.

get_namespace()

이 이름에 연결된 이름 공간을 돌려줍니다. 이 이름에 연결된 이름 공간이 둘 이상이거나 없으면, *ValueError*가 발생합니다.

32.2.3 Command-Line Usage

Added in version 3.13.

The *symtable* module can be executed as a script from the command line.

```
python -m symtable [infile...]
```

Symbol tables are generated for the specified Python source files and dumped to stdout. If no input file is specified, the content is read from stdin.

32.3 token — 파이썬 구문 분석 트리에 사용되는 상수

소스 코드: Lib/token.py

이 모듈은 구문 분석 트리의 말단 노드의 숫자 값을 나타내는 상수를 제공합니다 (터미널 토큰). 언어 문법의 문맥에서 이름의 정의는 파이썬 배포판의 `Grammar/Tokens` 파일을 참조하십시오. 이름이 매핑되는 특정 숫자 값은 파이썬 버전 간에 변경될 수 있습니다.

이 모듈은 숫자 코드에서 이름으로의 매핑과 몇몇 함수도 제공합니다. 이 함수는 파이썬 C 헤더 파일의 정의를 반영합니다.

Note that a token' s value may depend on tokenizer options. For example, a `"+"` token may be reported as either *PLUS* or *OP*, or a `"match"` token may be either *NAME* or *SOFT_KEYWORD*.

`token.`**`tok_name`**

이 모듈에 정의된 상수의 숫자 값을 다시 이름 문자열로 매핑하여 사람이 읽을 수 있는 구문 분석 트리 표현을 생성할 수 있도록 하는 딕셔너리.

`token.`**`ISTERMINAL`**(*x*)

터미널 토큰값이면 `True`를 반환합니다.

`token.`**`ISNONTERMINAL`**(*x*)

비 터미널 토큰값이면 `True`를 반환합니다.

`token.`**`ISEOF`**(*x*)

*x*가 입력의 마지막을 나타내는 표시면 `True`를 반환합니다.

토큰 상수는 다음과 같습니다:

`token.`**`NAME`**

Token value that indicates an identifier. Note that keywords are also initially tokenized as `NAME` tokens.

`token.`**`NUMBER`**

Token value that indicates a numeric literal

`token.`**`STRING`**

Token value that indicates a string or byte literal, excluding formatted string literals. The token string is not interpreted: it includes the surrounding quotation marks and the prefix (if given); backslashes are included literally, without processing escape sequences.

`token.`**`OP`**

A generic token value that indicates an operator or delimiter.

CPython 구현 상세: This value is only reported by the *tokenize* module. Internally, the tokenizer uses *exact token types* instead.

`token.`**`COMMENT`**

주석을 나타내는 데 사용되는 토큰값. 구문 분석기는 `COMMENT` 토큰을 무시합니다.

`token.`**`NEWLINE`**

Token value that indicates the end of a logical line.

`token.`**`NL`**

비종결 줄넘김을 나타내는데 사용되는 토큰값. `NL` 토큰은 코드의 논리적 줄이 여러 물리적 줄로 이어질 때 생성됩니다. 구문 분석기는 `NL` 토큰을 무시합니다.

`token.`**`INDENT`**

Token value used at the beginning of a logical line to indicate the start of an indented block.

`token.`**`DEDENT`**

Token value used at the beginning of a logical line to indicate the end of an indented block.

token.**FSTRING_START**

Token value used to indicate the beginning of an f-string literal.

CPython 구현 상세: The token string includes the prefix and the opening quote(s), but none of the contents of the literal.

token.**FSTRING_MIDDLE**

Token value used for literal text inside an f-string literal, including format specifications.

CPython 구현 상세: Replacement fields (that is, the non-literal parts of f-strings) use the same tokens as other expressions, and are delimited by *LBRACE*, *RBRACE*, *EXCLAMATION* and *COLON* tokens.

token.**FSTRING_END**

f-string의 끝을 나타내는 데 사용되는 토큰값.

CPython 구현 상세: The token string contains the closing quote(s).

token.**TSTRING_START**

Token value used to indicate the beginning of a template string literal.

CPython 구현 상세: The token string includes the prefix and the opening quote(s), but none of the contents of the literal.

Added in version 3.14.

token.**TSTRING_MIDDLE**

Token value used for literal text inside a template string literal including format specifications.

CPython 구현 상세: Replacement fields (that is, the non-literal parts of t-strings) use the same tokens as other expressions, and are delimited by *LBRACE*, *RBRACE*, *EXCLAMATION* and *COLON* tokens.

Added in version 3.14.

token.**TSTRING_END**

Token value used to indicate the end of a template string literal.

CPython 구현 상세: The token string contains the closing quote(s).

Added in version 3.14.

token.**ENDMARKER**

Token value that indicates the end of input. Used in top-level grammar rules.

token.**ENCODING**

소스 바이트열을 텍스트로 디코딩하는 데 사용되는 인코딩을 나타내는 토큰값. *tokenize.tokenize()*에 의해 반환되는 첫 번째 토큰은 항상 ENCODING 토큰입니다.

이 토큰 유형은 C 토크나이저가 사용하지 않지만 *tokenize* 모듈에 필요합니다.

The following token types are not produced by the *tokenize* module, and are defined for special uses in the tokenizer or parser:

token.**TYPE_IGNORE**

`type: ignore` 주석이 인식되었음을 나타내는 토큰값. 이러한 토큰은 *PyCF_TYPE_COMMENTS* 플래그가 있는 경우에만 일반 *COMMENT* 토큰 대신 생성됩니다.

token.**TYPE_COMMENT**

형 주석이 인식되었음을 나타내는 토큰값. 이러한 토큰은 *PyCF_TYPE_COMMENTS* 플래그가 있는 경우에만 일반 *COMMENT* 토큰 대신 생성됩니다.

token.**SOFT_KEYWORD**

Token value indicating a soft keyword.

The tokenizer never produces this value. To check for a soft keyword, pass a *NAME* token' s string to *keyword.issoftkeyword()*.

token.**ERRORTOKEN**

잘못된 입력을 나타내는 데 사용되는 토큰값.

The *tokenize* module generally indicates errors by raising exceptions instead of emitting this token. It can also emit tokens such as *OP* or *NAME* with strings that are later rejected by the parser.

The remaining tokens represent specific operators and delimiters. (The *tokenize* module reports these as *OP*; see `exact_type` in the *tokenize* documentation for details.)

토큰	값
token.**LPAR**	"("
token.**RPAR**	")"
token.**LSQB**	"["
token.**RSQB**	"]"
token.**COLON**	":"
token.**COMMA**	","
token.**SEMI**	";"
token.**PLUS**	"+"
token.**MINUS**	"-"
token.**STAR**	"*"
token.**SLASH**	"/"
token.**VBAR**	"|"
token.**AMPER**	"&"
token.**LESS**	"<"
token.**GREATER**	">"
token.**EQUAL**	"="

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

토큰	값
token.**DOT**	"."
token.**PERCENT**	"%"
token.**LBRACE**	"{"
token.**RBRACE**	"}"
token.**EQEQUAL**	"=="
token.**NOTEQUAL**	"!="
token.**LESSEQUAL**	"<="
token.**GREATEREQUAL**	">="
token.**TILDE**	"~"
token.**CIRCUMFLEX**	"^"
token.**LEFTSHIFT**	"<<"
token.**RIGHTSHIFT**	">>"
token.**DOUBLESTAR**	"**"
token.**PLUSEQUAL**	"+="
token.**MINEQUAL**	"-="
token.**STAREQUAL**	"*="
token.**SLASHEQUAL**	"/="
token.**PERCENTEQUAL**	"%="

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

토큰	값
token.**AMPEREQUAL**	"&="
token.**VBAREQUAL**	"|="
token.**CIRCUMFLEXEQUAL**	"^="
token.**LEFTSHIFTEQUAL**	"<<="
token.**RIGHTSHIFTEQUAL**	">>="
token.**DOUBLESTAREQUAL**	"**="
token.**DOUBLESLASH**	"//"
token.**DOUBLESLASHEQUAL**	"//="
token.**AT**	"@"
token.**ATEQUAL**	"@="
token.**RARROW**	"->"
token.**ELLIPSIS**	"..."
token.**COLONEQUAL**	":="
token.**EXCLAMATION**	"!"

다음과 같은 토큰이 아닌 상수가 제공됩니다:

token.**N_TOKENS**

The number of token types defined in this module.

token.**EXACT_TOKEN_TYPES**

A dictionary mapping the string representation of a token to its numeric code.

Added in version 3.8.

버전 3.5에서 변경: AWAIT 와 ASYNC 토큰이 추가되었습니다.

버전 3.7에서 변경: *COMMENT*, *NL* 및 *ENCODING* 토큰이 추가되었습니다.

버전 3.7에서 변경: AWAIT 와 ASYNC 토큰이 제거되었습니다. "async" 와 "await" 는 이제 *NAME* 토큰으로

토큰화됩니다.

버전 3.8에서 변경: *TYPE_COMMENT*, *TYPE_IGNORE*, *COLONEQUAL*이 추가되었습니다. AWAIT와 ASYNC 토큰을 다시 추가했습니다 (feature_version을 6 이하로 설정하여 *ast.parse()*로 구형 파이썬 버전의 구문 분석을 지원하는 데 필요합니다).

버전 3.12에서 변경: *EXCLAMATION*을 추가했습니다.

버전 3.13에서 변경: AWAIT 와 ASYNC 토큰을 다시 제거했습니다.

32.4 keyword — 파이썬 키워드 검사

소스 코드: Lib/keyword.py

이 모듈은 파이썬 프로그램이 문자열이 키워드 나 소프트 키워드인지 판단하게 합니다.

keyword.**iskeyword**(*s*)

*s*가 파이썬 키워드면 True를 반환합니다.

keyword.**kwlist**

인터프리터에 대해 정의된 모든 키워드를 포함하는 시퀀스. 특정 *__future__* 문이 적용될 때만 활성화되도록 키워드가 정의되어 있으면, 이러한 키워드도 함께 포함됩니다.

keyword.**issoftkeyword**(*s*)

*s*가 파이썬 소프트 키워드면 True를 반환합니다.

Added in version 3.9.

keyword.**softkwlist**

인터프리터에 대해 정의된 모든 소프트 키워드를 포함하는 시퀀스. 특정 *__future__* 문이 적용될 때만 활성화되도록 소프트(soft) 키워드가 정의되어 있으면, 이러한 키워드도 함께 포함됩니다.

Added in version 3.9.

32.5 tokenize — 파이썬 소스를 위한 토크나이저

소스 코드: Lib/tokenize.py

---

tokenize 모듈은 파이썬으로 구현된 파이썬 소스 코드를 위한 어휘 스캐너를 제공합니다. 이 모듈의 스캐너는 주석도 토큰으로 반환하므로, 화면 디스플레이용 색상 표시기를 포함하여 "예쁜 인쇄기"를 구현하는 데 유용합니다.

토큰 스트림 처리를 단순화하기 위해, 모든 연산자와 구분자 토큰과 *Ellipsis*는 범용 *OP* 토큰 유형을 사용하여 반환됩니다. 정확한 유형은 *tokenize.tokenize()*에서 반환된 네임드 튜플의 exact_type 프로퍼티를 확인하여 파악할 수 있습니다.

> **경고**
>
> Note that the functions in this module are only designed to parse syntactically valid Python code (code that does not raise when parsed using *ast.parse()*). The behavior of the functions in this module is **undefined** when providing invalid Python code and it can change at any point.

32.5.1 입력 토큰화하기

기본 진입점은 제너레이터입니다:

tokenize.**tokenize**(*readline*)

tokenize() 제너레이터는 하나의 인자 *readline*을 요구합니다. 이 인자는 파일 객체의 *io.IOBase.readline()* 메서드와 같은 인터페이스를 제공하는 콜러블 객체여야 합니다. 함수를 호출할 때마다 한 줄의 입력을 바이트열로 반환해야 합니다.

제너레이터는 다음 멤버를 갖는 5-튜플을 생성합니다: 토큰 유형; 토큰 문자열; 토큰이 소스에서 시작하는 줄과 열을 지정하는 정수의 2-튜플 `(srow, scol)`; 토큰이 소스에서 끝나는 줄과 열을 지정하는 정수의 2-튜플 `(erow, ecol)` 과 토큰이 발견된 줄. 전달된 줄(마지막 튜플 항목)은 물리적 줄입니다. 5-튜플은 필드 이름이 `type string start end line` 인 네임드 튜플로 반환됩니다.

반환된 네임드 튜플에는 *OP* 토큰에 대한 정확한 연산자 유형이 포함된 `exact_type` 이라는 추가 프로퍼티가 있습니다. 다른 모든 토큰 유형에서 `exact_type` 은 네임드 튜플 `type` 필드와 같습니다.

버전 3.1에서 변경: 네임드 튜플에 대한 지원이 추가되었습니다.

버전 3.3에서 변경: `exact_type` 에 대한 지원이 추가되었습니다.

tokenize() 는 **PEP 263**에 따라 UTF-8 BOM이나 인코딩 쿠키를 찾아 파일의 소스 인코딩을 결정합니다.

tokenize.**generate_tokens**(*readline*)

바이트열 대신에 유니코드 문자열을 읽는 소스를 토큰화합니다.

tokenize() 와 마찬가지로, *readline* 인자는 한 줄의 입력을 반환하는 콜러블입니다. 그러나, *generate_tokens()* 는 *readline* 이 바이트열이 아닌 문자열 객체를 반환할 것으로 기대합니다.

결과는 정확히 *tokenize()* 처럼 네임드 튜플을 산출하는 이터레이터입니다. *ENCODING* 토큰을 산출하지 않습니다.

token 모듈의 모든 상수도 *tokenize* 에서 내보냅니다.

토큰화 프로세스를 역전시키는 또 다른 함수가 제공됩니다. 이것은 스크립트를 토큰화하고, 토큰 스트림을 수정한 후, 수정된 스크립트를 다시 쓰는 도구를 만드는 데 유용합니다.

tokenize.**untokenize**(*iterable*)

토큰을 파이썬 소스 코드로 역 변환합니다. *iterable* 은 최소한 토큰 유형과 토큰 문자열의 두 요소가 있는 시퀀스를 반환해야 합니다. 추가 시퀀스 요소는 무시됩니다.

결과는 다시 토큰화하면 입력과 일치함이 보장되어, 변환은 무손실이고 왕복이 보장됩니다. 보증은 토큰 유형과 토큰 문자열에만 적용되어, 토큰 간의 간격(열 위치)은 변경될 수 있습니다.

tokenize() 에 의해 출력되는 첫 번째 토큰 시퀀스인 *ENCODING* 토큰을 사용하여 인코딩된 바이트열을 반환합니다. 입력에 인코딩 토큰이 없으면, 대신 str을 반환합니다.

tokenize() 는 토큰화하는 소스 파일의 인코딩을 감지해야 합니다. 이 작업을 수행하는 데 사용되는 함수를 사용할 수 있습니다:

tokenize.**detect_encoding**(*readline*)

detect_encoding() 함수는 파이썬 소스 파일을 디코딩할 때 사용해야 하는 인코딩을 감지하는 데 사용됩니다. *tokenize()* 제너레이터와 같은 방식으로, 하나의 인자 readline을 요구합니다.

readline을 최대 두 번 호출하고, 사용된 인코딩(문자열로)과 읽은 줄들(바이트열에서 디코드 되지 않습니다)의 리스트를 반환합니다.

PEP 263에 지정된 대로 UTF-8 BOM이나 인코딩 쿠키의 존재로부터 인코딩을 검색합니다. BOM과 쿠키가 모두 있지만 서로 일치하지 않으면 *SyntaxError* 가 발생합니다. BOM이 발견되면, `'utf-8-sig'` 가 인코딩으로 반환됩니다.

인코딩이 지정되지 않으면, 기본값인 `'utf-8'` 이 반환됩니다.

open() 을 사용하여 파이썬 소스 파일을 여십시오: *detect_encoding()* 을 사용하여 파일 인코딩을 감지합니다.

tokenize.**open**(*filename*)

detect_encoding() 에 의해 감지된 인코딩을 사용하여 읽기 전용 모드로 파일을 엽니다.

Added in version 3.2.

exception tokenize.**TokenError**

여러 줄로 나눌 수 있는 독스트링이나 표현식이 파일의 어디에서도 완료되지 않을 때 발생합니다, 예를 들어:

```
"""Beginning of
docstring
```

또는:

```
[1,
 2,
 3
```

32.5.2 명령 줄 사용법

Added in version 3.3.

tokenize 모듈은 명령 줄에서 스크립트로 실행될 수 있습니다. 이렇게 간단합니다:

```
python -m tokenize [-e] [filename.py]
```

허용되는 옵션은 다음과 같습니다:

-h, --help

이 도움말 메시지를 표시하고 종료합니다

-e, --exact

정확한 유형(exact_type)을 사용하여 토큰 이름을 표시합니다

`filename.py`가 지정되면 그 내용은 표준출력(stdout)으로 토큰화됩니다. 그렇지 않으면, 표준입력(stdin)에 대해 토큰화가 수행됩니다.

32.5.3 예제

float 리터럴을 Decimal 객체로 변환하는 스크립트 재 작성기의 예제:

```
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

def decistmt(s):
    """문장 문자열에 있는 float를 Decimal로 치환합니다.

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7'))"

    지수 형식은 플랫폼 C 라이브러리에서 상속됩니다. 알려진 경우는 "e-007" (윈도우) 와 "e-
↪07"
    (윈도우 외) 입니다. 우리는 단지 12자리 숫자를 보여주고 있고, 13번째 숫자는 5에 가깝지
    않으므로, 출력의 나머지 부분은 플랫폼 독립적이어야 합니다.

    >>> exec(s)  #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Decimal을 사용한 계산의 결과는 모든 플랫폼에서 같아야 합니다.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline)  # 문자열을 토큰화합니다
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # NUMBER 토큰을 바꿔치기합니다
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
            result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')
```

명령 줄에서 토큰화하는 예제. 스크립트:

```
def say_hello():
    print("Hello, World!")

say_hello()
```

는 다음 출력으로 토큰화됩니다. 여기서 첫 번째 열은 토큰이 발견된 줄/열 좌표의 범위이고, 두 번째 열은 토큰의 이름이며, 마지막 열은 토큰의 값입니다 (있다면)

```
$ python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          OP             '('
1,14-1,15:          OP             ')'
1,15-1,16:          OP             ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           OP             '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          OP             ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           OP             '('
4,10-4,11:          OP             ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

정확한 토큰 유형 이름은 `-e` 옵션을 사용하여 표시할 수 있습니다:

```
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          LPAR           '('
1,14-1,15:          RPAR           ')'
1,15-1,16:          COLON          ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
2,4-2,9:            NAME           'print'
2,9-2,10:           LPAR           '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          RPAR           ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           LPAR           '('
4,10-4,11:          RPAR           ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

*generate_tokens()*로 바이트열 대신 유니코드 문자열을 읽는, 프로그래밍 방식으로 파일을 토큰화하는 예:

```
import tokenize

with tokenize.open('hello.py') as f:
    tokens = tokenize.generate_tokens(f.readline)
    for token in tokens:
        print(token)
```

또는 *tokenize()*로 직접 바이트열을 읽는 예:

```
import tokenize

with open('hello.py', 'rb') as f:
    tokens = tokenize.tokenize(f.readline)
    for token in tokens:
        print(token)
```

32.6 tabnanny — 모호한 들여쓰기 감지

소스 코드: Lib/tabnanny.py

이 모듈은 당장은 스크립트로 호출하기 위한 것입니다. 하지만 IDE로 임포트 해서 아래에 설명된 *check()* 함수를 사용할 수 있습니다.

> **참고**
>
> 이 모듈에서 제공하는 API는 향후 배포에서 변경될 수 있습니다; 그러한 변경은 이전 버전과 호환되지 않을 수 있습니다.

tabnanny.**check**(*file_or_dir*)

*file_or_dir*가 디렉터리이고 심볼릭 링크가 아니면, *file_or_dir*라는 이름의 디렉터리 트리를 재귀적으로 내려가면서, 모든 .py 파일을 검사합니다. *file_or_dir*가 일반 파이썬 소스 파일이면, 공백과 관련된 문제가 있는지 확인합니다. 진단 메시지는 *print()* 함수를 사용하여 표준 출력에 기록됩니다.

tabnanny.**verbose**

상세 메시지를 인쇄할지를 나타내는 플래그. 이것은 스크립트로 호출되면 -v 옵션에 의해 증가합니다.

tabnanny.**filename_only**

공백 관련 문제가 있는 파일의 파일명만 인쇄할지를 나타내는 플래그. 이것은 스크립트로 호출되면 `-q` 옵션에 의해 참으로 설정됩니다.

exception tabnanny.**NannyNag**

모호한 들여쓰기를 감지하면 `process_tokens()`에 의해 발생합니다. `check()`에서 잡아서 처리됩니다.

tabnanny.**process_tokens**(*tokens*)

이 함수는 `tokenize` 모듈에서 생성된 토큰을 처리하기 위해 `check()`에서 사용됩니다.

더 보기

모듈 ***tokenize***
파이썬 소스 코드를 위한 어휘 스캐너.

32.7 pyclbr — 파이썬 모듈 브라우저 지원

소스 코드: Lib/pyclbr.py

`pyclbr` 모듈은 파이썬 코드 모듈에 정의된 함수, 클래스 및 메서드에 대한 제한된 정보를 제공합니다. 이 정보는 모듈 브라우저를 구현하기에 충분합니다. 정보는 모듈을 임포트 하기보다는 파이썬 소스 코드에서 추출되므로 이 모듈은 신뢰할 수 없는 코드와 함께 사용하는 것이 안전합니다. 이 제한으로 인해 이 모듈을 모든 표준 및 선택 확장 모듈을 포함하여 파이썬으로 구현되지 않은 모듈에 사용할 수 없습니다.

pyclbr.**readmodule**(*module*, *path=None*)

모듈 수준의 클래스 이름을 클래스 설명자에 매핑하는 딕셔너리를 돌려줍니다. 가능하면, 임포트 된 베이스 클래스에 관한 설명자가 포함됩니다. 매개 변수 *module*은 읽을 모듈 이름이 들어있는 문자열입니다; 패키지 내의 모듈 이름일 수 있습니다. 주어지면, *path*는 `sys.path` 앞에 추가된 디렉터리 경로 시퀀스인데, 모듈 소스 코드의 위치를 찾는 데 사용됩니다.

이 함수는 원래 인터페이스이며 이전 버전과의 호환성을 위해서만 유지됩니다. 다음 함수의 필터링 된 버전을 반환합니다.

pyclbr.**readmodule_ex**(*module*, *path=None*)

`def` 나 `class` 문을 사용하여 모듈에 정의된 각 함수 및 클래스에 대한 함수나 클래스 설명자를 포함하는 딕셔너리 기반 트리를 반환합니다. 반환된 딕셔너리는 모듈 수준의 함수와 클래스 이름을 해당 설명자에 대응합니다. 중첩된 객체는 그들의 parent의 children 딕셔너리에 들어갑니다. readmodule과 마찬가지로, *module*은 읽을 모듈의 이름을 지정하고 *path*는 sys.path의 앞에 추가됩니다. 읽히는 모듈이 패키지면, 반환된 딕셔너리는 키 `'__path__'`를 가지는데, 값은 패키지 검색 경로를 포함하는 리스트입니다.

Added in version 3.7: 중첩된 정의에 대한 설명자. 새로운 children 어트리뷰트를 통해 액세스할 수 있습니다. 각에는 새로운 parent 어트리뷰트가 있습니다.

이러한 함수에 의해 반환되는 설명자는 Function과 Class 클래스의 인스턴스입니다. 사용자가 이러한 클래스의 인스턴스를 만들 것으로 기대하지 않습니다.

32.7.1 Function 객체

class pyclbr.**Function**

클래스 `Function` 인스턴스는 def 문으로 정의된 함수를 설명합니다. 다음과 같은 어트리뷰트를 가지고 있습니다:

file

함수가 정의된 파일의 이름.

module

설명된 함수를 정의하는 모듈의 이름.

name

함수의 이름.

lineno

정의가 시작되는 파일의 줄 번호.

parent

최상위 함수면, `None`. 중첩된 함수면, 부모.

Added in version 3.7.

children

이름을 중첩된 함수와 클래스에 관한 설명자로 매핑하는 딕셔너리.

Added in version 3.7.

is_async

`True` for functions that are defined with the `async` prefix, `False` otherwise.

Added in version 3.10.

32.7.2 Class 객체

class `pyclbr.`**Class**

클래스 `Class` 인스턴스는 class 문으로 정의된 클래스를 설명합니다. *`Function`*과 같은 어트리뷰트에 더해 두 개의 어트리뷰트가 더 있습니다.

file

클래스가 정의된 파일의 이름.

module

설명된 클래스를 정의하는 모듈의 이름.

name

클래스의 이름.

lineno

정의가 시작되는 파일의 줄 번호.

parent

최상위 클래스면, `None`. 중첩된 클래스면, 부모.

Added in version 3.7.

children

이름을 중첩된 함수와 클래스에 관한 설명자로 매핑하는 딕셔너리.

Added in version 3.7.

super

설명되는 클래스의 직접적인 베이스 클래스를 설명하는 `Class` 객체의 리스트. 슈퍼 클래스로 명명되었지만 *`readmodule_ex()`*가 찾을 수 없는 클래스는 `Class` 객체가 아니라 클래스 이름을 담은 문자열로 나열됩니다.

methods

메서드 이름을 줄 번호에 매핑하는 딕셔너리. 최신 *`children`* 딕셔너리에서 파생될 수 있지만, 이전 버전과의 호환성을 위해 남아있습니다.

32.8 py_compile — 파이썬 소스 파일 컴파일

소스 코드: Lib/py_compile.py

---

py_compile 모듈은 소스 파일에서 바이트 코드 파일을 생성하는 함수와 모듈 소스 파일이 스크립트로 호출될 때 사용되는 또 다른 함수를 제공합니다.

자주 사용되지는 않지만, 특히 일부 사용자가 소스 코드가 들어있는 디렉터리에 바이트 코드 캐시 파일을 쓸 수 있는 권한이 없을 때, 이 함수는 공유 사용을 위해 모듈을 설치할 때 유용할 수 있습니다.

exception py_compile.**PyCompileError**

파일을 컴파일하는 도중 에러가 일어날 때 발생하는 예외.

py_compile.**compile**(*file*, *cfile=None*, *dfile=None*, *doraise=False*, *optimize=-1*, *invalidation_mode=PycInvalidationMode.TIMESTAMP*, *quiet=0*)

바이트 코드로 소스 파일을 컴파일하고 바이트 코드 캐시 파일을 기록합니다. 소스 코드는 *file*이 지정하는 이름의 파일에서 로드됩니다. 바이트 코드는 *cfile*에 기록되며, 기본값은 `.pyc`로 끝나는 **PEP 3147/PEP 488** 경로입니다. 예를 들어, *file*이 `/foo/bar/baz.py`이면 *cfile*은 파이썬 3.2의 경우 `/foo/bar/__pycache__/baz.cpython-32.pyc`로 기본 설정됩니다. *dfile*이 지정되면, 예외 트레이스백에 표시하기 위해 소스 줄을 얻은 소스 파일 이름으로 *file* 대신 사용됩니다. *doraise*가 참이면, *file*을 컴파일하는 동안 에러를 만나면 *PyCompileError*가 발생합니다. *doraise*가 거짓(기본값)이면, `sys.stderr`에 에러 문자열이 기록되지만, 예외는 발생하지 않습니다. 이 함수는 바이트 컴파일된 파일의 경로, 즉 사용된 *cfile* 값을 반환합니다.

*doraise*와 *quiet* 인자는 파일을 컴파일하는 동안 에러를 처리하는 방법을 결정합니다. *quiet*가 0이나 1이고, *doraise*가 거짓이면, 기본 동작이 활성화됩니다: 에러 문자열이 `sys.stderr`에 기록되고, 함수는 경로 대신 `None`을 반환합니다. *doraise*가 참이면, 대신 *PyCompileError*가 발생합니다. 그러나 *quiet*가 2이면, 아무런 메시지도 기록되지 않고, *doraise*는 효과가 없습니다.

*cfile*이 되는 경로(명시적으로 지정되거나 계산된 경로)가 심볼릭 링크나 비정규 파일이면, *FileExistsError*가 발생합니다. 이것은 바이트 컴파일된 파일을 해당 경로에 쓸 수 있을 때 임포트가 해당 경로를 일반 파일로 바꾼다는 경고로 작용합니다. 이는 동시 파일 기록 문제를 방지하기 위해 최종 바이트 컴파일된 파일을 위치시키는데 파일 이름 바꾸기를 사용하는 임포트의 부작용입니다.

*optimize*는 최적화 수준을 제어하고 내장 *compile()* 함수로 전달됩니다. 기본값 `-1`은 현재 인터프리터의 최적화 수준을 선택합니다.

*invalidation_mode*는 *PycInvalidationMode* enum의 멤버여야 하며 실행 시간에 생성된 바이트 코드 캐시를 무효로 하는 방법을 제어합니다. `SOURCE_DATE_EPOCH` 환경 변수가 설정되면 기본값은 *PycInvalidationMode.CHECKED_HASH*이고, 그렇지 않으면 기본값은 *PycInvalidationMode.TIMESTAMP*입니다.

버전 3.2에서 변경: *cfile*의 기본값을 **PEP 3147**과 호환되도록 변경했습니다. 이전 기본값은 *file* + `'c'`(최적화가 활성화되었으면 `'o'`)입니다. 또한 *optimize* 매개 변수가 추가되었습니다.

버전 3.4에서 변경: 바이트 코드 캐시 파일 쓰기에 *importlib*를 사용하도록 코드를 변경했습니다. 이것은 파일 생성/기록 의미가 이제 *importlib*가 하는 것과 일치한다는 것을 의미합니다, 예를 들어 권한, 쓰기-와-이동 의미 등. 또한, *cfile*이 심볼릭 링크나 비정규 파일이면 *FileExistsError*가 발생시키는 경고를 추가했습니다.

버전 3.7에서 변경: *invalidation_mode* 매개 변수가 **PEP 552**에 지정된 대로 추가되었습니다. `SOURCE_DATE_EPOCH` 환경 변수가 설정되면, *invalidation_mode*는 *PycInvalidationMode.CHECKED_HASH*로 강제 설정됩니다.

버전 3.7.2에서 변경: `SOURCE_DATE_EPOCH` 환경 변수는 더는 *invalidation_mode* 인자의 값을 재정의하지 않으며, 대신 기본값을 결정합니다.

버전 3.8에서 변경: *quiet* 매개 변수가 추가되었습니다.

class py_compile.**PycInvalidationMode**

인터프리터가 바이트 코드 파일이 소스 파일에 대해 최신 버전인지를 결정하는 데 사용할 수 있는 가능한 방법의 열거형입니다. .pyc 파일은 헤더에서 원하는 무효화 모드를 가리킵니다. 파이썬이 실행 시간에 .pyc 파일을 무효로 하는 방법에 대한 자세한 내용은 pyc-invalidation를 참조하십시오.

Added in version 3.7.

TIMESTAMP

.pyc 파일은 파이썬이 실행 시간에 소스 파일의 메타 데이터와 비교하여 .pyc 파일을 재생성해야 하는지를 결정할 소스 파일의 타임스탬프와 크기를 포함합니다.

CHECKED_HASH

.pyc 파일은 파이썬이 실행 시간에 소스와 비교하여 .pyc 파일을 다시 생성해야 하는지를 결정할 소스 파일 내용의 해시를 포함합니다.

UNCHECKED_HASH

*CHECKED_HASH*와 마찬가지로, .pyc 파일에는 소스 파일 내용의 해시가 포함됩니다. 하지만, 파이썬은 실행 시간에 .pyc 파일이 최신 버전이라고 가정하고, 소스 파일에 대해 .pyc를 전혀 검증하지 않습니다.

이 옵션은 .pycs가 빌드 시스템처럼 파이썬 외부 시스템에 의해 최신 상태로 유지될 때 유용합니다.

32.8.1 명령줄 인터페이스

이 모듈은 스크립트로 호출하여 여러 소스 파일을 컴파일할 수 있습니다. *filenames*로 이름이 지정된 파일이 컴파일되고 결과 바이트 코드가 일반적인 방식으로 캐시 됩니다. 이 프로그램은 소스 파일을 찾기 위해 디렉터리 구조를 검색하지 않습니다; 명시적으로 이름이 지정된 파일 만 컴파일합니다. 컴파일할 수 없는 파일이 있으면 종료 상태가 0이 아닙니다.

<file> ... <fileN>

-

위치 인자는 컴파일할 파일입니다. -가 유일한 매개 변수면, 파일 목록을 표준 입력에서 가져옵니다.

-q, --quiet

에러 출력을 억제합니다.

버전 3.2에서 변경: -에 대한 지원이 추가되었습니다.

버전 3.10에서 변경: *-q*에 대한 지원이 추가되었습니다.

더 보기

모듈 ***compileall***
디렉터리 트리에 있는 모든 파이썬 소스 파일을 컴파일하는 유틸리티.

32.9 compileall — 파이썬 라이브러리 바이트 컴파일하기

소스 코드: Lib/compileall.py

이 모듈은 파이썬 라이브러리 설치를 지원하는 몇 가지 유틸리티 함수를 제공합니다. 이 함수는 디렉터리 트리에서 파이썬 소스 파일을 컴파일합니다. 이 모듈을 사용하면 라이브러리 설치 시 캐시 된 바이트 코드 파일을 만들 수 있으므로, 라이브러리 디렉터리에 쓰기 권한이 없는 사용자도 사용할 수 있도록 합니다.

가용성: not WASI.

이 모듈은 웹어셈블리에서 작동하지 않거나 제공되지 않습니다. 자세한 내용은 웹어셈블리 플랫폼을 참조하세요.

32.9.1 명령 줄 사용

이 모듈은 파이썬 소스를 컴파일하는 스크립트로 작동할 수 있습니다 (**python -m compileall**을 사용합니다).

directory ...

file ...

위치 인자는 컴파일할 파일이나 소스 파일을 포함하는 디렉터리이며 재귀적으로 탐색 됩니다. 인자가 주어지지 않으면, 명령 줄이 `-l <directories from sys.path>` 인 것처럼 행동합니다.

-l

서브 디렉터리를 재귀적으로 탐색하지 않고, 이름이 지정되었거나 암시된 디렉터리에 직접 포함된 소스 코드 파일 만 컴파일합니다.

-f

타임스탬프가 최신일 때도 강제로 다시 빌드합니다.

-q

컴파일된 파일 목록을 인쇄하지 않습니다. 한 번 전달하면, 에러 메시지는 여전히 인쇄됩니다. 두 번 전달하면 (`-qq`), 모든 출력이 억제됩니다.

-d `destdir`

디렉터리가 컴파일되는 각 파일의 경로 앞에 추가됩니다. 이것은 컴파일 시간 트레이스백에 나타나며, 바이트 코드 파일에 컴파일되어 들어가서, 바이트 코드 파일이 실행되는 시점에 소스 파일이 존재하지 않으면 트레이스백과 기타 메시지에 사용됩니다.

-s `strip_prefix`

Remove the given prefix from paths recorded in the `.pyc` files. Paths are made relative to the prefix.

This option can be used with `-p` but not with `-d`.

-p `prepend_prefix`

Prepend the given prefix to paths recorded in the `.pyc` files. Use `-p /` to make the paths absolute.

This option can be used with `-s` but not with `-d`.

-x `regex`

regex는 컴파일 대상으로 고려되는 각 파일의 전체 경로를 검색(search)하는 데 사용되며, 정규식이 일치를 생성하면 그 파일을 건너뜁니다.

-i `list`

파일 `list`를 읽고 포함된 각 줄을 컴파일할 파일과 디렉터리 목록에 추가합니다. `list`가 `-`이면, `stdin`에서 줄을 읽습니다.

-b

바이트 코드 파일을 레거시 위치 및 이름에 써서, 다른 버전의 파이썬이 만든 바이트 코드 파일을 덮어쓸 수 있습니다. 기본값은 여러 버전의 파이썬의 바이트 코드 파일이 공존할 수 있는 **PEP 3147** 위치와 이름에 파일을 쓰는 것입니다.

-r

서브 디렉터리의 최대 재귀 수준을 제어합니다. 이것이 주어지면, `-l` 옵션은 고려되지 않습니다. **python -m compileall <directory> -r 0**은 **python -m compileall <directory> -l**과 동등합니다.

-j `N`

주어진 디렉터리 내의 파일을 컴파일하는 데 *N* 작업자를 사용합니다. `0`이 사용되면, *os.process_cpu_count()*의 결과가 사용됩니다.

--invalidation-mode `[timestamp|checked-hash|unchecked-hash]`

생성된 바이트 코드 파일이 실행 시간에 무효가 되는 방식을 제어합니다. `timestamp` 값은 소스 타임스탬프와 크기가 포함된 `.pyc` 파일이 생성됨을 의미합니다. `checked-hash`와 `unchecked-hash` 값은 해시 기반 pyc를 생성합니다. 해시 기반 pyc는 타임스탬프 대신 소스 파일 내용의 해시를 포함합니다. 파이썬이 실행 시간에 바이트 코드 캐시 파일의 유효성을 검사하는 방법에 대한 자세한 내

용은 pyc-invalidation를 참조하십시오. 기본값은 SOURCE_DATE_EPOCH 환경 변수가 설정되지 않으면 timestamp이고, SOURCE_DATE_EPOCH 환경 변수가 설정되면 checked-hash입니다.

-o level

주어진 최적화 수준으로 컴파일합니다. 한 번에 여러 수준으로 컴파일하기 위해 여러 번 사용할 수 있습니다 (예를 들어, compileall -o 1 -o 2).

-e dir

지정된 디렉터리 외부를 가리키는 심볼릭 링크를 무시합니다.

--hardlink-dupes

최적화 수준이 다른 두 개의 .pyc 파일의 내용이 같으면, 하드 링크를 사용하여 중복 파일을 통합합니다.

버전 3.2에서 변경: -i, -b 및 -h 옵션이 추가되었습니다.

버전 3.5에서 변경: -j, -r 및 -qq 옵션이 추가되었습니다. -q 옵션이 다중 수준 값으로 변경되었습니다. -b는 항상 .pyc로 끝나는 바이트 코드 파일을 생성하며, 결코 .pyo를 생성하지 않습니다.

버전 3.7에서 변경: --invalidation-mode 옵션이 추가되었습니다.

버전 3.9에서 변경: -s, -p, -e 및 --hardlink-dupes 옵션을 추가했습니다. 기본 재귀 제한을 10에서 *sys.getrecursionlimit()*로 올렸습니다. -o 옵션을 여러 번 지정할 수 있는 가능성이 추가되었습니다.

compile() 함수가 사용하는 최적화 수준을 제어하는 명령 줄 옵션은 없습니다. 파이썬 인터프리터 자신이 그 옵션을 제공하고 있기 때문입니다: **python -O -m compileall**.

마찬가지로, *compile()* 함수는 *sys.pycache_prefix* 설정을 따릅니다. 생성된 바이트 코드 캐시는 *compile()*이 실행 시간에 사용될 것과 같은 *sys.pycache_prefix*(있다면)로 실행될 때만 유용합니다.

32.9.2 공용 함수

compileall.**compile_dir**(*dir, maxlevels=sys.getrecursionlimit(), ddir=None, force=False, rx=None, quiet=0, legacy=False, optimize=-1, workers=1, invalidation_mode=None, *, stripdir=None, prependdir=None, limit_sl_dest=None, hardlink_dupes=False*)

*dir*로 명명된 디렉터리 트리를 재귀적으로 탐색해 내려가면서, 발견되는 모든 .py 파일을 컴파일합니다. 모든 파일이 성공적으로 컴파일되면 참값을 반환하고, 그렇지 않으면 거짓값을 반환합니다.

maxlevels 매개 변수는 재귀의 깊이를 제한하는 데 사용됩니다; 기본값은 sys.getrecursionlimit()입니다.

*ddir*이 주어지면, 컴파일 시간 트레이스백에서 사용하기 위해 컴파일되는 각 파일의 경로 앞에 추가되며, 바이트 코드 파일에 컴파일되어 들어가서, 바이트 코드 파일이 실행되는 시점에 소스 파일이 존재하지 않으면 트레이스백과 기타 메시지에 사용됩니다.

*force*가 참이면, 타임스탬프가 최신일 때도 모듈이 다시 컴파일됩니다.

*rx*가 주어지면, 컴파일 대상으로 고려되는 각 파일의 전체 경로로 그것의 search 메서드를 호출하고, 참값을 반환하면 그 파일을 건너뜁니다. 이는 *re.Pattern* 객체로 주어지는 정규식과 일치하는 파일을 제외하는 데 사용할 수 있습니다.

*quiet*가 False나 0(기본값)이면, 파일명과 기타 정보가 표준 출력에 인쇄됩니다. 1로 설정하면, 에러만 인쇄됩니다. 2로 설정하면, 모든 출력이 억제됩니다.

*legacy*가 참이면, 바이트 코드 파일을 레거시 위치 및 이름에 써서, 다른 버전의 파이썬이 만든 바이트 코드 파일을 덮어쓸 수 있습니다. 기본값은 여러 버전의 파이썬의 바이트 코드 파일이 공존할 수 있는 **PEP 3147** 위치와 이름에 파일을 쓰는 것입니다.

*optimize*는 컴파일러의 최적화 수준을 지정합니다. 내장 *compile()* 함수로 전달됩니다. 한 번의 호출로 하나의 .py 파일을 여러 번 컴파일하도록 하는 최적화 수준의 시퀀스도 허용합니다.

인자 *workers*는 파일을 컴파일하는 데 병렬로 사용되는 작업자 수를 지정합니다. 기본값은 다중 작업자를 사용하지 않는 것입니다. 플랫폼이 다중 작업자를 사용할 수 없고 *workers* 인자가 주어지면, 순차 컴파일로 대체합니다. *workers*가 0이면 시스템의 코어 수가 사용됩니다. *workers*가 0보다 작으면, *ValueError*가 발생합니다.

*invalidation_mode*는 `py_compile.PycInvalidationMode` 열거형의 멤버여야 하며 실행 시간에 생성된 pyc가 무효가 되는 방식을 제어합니다.

stripdir, *prependdir* 및 *limit_sl_dest* 인자는 위에서 설명한 `-s`, `-p` 및 `-e` 옵션에 해당합니다. `str` 이나 `os.PathLike`으로 지정할 수 있습니다.

*hardlink_dupes*가 참이고 최적화 수준이 다른 두 개의 `.pyc` 파일의 내용이 같으면, 하드 링크를 사용하여 중복 파일을 통합합니다.

버전 3.2에서 변경: *legacy*와 *optimize* 매개 변수가 추가되었습니다.

버전 3.5에서 변경: *workers* 매개 변수가 추가되었습니다.

버전 3.5에서 변경: *quiet* 매개 변수가 다중 수준 값으로 변경되었습니다.

버전 3.5에서 변경: *legacy* 매개 변수는 *optimize* 값과 상관없이 `.pyo` 파일이 아니라 `.pyc` 파일 만 기록합니다.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

버전 3.7에서 변경: *invalidation_mode* 매개 변수가 추가되었습니다.

버전 3.7.2에서 변경: *invalidation_mode* 매개 변수의 기본값이 `None`으로 변경되었습니다.

버전 3.8에서 변경: *workers*를 0으로 설정하면 최적의 코어 수가 선택됩니다.

버전 3.9에서 변경: *stripdir*, *prependdir*, *limit_sl_dest* 및 *hardlink_dupes* 인자가 추가되었습니다. *maxlevels*의 기본값이 `10`에서 `sys.getrecursionlimit()`으로 변경되었습니다.

`compileall.`**`compile_file`**(*fullname*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*, **, stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)

경로 *fullname*의 파일을 컴파일합니다. 파일이 성공적으로 컴파일되면 참값을 반환하고, 그렇지 않으면 거짓값을 반환합니다.

*ddir*이 주어지면, 컴파일 시간 트레이스백에서 사용하기 위해 컴파일되는 파일의 경로 앞에 추가되며, 바이트 코드 파일에 컴파일되어 들어가서, 바이트 코드 파일이 실행되는 시점에 소스 파일이 존재하지 않으면 트레이스백과 기타 메시지에 사용됩니다.

*rx*가 주어지면, 컴파일 중인 파일의 전체 경로가 그것의 `search` 메서드로 전달되고, 참값을 반환하면, 파일이 컴파일되지 않고 `True`가 반환됩니다. 이는 *re.Pattern* 객체로 주어지는 정규식과 일치하는 파일을 제외하는 데 사용할 수 있습니다.

*quiet*가 `False`나 `0`(기본값)이면, 파일명과 기타 정보가 표준 출력에 인쇄됩니다. `1`로 설정하면, 에러만 인쇄됩니다. `2`로 설정하면, 모든 출력이 억제됩니다.

*legacy*가 참이면, 바이트 코드 파일을 레거시 위치 및 이름에 써서, 다른 버전의 파이썬이 만든 바이트 코드 파일을 덮어쓸 수 있습니다. 기본값은 여러 버전의 파이썬의 바이트 코드 파일이 공존할 수 있는 **PEP 3147** 위치와 이름에 파일을 쓰는 것입니다.

*optimize*는 컴파일러의 최적화 수준을 지정합니다. 내장 `compile()` 함수로 전달됩니다. 한 번의 호출로 하나의 `.py` 파일을 여러 번 컴파일하도록 하는 최적화 수준의 시퀀스도 허용합니다.

*invalidation_mode*는 `py_compile.PycInvalidationMode` 열거형의 멤버여야 하며 실행 시간에 생성된 pyc가 무효가 되는 방식을 제어합니다.

stripdir, *prependdir* 및 *limit_sl_dest* 인자는 위에서 설명한 `-s`, `-p` 및 `-e` 옵션에 해당합니다. `str` 이나 `os.PathLike`으로 지정할 수 있습니다.

*hardlink_dupes*가 참이고 최적화 수준이 다른 두 개의 `.pyc` 파일의 내용이 같으면, 하드 링크를 사용하여 중복 파일을 통합합니다.

Added in version 3.2.

버전 3.5에서 변경: *quiet* 매개 변수가 다중 수준 값으로 변경되었습니다.

버전 3.5에서 변경: *legacy* 매개 변수는 *optimize* 값과 상관없이 `.pyo` 파일이 아니라 `.pyc` 파일 만 기록합니다.

버전 3.7에서 변경: *invalidation_mode* 매개 변수가 추가되었습니다.

버전 3.7.2에서 변경: *invalidation_mode* 매개 변수의 기본값이 `None`으로 변경되었습니다.

버전 3.9에서 변경: *stripdir*, *prependdir*, *limit_sl_dest* 및 *hardlink_dupes* 인자가 추가되었습니다.

`compileall.`**`compile_path`**(*skip_curdir=True, maxlevels=0, force=False, quiet=0, legacy=False, optimize=-1, invalidation_mode=None*)

`sys.path`에서 발견된 모든 `.py` 파일을 바이트 컴파일합니다. 모든 파일이 성공적으로 컴파일되면 참값을 반환하고, 그렇지 않으면 거짓값을 반환합니다.

*skip_curdir*가 참(기본값)이면, 현재 디렉터리가 검색에 포함되지 않습니다. 다른 모든 매개 변수는 *compile_dir()* 함수에 전달됩니다. 다른 컴파일 함수와 달리, `maxlevels`의 기본값은 `0`임에 유의하십시오.

버전 3.2에서 변경: *legacy*와 *optimize* 매개 변수가 추가되었습니다.

버전 3.5에서 변경: *quiet* 매개 변수가 다중 수준 값으로 변경되었습니다.

버전 3.5에서 변경: *legacy* 매개 변수는 *optimize* 값과 상관없이 `.pyo` 파일이 아니라 `.pyc` 파일 만 기록합니다.

버전 3.7에서 변경: *invalidation_mode* 매개 변수가 추가되었습니다.

버전 3.7.2에서 변경: *invalidation_mode* 매개 변수의 기본값이 `None`으로 변경되었습니다.

`Lib/` 서브 디렉터리와 그것의 모든 서브 디렉터리에 있는 모든 `.py` 파일을 강제로 다시 컴파일하려면 다음과 같이 합니다:

```
import compileall

compileall.compile_dir('Lib/', force=True)

# .svn 디렉터리의 파일을 제외하고, 같은 컴파일을 수행합니다.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)

# pathlib.Path 객체도 사용할 수 있습니다.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

더 보기

모듈 *py_compile*
단일 소스 파일을 바이트 컴파일합니다.

32.10 `dis` — Disassembler for Python bytecode

소스 코드: Lib/dis.py

---

dis 모듈은 CPython 바이트 코드를 역 어셈블 하여 분석을 지원합니다. 이 모듈이 입력으로 취하는 CPython 바이트 코드는 파일 `Include/opcode.h`에 정의되어 있으며 컴파일러와 인터프리터에서 사용됩니다.

바이트 코드는 CPython 인터프리터의 구현 세부 사항입니다. 파이썬 버전 간에 바이트 코드가 추가, 제거 또는 변경되지 않을 것이라는 보장은 없습니다. 이 모듈을 사용하는 것이 파이썬 VM이나 파이썬 릴리스에 걸쳐 작동할 것으로 생각하지 말아야 합니다.

버전 3.6에서 변경: 명령어마다 2바이트를 사용합니다. 이전에는 바이트 수가 명령어에 따라 달랐습니다.

버전 3.10에서 변경: The argument of jump, exception handling and loop instructions is now the instruction offset rather than the byte offset.

버전 3.11에서 변경: Some instructions are accompanied by one or more inline cache entries, which take the form of *CACHE* instructions. These instructions are hidden by default, but can be shown by passing `show_caches=True` to any *dis* utility. Furthermore, the interpreter now adapts the bytecode to specialize it for different runtime conditions. The adaptive bytecode can be shown by passing `adaptive=True`.

버전 3.12에서 변경: The argument of a jump is the offset of the target instruction relative to the instruction that appears immediately after the jump instruction' s *CACHE* entries.

As a consequence, the presence of the *CACHE* instructions is transparent for forward jumps but needs to be taken into account when reasoning about backward jumps.

버전 3.13에서 변경: The output shows logical labels rather than instruction offsets for jump targets and exception handlers. The `-O` command line option and the `show_offsets` argument were added.

버전 3.14에서 변경: The *-P* command-line option and the `show_positions` argument were added.

The *-S* command-line option is added.

Example: Given the function `myfunc()`:

```
def myfunc(alist):
    return len(alist)
```

the following command can be used to display the disassembly of `myfunc()`:

```
>>> dis.dis(myfunc)
  2           RESUME                   0

  3           LOAD_GLOBAL              1 (len + NULL)
              LOAD_FAST_BORROW         0 (alist)
              CALL                     1
              RETURN_VALUE
```

("2"는 줄 번호입니다).

32.10.1 Command-line interface

The *dis* module can be invoked as a script from the command line:

```
python -m dis [-h] [-C] [-O] [-P] [-S] [infile]
```

The following options are accepted:

-h, --help

Display usage and exit.

-C, --show-caches

Show inline caches.

Added in version 3.13.

-O, --show-offsets

Show offsets of instructions.

Added in version 3.13.

-P, --show-positions

Show positions of instructions in the source code.

Added in version 3.14.

-S, --specialized

Show specialized bytecode.

Added in version 3.14.

If `infile` is specified, its disassembled code will be written to stdout. Otherwise, disassembly is performed on compiled source code received from stdin.

32.10.2 바이트 코드 분석

Added in version 3.4.

바이트 코드 분석 API는 컴파일된 코드의 세부 사항에 쉽게 액세스 할 수 있도록 하는 `Bytecode` 객체로 파이썬 코드 조각을 감쌀 수 있도록 합니다.

class dis.**Bytecode**(*x*, **, first_line=None, current_offset=None, show_caches=False, adaptive=False, show_offsets=False, show_positions=False*)

함수, 제너레이터, 비동기 제너레이터, 코루틴, 메서드, 소스 코드 문자열 또는 (`compile()`에서 반환된) 코드 객체에 해당하는 바이트 코드를 분석합니다.

이것은 아래에 나열된 많은 함수, 특히 `get_instructions()`를 둘러싼 편리한 래퍼입니다, `Bytecode` 인스턴스를 이터레이트 하면 바이트 코드 연산이 `Instruction` 인스턴스로 산출되기 때문입니다.

*first_line*이 `None`이 아니면, 역 어셈블 된 코드에서 첫 번째 소스 줄에 대해 보고해야 하는 줄 번호를 나타냅니다. 그렇지 않으면, 소스 줄 정보(있다면)를 역 어셈블 된 코드 객체에서 직접 취합니다.

*current_offset*이 `None`이 아니면, 역 어셈블 된 코드의 명령어 오프셋을 나타냅니다. 이를 설정하면, `dis()`가 지정된 옵코드(opcode)에 대해 "현재 명령어" 마커를 표시합니다.

If *show_caches* is `True`, `dis()` will display inline cache entries used by the interpreter to specialize the bytecode.

If *adaptive* is `True`, `dis()` will display specialized bytecode that may be different from the original bytecode.

If *show_offsets* is `True`, `dis()` will include instruction offsets in the output.

If *show_positions* is `True`, `dis()` will include instruction source code positions in the output.

classmethod from_traceback(*tb*, **, show_caches=False*)

주어진 트레이스백에서 `Bytecode` 인스턴스를 구성하고, *current_offset*을 예외를 일으킨 명령어로 설정합니다.

codeobj

컴파일된 코드 객체.

first_line

코드 객체의 첫 번째 소스 줄 (사용 가능하다면)

dis()

바이트 코드 연산의 포맷된 보기를 반환합니다 (`dis.dis()`가 인쇄하는 것과 같지만, 여러 줄 문자열로 반환됩니다).

info()

`code_info()`처럼, 코드 객체에 대한 자세한 정보가 포함된 포맷된 여러 줄 문자열을 반환합니다.

버전 3.7에서 변경: 이제 코루틴과 비동기 제너레이터 객체를 처리할 수 있습니다.

버전 3.11에서 변경: Added the *show_caches* and *adaptive* parameters.

버전 3.13에서 변경: Added the *show_offsets* parameter

버전 3.14에서 변경: Added the *show_positions* parameter.

Example:

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
RESUME
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
LOAD_GLOBAL
LOAD_FAST_BORROW
CALL
RETURN_VALUE
```

32.10.3 분석 함수

dis 모듈은 또한 입력을 원하는 출력으로 직접 변환하는 다음 분석 함수를 정의합니다. 단일 작업만 수행해서, 중간 분석 객체가 유용하지 않을 때 유용할 수 있습니다:

`dis.code_info(x)`

제공된 함수, 제너레이터, 비동기 제너레이터, 코루틴, 메서드, 소스 코드 문자열 또는 코드 객체에 대한 자세한 코드 객체 정보가 포함된 포맷된 여러 줄 문자열을 반환합니다.

코드 정보 문자열의 정확한 내용은 구현에 따라 달라지며 파이썬 VM이나 파이썬 릴리스에 걸쳐 임의로 변경될 수 있습니다.

Added in version 3.2.

버전 3.7에서 변경: 이제 코루틴과 비동기 제너레이터 객체를 처리할 수 있습니다.

`dis.show_code(x, *, file=None)`

제공된 함수, 메서드, 소스 코드 문자열 또는 코드 객체에 대한 자세한 코드 객체 정보를 *file*(또는 *file*이 지정되지 않으면 `sys.stdout`)로 인쇄합니다.

이것은 `print(code_info(x), file=file)`의 편리한 축약 형으로, 인터프리터 프롬프트에서의 대화식 탐색을 위한 것입니다.

Added in version 3.2.

버전 3.4에서 변경: *file* 매개 변수를 추가했습니다.

`dis.dis(x=None, *, file=None, depth=None, show_caches=False, adaptive=False, show_offsets=False, show_positions=False)`

Disassemble the *x* object. *x* can denote either a module, a class, a method, a function, a generator, an asynchronous generator, a coroutine, a code object, a string of source code or a byte sequence of raw bytecode. For a module, it disassembles all functions. For a class, it disassembles all methods (including class and static methods). For a code object or sequence of raw bytecode, it prints one line per bytecode instruction. It also recursively disassembles nested code objects. These can include generator expressions, nested functions, the bodies of nested classes, and the code objects used for annotation scopes. Strings are first compiled to code objects with the *compile()* built-in function before being disassembled. If no object is provided, this function disassembles the last traceback.

역 어셈블리는 제공된다면 제공된 *file* 인자에, 그렇지 않으면 `sys.stdout`에 텍스트로 기록됩니다.

재귀의 최대 깊이는 `None`이 아닌 한 *depth*에 의해 제한됩니다. `depth=0`은 재귀가 없음을 의미합니다.

If *show_caches* is `True`, this function will display inline cache entries used by the interpreter to specialize the bytecode.

If *adaptive* is `True`, this function will display specialized bytecode that may be different from the original bytecode.

버전 3.4에서 변경: *file* 매개 변수를 추가했습니다.

버전 3.7에서 변경: 재귀 역 어셈블을 구현하고 *depth* 매개 변수를 추가했습니다.

버전 3.7에서 변경: 이제 코루틴과 비동기 제너레이터 객체를 처리할 수 있습니다.

버전 3.11에서 변경: Added the *show_caches* and *adaptive* parameters.

버전 3.13에서 변경: Added the *show_offsets* parameter.

버전 3.14에서 변경: Added the *show_positions* parameter.

dis.**distb**(*tb=None*, ***, *file=None*, *show_caches=False*, *adaptive=False*, *show_offset=False*, *show_positions=False*)

트레이스백의 최상단 함수를 역 어셈블 합니다. 전달되지 않으면 마지막 트레이스백을 사용합니다. 예외를 일으키는 명령어가 표시됩니다.

역 어셈블리는 제공된다면 제공된 *file* 인자에, 그렇지 않으면 `sys.stdout`에 텍스트로 기록됩니다.

버전 3.4에서 변경: *file* 매개 변수를 추가했습니다.

버전 3.11에서 변경: Added the *show_caches* and *adaptive* parameters.

버전 3.13에서 변경: Added the *show_offsets* parameter.

버전 3.14에서 변경: Added the *show_positions* parameter.

dis.**disassemble**(*code*, *lasti=-1*, ***, *file=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*, *show_positions=False*)

dis.**disco**(*code*, *lasti=-1*, ***, *file=None*, *show_caches=False*, *adaptive=False*, *show_offsets=False*, *show_positions=False*)

코드 객체를 역 어셈블 하고, *lasti*가 제공되면 마지막 명령어를 표시합니다. 출력은 다음 열로 나뉩니다:

1. the source code location of the instruction. Complete location information is shown if *show_positions* is true. Otherwise (the default) only the line number is displayed.
2. 현재 명령어, `-->`로 표시됩니다,
3. 레이블이 있는 명령어, `>>`로 표시됩니다,
4. 명령어의 주소,
5. 연산 코드 이름,
6. 연산 매개 변수, 그리고
7. 괄호 안에 있는 매개 변수의 해석.

매개 변수 해석은 지역과 전역 변수 이름, 상숫값, 분기 대상 및 비교 연산자를 인식합니다.

역 어셈블리는 제공된다면 제공된 *file* 인자에, 그렇지 않으면 `sys.stdout`에 텍스트로 기록됩니다.

버전 3.4에서 변경: *file* 매개 변수를 추가했습니다.

버전 3.11에서 변경: Added the *show_caches* and *adaptive* parameters.

버전 3.13에서 변경: Added the *show_offsets* parameter.

버전 3.14에서 변경: Added the *show_positions* parameter.

dis.**get_instructions**(*x*, ***, *first_line=None*, *show_caches=False*, *adaptive=False*)

제공된 함수, 메서드, 소스 코드 문자열 또는 코드 객체의 명령어들에 대한 이터레이터를 반환합니다.

이터레이터는 제공된 코드의 각 연산에 대한 세부 정보를 제공하는 `Instruction` 네임드 튜플의 연속을 생성합니다.

*first_line*이 `None`이 아니면, 역 어셈블 된 코드에서 첫 번째 소스 줄에 대해 보고해야 하는 줄 번호를 나타냅니다. 그렇지 않으면, 소스 줄 정보(있다면)를 역 어셈블 된 코드 객체에서 직접 취합니다.

The *adaptive* parameter works as it does in `dis()`.

Added in version 3.4.

버전 3.11에서 변경: Added the *show_caches* and *adaptive* parameters.

버전 3.13에서 변경: The *show_caches* parameter is deprecated and has no effect. The iterator generates the `Instruction` instances with the *cache_info* field populated (regardless of the value of *show_caches*) and it no longer generates separate items for the cache entries.

dis.**findlinestarts**(*code*)

This generator function uses the `co_lines()` method of the code object *code* to find the offsets which are starts of lines in the source code. They are generated as `(offset, lineno)` pairs.

버전 3.6에서 변경: 줄 번호가 줄어들 수 있습니다. 전에는, 언제나 증가했습니다.

버전 3.10에서 변경: The **PEP 626** `co_lines()` method is used instead of the `co_firstlineno` and `co_lnotab` attributes of the code object.

버전 3.13에서 변경: Line numbers can be `None` for bytecode that does not map to source lines.

dis.**findlabels**(*code*)

원시 컴파일된 바이트 코드 문자열 *code*에서 점프 대상인 모든 오프셋을 감지하고, 이러한 오프셋의 리스트를 반환합니다.

dis.**stack_effect**(*opcode*, *oparg=None*, *, *jump=None*)

인자 *oparg*를 갖는 *opcode*의 스택 효과를 계산합니다.

코드에 점프 대상이 있고 *jump*가 `True`이면, `stack_effect()`는 점프의 스택 효과를 반환합니다. *jump*가 `False`이면, 점프하지 않는 스택 효과를 반환합니다. *jump*가 `None`(기본값)이면, 두 경우의 최대 스택 효과를 반환합니다.

Added in version 3.4.

버전 3.8에서 변경: *jump* 매개 변수를 추가했습니다.

버전 3.13에서 변경: If `oparg` is omitted (or `None`), the stack effect is now returned for `oparg=0`. Previously this was an error for opcodes that use their arg. It is also no longer an error to pass an integer `oparg` when the `opcode` does not use it; the `oparg` in this case is ignored.

32.10.4 파이썬 바이트 코드 명령어

`get_instructions()` 함수와 `Bytecode` 클래스는 바이트 코드 명령어의 세부 사항을 `Instruction` 인스턴스로 제공합니다:

class dis.**Instruction**

바이트 코드 연산에 대한 세부 사항

opcode

연산의 숫자 코드, 아래 나열된 옵코드 값과 옵코드 모음에 있는 바이트 코드 값에 해당합니다.

opname

연산의 사람이 읽을 수 있는 이름

baseopcode

numeric code for the base operation if operation is specialized; otherwise equal to `opcode`

baseopname

human readable name for the base operation if operation is specialized; otherwise equal to `opname`

arg

연산에 대한 숫자 인자 (있다면), 그렇지 않으면 `None`

oparg

alias for `arg`

argval

resolved arg value (if any), otherwise `None`

argrepr

human readable description of operation argument (if any), otherwise an empty string.

offset

바이트 코드 시퀀스 내에서 연산의 시작 인덱스

start_offset
start index of operation within bytecode sequence, including prefixed `EXTENDED_ARG` operations if present; otherwise equal to *offset*

cache_offset
start index of the cache entries following the operation

end_offset
end index of the cache entries following the operation

starts_line
`True` if this opcode starts a source line, otherwise `False`

line_number
source line number associated with this opcode (if any), otherwise `None`

is_jump_target
다른 코드가 여기로 점프하면 `True`, 그렇지 않으면 `False`

jump_target
bytecode index of the jump target if this is a jump operation, otherwise `None`

positions
dis.Positions object holding the start and end locations that are covered by this instruction.

cache_info
Information about the cache entries of this instruction, as triplets of the form `(name, size, data)`, where the `name` and `size` describe the cache format and data is the contents of the cache. `cache_info` is `None` if the instruction does not have caches.

Added in version 3.4.

버전 3.11에서 변경: Field `positions` is added.

버전 3.13에서 변경: Changed field `starts_line`.

Added fields `start_offset`, `cache_offset`, `end_offset`, `baseopname`, `baseopcode`, `jump_target`, `oparg`, `line_number` and `cache_info`.

class dis.**Positions**
In case the information is not available, some fields might be `None`.

lineno

end_lineno

col_offset

end_col_offset

Added in version 3.11.

파이썬 컴파일러는 현재 다음 바이트 코드 명령어를 생성합니다.

일반 명령어

In the following, We will refer to the interpreter stack as `STACK` and describe operations on it as if it was a Python list. The top of the stack corresponds to `STACK[-1]` in this language.

NOP
Do nothing code. Used as a placeholder by the bytecode optimizer, and to generate line tracing events.

NOT_TAKEN
Do nothing code. Used by the interpreter to record *BRANCH_LEFT* and *BRANCH_RIGHT* events for *sys.monitoring*.

Added in version 3.14.

POP_ITER

Removes the iterator from the top of the stack.

Added in version 3.14.

POP_TOP

Removes the top-of-stack item:

```
STACK.pop()
```

END_FOR

Removes the top-of-stack item. Equivalent to `POP_TOP`. Used to clean up at the end of loops, hence the name.

Added in version 3.12.

END_SEND

Implements `del STACK[-2]`. Used to clean up when a generator exits.

Added in version 3.12.

COPY (*i*)

Push the i-th item to the top of the stack without removing it from its original location:

```
assert i > 0
STACK.append(STACK[-i])
```

Added in version 3.11.

SWAP (*i*)

Swap the top of the stack with the i-th element:

```
STACK[-i], STACK[-1] = STACK[-1], STACK[-i]
```

Added in version 3.11.

CACHE

Rather than being an actual instruction, this opcode is used to mark extra space for the interpreter to cache useful data directly in the bytecode itself. It is automatically hidden by all `dis` utilities, but can be viewed with `show_caches=True`.

Logically, this space is part of the preceding instruction. Many opcodes expect to be followed by an exact number of caches, and will instruct the interpreter to skip over them at runtime.

Populated caches can look like arbitrary instructions, so great care should be taken when reading or modifying raw, adaptive bytecode containing quickened data.

Added in version 3.11.

단항 연산

단항 연산은 스택의 최상단을 취하고, 연산을 적용한 다음, 결과를 스택에 다시 푸시합니다.

UNARY_NEGATIVE

Implements `STACK[-1] = -STACK[-1]`.

UNARY_NOT

Implements `STACK[-1] = not STACK[-1]`.

버전 3.13에서 변경: This instruction now requires an exact *bool* operand.

UNARY_INVERT

Implements `STACK[-1] = ~STACK[-1]`.

GET_ITER

Implements `STACK[-1] = iter(STACK[-1])`.

GET_YIELD_FROM_ITER

If `STACK[-1]` is a *generator iterator* or *coroutine* object it is left as is. Otherwise, implements `STACK[-1] = iter(STACK[-1])`.

Added in version 3.5.

TO_BOOL

Implements `STACK[-1] = bool(STACK[-1])`.

Added in version 3.13.

Binary and in-place operations

Binary operations remove the top two items from the stack (`STACK[-1]` and `STACK[-2]`). They perform the operation, then put the result back on the stack.

In-place operations are like binary operations, but the operation is done in-place when `STACK[-2]` supports it, and the resulting `STACK[-1]` may be (but does not have to be) the original `STACK[-2]`.

BINARY_OP (*op*)

Implements the binary and in-place operators (depending on the value of *op*):

```
rhs = STACK.pop()
lhs = STACK.pop()
STACK.append(lhs op rhs)
```

Added in version 3.11.

버전 3.14 에서 변경: With oparg :`NB_SUBSCR`, implements binary subscript (replaces opcode `BINARY_SUBSCR`)

STORE_SUBSCR

Implements:

```
key = STACK.pop()
container = STACK.pop()
value = STACK.pop()
container[key] = value
```

DELETE_SUBSCR

Implements:

```
key = STACK.pop()
container = STACK.pop()
del container[key]
```

BINARY_SLICE

Implements:

```
end = STACK.pop()
start = STACK.pop()
container = STACK.pop()
STACK.append(container[start:end])
```

Added in version 3.12.

STORE_SLICE

Implements:

```
end = STACK.pop()
start = STACK.pop()
container = STACK.pop()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
values = STACK.pop()
container[start:end] = value
```

Added in version 3.12.

코루틴 옵코드

GET_AWAITABLE (*where*)

Implements `STACK[-1] = get_awaitable(STACK[-1])`, where `get_awaitable(o)` returns `o` if `o` is a coroutine object or a generator object with the *CO_ITERABLE_COROUTINE* flag, or resolves `o.__await__`.

If the `where` operand is nonzero, it indicates where the instruction occurs:

- `1`: After a call to `__aenter__`
- `2`: After a call to `__aexit__`

Added in version 3.5.

버전 3.11에서 변경: Previously, this instruction did not have an oparg.

GET_AITER

Implements `STACK[-1] = STACK[-1].__aiter__()`.

Added in version 3.5.

버전 3.7에서 변경: `__aiter__`로부터 어웨이터블 객체를 반환하는 것은 더는 지원되지 않습니다.

GET_ANEXT

Implement `STACK.append(get_awaitable(STACK[-1].__anext__()))` to the stack. See `GET_AWAITABLE` for details about `get_awaitable`.

Added in version 3.5.

END_ASYNC_FOR

Terminates an `async for` loop. Handles an exception raised when awaiting a next item. The stack contains the async iterable in `STACK[-2]` and the raised exception in `STACK[-1]`. Both are popped. If the exception is not *StopAsyncIteration*, it is re-raised.

Added in version 3.8.

버전 3.11에서 변경: Exception representation on the stack now consist of one, not three, items.

CLEANUP_THROW

Handles an exception raised during a `throw()` or `close()` call through the current frame. If `STACK[-1]` is an instance of *StopIteration*, pop three values from the stack and push its `value` member. Otherwise, re-raise `STACK[-1]`.

Added in version 3.12.

기타 옵코드

SET_ADD (*i*)

Implements:

```
item = STACK.pop()
set.add(STACK[-i], item)
```

Used to implement set comprehensions.

LIST_APPEND (*i*)

Implements:

```
item = STACK.pop()
list.append(STACK[-i], item)
```

Used to implement list comprehensions.

MAP_ADD (*i*)

Implements:

```
value = STACK.pop()
key = STACK.pop()
dict.__setitem__(STACK[-i], key, value)
```

Used to implement dict comprehensions.

Added in version 3.1.

버전 3.8에서 변경: Map value is `STACK[-1]` and map key is `STACK[-2]`. Before, those were reversed.

모든 *SET_ADD*, *LIST_APPEND* 및 *MAP_ADD* 명령어에 대해, 추가된 값이나 키/값 쌍이 팝 되지만, 컨테이너 객체는 스택에 남아 있어서 루프의 추가 이터레이션에 사용할 수 있습니다.

RETURN_VALUE

Returns with `STACK[-1]` to the caller of the function.

YIELD_VALUE

Yields `STACK.pop()` from a *generator*.

버전 3.11에서 변경: oparg set to be the stack depth.

버전 3.12에서 변경: oparg set to be the exception block depth, for efficient closing of generators.

버전 3.13에서 변경: oparg is `1` if this instruction is part of a yield-from or await, and `0` otherwise.

SETUP_ANNOTATIONS

`locals()`에 `__annotations__`가 정의되어 있는지 확인합니다, 그렇지 않으면 비어있는 `dict`로 설정됩니다. 이 옵코드는 클래스나 모듈 본문에 변수 어노테이션이 정적으로 포함될 때만 생성됩니다.

Added in version 3.6.

POP_EXCEPT

Pops a value from the stack, which is used to restore the exception state.

버전 3.11에서 변경: Exception representation on the stack now consist of one, not three, items.

RERAISE

Re-raises the exception currently on top of the stack. If oparg is non-zero, pops an additional value from the stack which is used to set `f_lasti` of the current frame.

Added in version 3.9.

버전 3.11에서 변경: Exception representation on the stack now consist of one, not three, items.

PUSH_EXC_INFO

Pops a value from the stack. Pushes the current exception to the top of the stack. Pushes the value originally popped back to the stack. Used in exception handlers.

Added in version 3.11.

CHECK_EXC_MATCH

Performs exception matching for `except`. Tests whether the `STACK[-2]` is an exception matching `STACK[-1]`. Pops `STACK[-1]` and pushes the boolean result of the test.

Added in version 3.11.

CHECK_EG_MATCH

Performs exception matching for `except*`. Applies `split(STACK[-1])` on the exception group representing `STACK[-2]`.

In case of a match, pops two items from the stack and pushes the non-matching subgroup (`None` in case of full match) followed by the matching subgroup. When there is no match, pops one item (the match type) and pushes `None`.

Added in version 3.11.

WITH_EXCEPT_START

Calls the function in position 4 on the stack with arguments (type, val, tb) representing the exception at the top of the stack. Used to implement the call `context_manager.__exit__(*exc_info())` when an exception has occurred in a `with` statement.

Added in version 3.9.

버전 3.11에서 변경: The `__exit__` function is in position 4 of the stack rather than 7. Exception representation on the stack now consist of one, not three, items.

LOAD_COMMON_CONSTANT

Pushes a common constant onto the stack. The interpreter contains a hardcoded list of constants supported by this instruction. Used by the `assert` statement to load `AssertionError`.

Added in version 3.14.

LOAD_BUILD_CLASS

Pushes `builtins.__build_class__()` onto the stack. It is later called to construct a class.

GET_LEN

Perform `STACK.append(len(STACK[-1]))`. Used in `match` statements where comparison with structure of pattern is needed.

Added in version 3.10.

MATCH_MAPPING

If `STACK[-1]` is an instance of `collections.abc.Mapping` (or, more technically: if it has the `Py_TPFLAGS_MAPPING` flag set in its `tp_flags`), push `True` onto the stack. Otherwise, push `False`.

Added in version 3.10.

MATCH_SEQUENCE

If `STACK[-1]` is an instance of `collections.abc.Sequence` and is *not* an instance of `str`/`bytes`/`bytearray` (or, more technically: if it has the `Py_TPFLAGS_SEQUENCE` flag set in its `tp_flags`), push `True` onto the stack. Otherwise, push `False`.

Added in version 3.10.

MATCH_KEYS

`STACK[-1]` is a tuple of mapping keys, and `STACK[-2]` is the match subject. If `STACK[-2]` contains all of the keys in `STACK[-1]`, push a `tuple` containing the corresponding values. Otherwise, push `None`.

Added in version 3.10.

버전 3.11에서 변경: Previously, this instruction also pushed a boolean value indicating success (`True`) or failure (`False`).

STORE_NAME (*namei*)

Implements `name = STACK.pop()`. *namei* is the index of *name* in the attribute `co_names` of the code object. The compiler tries to use `STORE_FAST` or `STORE_GLOBAL` if possible.

DELETE_NAME (*namei*)

Implements `del name`, where *namei* is the index into `co_names` attribute of the code object.

UNPACK_SEQUENCE (*count*)

Unpacks `STACK[-1]` into *count* individual values, which are put onto the stack right-to-left. Require there to be exactly *count* values.:

```
assert(len(STACK[-1]) == count)
STACK.extend(STACK.pop()[:-count-1:-1])
```

UNPACK_EX (*counts*)

Implements assignment with a starred target: Unpacks an iterable in `STACK[-1]` into individual values, where the total number of values can be smaller than the number of items in the iterable: one of the new values will be a list of all leftover items.

The number of values before and after the list value is limited to 255.

The number of values before the list value is encoded in the argument of the opcode. The number of values after the list if any is encoded using an `EXTENDED_ARG`. As a consequence, the argument can be seen as a two bytes values where the low byte of *counts* is the number of values before the list value, the high byte of *counts* the number of values after it.

The extracted values are put onto the stack right-to-left, i.e. `a, *b, c = d` will be stored after execution as `STACK.extend((a, b, c))`.

STORE_ATTR (*namei*)

Implements:

```
obj = STACK.pop()
value = STACK.pop()
obj.name = value
```

where *namei* is the index of name in `co_names` of the code object.

DELETE_ATTR (*namei*)

Implements:

```
obj = STACK.pop()
del obj.name
```

where *namei* is the index of name into `co_names` of the code object.

STORE_GLOBAL (*namei*)

*STORE_NAME*처럼 작동하지만, 이름을 전역으로 저장합니다.

DELETE_GLOBAL (*namei*)

*DELETE_NAME*처럼 작동하지만, 전역 이름을 삭제합니다.

LOAD_CONST (*consti*)

`co_consts[consti]`를 스택으로 푸시합니다.

LOAD_SMALL_INT (*i*)

Pushes the integer `i` onto the stack. `i` must be in `range(256)`

Added in version 3.14.

LOAD_NAME (*namei*)

Pushes the value associated with `co_names[namei]` onto the stack. The name is looked up within the locals, then the globals, then the builtins.

LOAD_LOCALS

Pushes a reference to the locals dictionary onto the stack. This is used to prepare namespace dictionaries for *LOAD_FROM_DICT_OR_DEREF* and *LOAD_FROM_DICT_OR_GLOBALS*.

Added in version 3.12.

LOAD_FROM_DICT_OR_GLOBALS (*i*)

Pops a mapping off the stack and looks up the value for `co_names[namei]`. If the name is not found there, looks it up in the globals and then the builtins, similar to *LOAD_GLOBAL*. This is used for loading global variables in annotation scopes within class bodies.

Added in version 3.12.

BUILD_TEMPLATE

Constructs a new *Template* instance from a tuple of strings and a tuple of interpolations and pushes the resulting object onto the stack:

```
interpolations = STACK.pop()
strings = STACK.pop()
STACK.append(_build_template(strings, interpolations))
```

Added in version 3.14.

BUILD_INTERPOLATION (*format*)

Constructs a new *Interpolation* instance from a value and its source expression and pushes the resulting object onto the stack.

If no conversion or format specification is present, `format` is set to `2`.

If the low bit of `format` is set, it indicates that the interpolation contains a format specification.

If `format >> 2` is non-zero, it indicates that the interpolation contains a conversion. The value of `format >> 2` is the conversion type (`0` for no conversion, `1` for `!s`, `2` for `!r`, and `3` for `!a`):

```
conversion = format >> 2
if format & 1:
    format_spec = STACK.pop()
else:
    format_spec = None
expression = STACK.pop()
value = STACK.pop()
STACK.append(_build_interpolation(value, expression, conversion, format_spec))
```

Added in version 3.14.

BUILD_TUPLE (*count*)

Creates a tuple consuming *count* items from the stack, and pushes the resulting tuple onto the stack:

```
if count == 0:
    value = ()
else:
    value = tuple(STACK[-count:])
    STACK = STACK[:-count]

STACK.append(value)
```

BUILD_LIST (*count*)

*BUILD_TUPLE*처럼 작동하지만, 리스트를 만듭니다.

BUILD_SET (*count*)

*BUILD_TUPLE*처럼 작동하지만, 집합을 만듭니다.

BUILD_MAP (*count*)

Pushes a new dictionary object onto the stack. Pops `2 * count` items so that the dictionary holds *count* entries: `{..., STACK[-4]: STACK[-3], STACK[-2]: STACK[-1]}`.

버전 3.5에서 변경: 딕셔너리는 *count* 항목을 갖도록 미리 크기가 조정된 빈 딕셔너리를 만드는 대신 스택 항목에서 만들어집니다.

BUILD_STRING (*count*)

스택에서 *count* 문자열을 이어붙이고 결과 문자열을 스택으로 푸시합니다.

Added in version 3.6.

LIST_EXTEND (*i*)

Implements:

```
seq = STACK.pop()
list.extend(STACK[-i], seq)
```

Used to build lists.

Added in version 3.9.

SET_UPDATE (*i*)

Implements:

```
seq = STACK.pop()
set.update(STACK[-i], seq)
```

Used to build sets.

Added in version 3.9.

DICT_UPDATE (*i*)

Implements:

```
map = STACK.pop()
dict.update(STACK[-i], map)
```

Used to build dicts.

Added in version 3.9.

DICT_MERGE (*i*)

*DICT_UPDATE*와 유사하지만, 중복 키에 대해 예외를 발생시킵니다.

Added in version 3.9.

LOAD_ATTR (*namei*)

If the low bit of `namei` is not set, this replaces `STACK[-1]` with `getattr(STACK[-1], co_names[namei>>1])`.

If the low bit of `namei` is set, this will attempt to load a method named `co_names[namei>>1]` from the `STACK[-1]` object. `STACK[-1]` is popped. This bytecode distinguishes two cases: if `STACK[-1]` has a method with the correct name, the bytecode pushes the unbound method and `STACK[-1]`. `STACK[-1]` will be used as the first argument (`self`) by *CALL* or *CALL_KW* when calling the unbound method. Otherwise, `NULL` and the object returned by the attribute lookup are pushed.

버전 3.12에서 변경: If the low bit of `namei` is set, then a `NULL` or `self` is pushed to the stack before the attribute or unbound method respectively.

LOAD_SUPER_ATTR (*namei*)

This opcode implements *super()*, both in its zero-argument and two-argument forms (e.g. `super().method()`, `super().attr` and `super(cls, self).method()`, `super(cls, self).attr`).

It pops three values from the stack (from top of stack down):

- `self`: the first argument to the current method
- `cls`: the class within which the current method was defined
- the global `super`

With respect to its argument, it works similarly to *LOAD_ATTR*, except that `namei` is shifted left by 2 bits instead of 1.

The low bit of `namei` signals to attempt a method load, as with *LOAD_ATTR*, which results in pushing `NULL` and the loaded method. When it is unset a single value is pushed to the stack.

The second-low bit of `namei`, if set, means that this was a two-argument call to `super()` (unset means zero-argument).

Added in version 3.12.

COMPARE_OP (*opname*)

Performs a Boolean operation. The operation name can be found in `cmp_op[opname >> 5]`. If the fifth-lowest bit of `opname` is set (`opname & 16`), the result should be coerced to `bool`.

버전 3.13에서 변경: The fifth-lowest bit of the oparg now indicates a forced conversion to `bool`.

IS_OP (*invert*)

`is` 비교를 수행하거나, `invert`가 1이면 `is not`을 수행합니다.

Added in version 3.9.

CONTAINS_OP (*invert*)

`in` 비교를 수행하거나, `invert`가 1이면 `not in`을 수행합니다.

Added in version 3.9.

IMPORT_NAME (*namei*)

Imports the module `co_names[namei]`. `STACK[-1]` and `STACK[-2]` are popped and provide the *fromlist* and *level* arguments of `__import__()`. The module object is pushed onto the stack. The current namespace is not affected: for a proper import statement, a subsequent `STORE_FAST` instruction modifies the namespace.

IMPORT_FROM (*namei*)

Loads the attribute `co_names[namei]` from the module found in `STACK[-1]`. The resulting object is pushed onto the stack, to be subsequently stored by a `STORE_FAST` instruction.

JUMP_FORWARD (*delta*)

바이트 코드 카운터를 *delta*만큼 증가시킵니다.

JUMP_BACKWARD (*delta*)

Decrements bytecode counter by *delta*. Checks for interrupts.

Added in version 3.11.

JUMP_BACKWARD_NO_INTERRUPT (*delta*)

Decrements bytecode counter by *delta*. Does not check for interrupts.

Added in version 3.11.

POP_JUMP_IF_TRUE (*delta*)

If `STACK[-1]` is true, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

버전 3.11에서 변경: The oparg is now a relative delta rather than an absolute target. This opcode is a pseudo-instruction, replaced in final bytecode by the directed versions (forward/backward).

버전 3.12에서 변경: This is no longer a pseudo-instruction.

버전 3.13에서 변경: This instruction now requires an exact `bool` operand.

POP_JUMP_IF_FALSE (*delta*)

If `STACK[-1]` is false, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

버전 3.11에서 변경: The oparg is now a relative delta rather than an absolute target. This opcode is a pseudo-instruction, replaced in final bytecode by the directed versions (forward/backward).

버전 3.12에서 변경: This is no longer a pseudo-instruction.

버전 3.13에서 변경: This instruction now requires an exact `bool` operand.

POP_JUMP_IF_NOT_NONE(*delta*)

If `STACK[-1]` is not `None`, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

Added in version 3.11.

버전 3.12에서 변경: This is no longer a pseudo-instruction.

POP_JUMP_IF_NONE(*delta*)

If `STACK[-1]` is `None`, increments the bytecode counter by *delta*. `STACK[-1]` is popped.

Added in version 3.11.

버전 3.12에서 변경: This is no longer a pseudo-instruction.

FOR_ITER(*delta*)

`STACK[-1]` is an *iterator*. Call its `__next__()` method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted then the byte code counter is incremented by *delta*.

버전 3.12에서 변경: Up until 3.11 the iterator was popped when it was exhausted.

LOAD_GLOBAL(*namei*)

Loads the global named `co_names[namei>>1]` onto the stack.

버전 3.11에서 변경: If the low bit of `namei` is set, then a `NULL` is pushed to the stack before the global variable.

LOAD_FAST(*var_num*)

지역 `co_varnames[var_num]` 에 대한 참조를 스택으로 푸시합니다.

버전 3.12에서 변경: This opcode is now only used in situations where the local variable is guaranteed to be initialized. It cannot raise `UnboundLocalError`.

LOAD_FAST_BORROW(*var_num*)

Pushes a borrowed reference to the local `co_varnames[var_num]` onto the stack.

Added in version 3.14.

LOAD_FAST_LOAD_FAST(*var_nums*)

Pushes references to `co_varnames[var_nums >> 4]` and `co_varnames[var_nums & 15]` onto the stack.

Added in version 3.13.

LOAD_FAST_BORROW_LOAD_FAST_BORROW(*var_nums*)

Pushes borrowed references to `co_varnames[var_nums >> 4]` and `co_varnames[var_nums & 15]` onto the stack.

Added in version 3.14.

LOAD_FAST_CHECK(*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack, raising an `UnboundLocalError` if the local variable has not been initialized.

Added in version 3.12.

LOAD_FAST_AND_CLEAR(*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack (or pushes `NULL` onto the stack if the local variable has not been initialized) and sets `co_varnames[var_num]` to `NULL`.

Added in version 3.12.

STORE_FAST(*var_num*)

Stores `STACK.pop()` into the local `co_varnames[var_num]`.

STORE_FAST_STORE_FAST (*var_nums*)

Stores `STACK[-1]` into `co_varnames[var_nums >> 4]` and `STACK[-2]` into `co_varnames[var_nums & 15]`.

Added in version 3.13.

STORE_FAST_LOAD_FAST (*var_nums*)

Stores `STACK.pop()` into the local `co_varnames[var_nums >> 4]` and pushes a reference to the local `co_varnames[var_nums & 15]` onto the stack.

Added in version 3.13.

DELETE_FAST (*var_num*)

지역 `co_varnames[var_num]`을 삭제합니다.

MAKE_CELL (*i*)

Creates a new cell in slot `i`. If that slot is nonempty then that value is stored into the new cell.

Added in version 3.11.

LOAD_DEREF (*i*)

Loads the cell contained in slot `i` of the "fast locals" storage. Pushes a reference to the object the cell contains on the stack.

버전 3.11에서 변경: `i` is no longer offset by the length of `co_varnames`.

LOAD_FROM_DICT_OR_DEREF (*i*)

Pops a mapping off the stack and looks up the name associated with slot `i` of the "fast locals" storage in this mapping. If the name is not found there, loads it from the cell contained in slot `i`, similar to *LOAD_DEREF*. This is used for loading *closure variables* in class bodies (which previously used `LOAD_CLASSDEREF`) and in annotation scopes within class bodies.

Added in version 3.12.

STORE_DEREF (*i*)

Stores `STACK.pop()` into the cell contained in slot `i` of the "fast locals" storage.

버전 3.11에서 변경: `i` is no longer offset by the length of `co_varnames`.

DELETE_DEREF (*i*)

Empties the cell contained in slot `i` of the "fast locals" storage. Used by the `del` statement.

Added in version 3.2.

버전 3.11에서 변경: `i` is no longer offset by the length of `co_varnames`.

COPY_FREE_VARS (*n*)

Copies the `n` *free (closure) variables* from the closure into the frame. Removes the need for special code on the caller' s side when calling closures.

Added in version 3.11.

RAISE_VARARGS (*argc*)

*argc*의 값에 따라, `raise` 문의 3가지 형식 중 하나를 사용하여 예외를 발생시킵니다:

- 0: `raise` (이전 예외를 다시 발생시킵니다)
- 1: `raise STACK[-1]` (raise exception instance or type at `STACK[-1]`)
- 2: `raise STACK[-2] from STACK[-1]` (raise exception instance or type at `STACK[-2]` with `__cause__` set to `STACK[-1]`)

CALL (*argc*)

Calls a callable object with the number of arguments specified by `argc`. On the stack are (in ascending order):

- The callable
- `self` or `NULL`

- The remaining positional arguments

`argc` is the total of the positional arguments, excluding `self`.

`CALL` pops all arguments and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Added in version 3.11.

버전 3.13에서 변경: The callable now always appears at the same position on the stack.

버전 3.13에서 변경: Calls with keyword arguments are now handled by *CALL_KW*.

CALL_KW (*argc*)

Calls a callable object with the number of arguments specified by `argc`, including one or more named arguments. On the stack are (in ascending order):

- The callable
- `self` or `NULL`
- The remaining positional arguments
- The named arguments
- A *tuple* of keyword argument names

`argc` is the total of the positional and named arguments, excluding `self`. The length of the tuple of keyword argument names is the number of named arguments.

`CALL_KW` pops all arguments, the keyword names, and the callable object off the stack, calls the callable object with those arguments, and pushes the return value returned by the callable object.

Added in version 3.13.

CALL_FUNCTION_EX (*flags*)

위치와 키워드 인자의 변수 집합으로 콜러블 객체를 호출합니다. *flags*의 최하위 비트가 설정되면, 스택의 맨 위에 추가 키워드 인자가 포함된 매핑 객체가 포함됩니다. 콜러블이 호출되기 전에, 매핑 객체와 이터러블 객체는 각각 "언팩" 되고 그 내용이 각각 키워드와 위치 인자로 전달됩니다. `CALL_FUNCTION_EX`는 모든 인자와 콜러블 객체를 스택에서 팝하고, 해당 인자로 콜러블 객체를 호출한 다음, 콜러블 객체가 반환한 반환 값을 푸시합니다.

Added in version 3.6.

PUSH_NULL

Pushes a `NULL` to the stack. Used in the call sequence to match the `NULL` pushed by `LOAD_METHOD` for non-method calls.

Added in version 3.11.

MAKE_FUNCTION

Pushes a new function object on the stack built from the code object at `STACK[-1]`.

버전 3.10에서 변경: Flag value `0x04` is a tuple of strings instead of dictionary

버전 3.11에서 변경: Qualified name at `STACK[-1]` was removed.

버전 3.13에서 변경: Extra function attributes on the stack, signaled by oparg flags, were removed. They now use *SET_FUNCTION_ATTRIBUTE*.

SET_FUNCTION_ATTRIBUTE (*flag*)

Sets an attribute on a function object. Expects the function at `STACK[-1]` and the attribute value to set at `STACK[-2]`; consumes both and leaves the function at `STACK[-1]`. The flag determines which attribute to set:

- `0x01` 위치 전용과 위치-키워드 매개 변수를 위한 기본값의 위치 순서 튜플
- `0x02` 키워드 전용 매개 변수의 기본값 딕셔너리
- `0x04` a tuple of strings containing parameters' annotations

- `0x08` 자유 변수를 위한 셀을 포함하는 튜플, 클로저를 만듭니다

Added in version 3.13.

BUILD_SLICE (*argc*)

Pushes a slice object on the stack. *argc* must be 2 or 3. If it is 2, implements:

```
end = STACK.pop()
start = STACK.pop()
STACK.append(slice(start, end))
```

if it is 3, implements:

```
step = STACK.pop()
end = STACK.pop()
start = STACK.pop()
STACK.append(slice(start, end, step))
```

See the `slice()` built-in function for more information.

EXTENDED_ARG (*ext*)

너무 커서 기본 1바이트에 맞지 않는 인자를 가진 옵코드에 접두어로 붙입니다. *ext*는 인자에서 더 높은 비트로 작동하는 추가 바이트를 보유합니다. 각 옵코드마다, 최대 3개의 접두사 `EXTENDED_ARG` 가 허용되며, 2바이트에서 4바이트 사이의 인자를 형성합니다.

CONVERT_VALUE (*oparg*)

Convert value to a string, depending on `oparg`:

```
value = STACK.pop()
result = func(value)
STACK.append(result)
```

- `oparg == 1`: call `str()` on *value*
- `oparg == 2`: call `repr()` on *value*
- `oparg == 3`: call `ascii()` on *value*

Used for implementing formatted string literals (f-strings).

Added in version 3.13.

FORMAT_SIMPLE

Formats the value on top of stack:

```
value = STACK.pop()
result = value.__format__("")
STACK.append(result)
```

Used for implementing formatted string literals (f-strings).

Added in version 3.13.

FORMAT_WITH_SPEC

Formats the given value with the given format spec:

```
spec = STACK.pop()
value = STACK.pop()
result = value.__format__(spec)
STACK.append(result)
```

Used for implementing formatted string literals (f-strings).

Added in version 3.13.

MATCH_CLASS (*count*)

`STACK[-1]` is a tuple of keyword attribute names, `STACK[-2]` is the class being matched against, and `STACK[-3]` is the match subject. *count* is the number of positional sub-patterns.

Pop `STACK[-1]`, `STACK[-2]`, and `STACK[-3]`. If `STACK[-3]` is an instance of `STACK[-2]` and has the positional and keyword attributes required by *count* and `STACK[-1]`, push a tuple of extracted attributes. Otherwise, push `None`.

Added in version 3.10.

버전 3.11에서 변경: Previously, this instruction also pushed a boolean value indicating success (`True`) or failure (`False`).

RESUME (*context*)

A no-op. Performs internal tracing, debugging and optimization checks.

The `context` operand consists of two parts. The lowest two bits indicate where the `RESUME` occurs:

- `0` The start of a function, which is neither a generator, coroutine nor an async generator
- `1` After a `yield` expression
- `2` After a `yield from` expression
- `3` After an `await` expression

The next bit is `1` if the RESUME is at except-depth `1`, and `0` otherwise.

Added in version 3.11.

버전 3.13에서 변경: The oparg value changed to include information about except-depth

RETURN_GENERATOR

Create a generator, coroutine, or async generator from the current frame. Used as first opcode of in code object for the above mentioned callables. Clear the current frame and return the newly created generator.

Added in version 3.11.

SEND (*delta*)

Equivalent to `STACK[-1] = STACK[-2].send(STACK[-1])`. Used in `yield from` and `await` statements.

If the call raises `StopIteration`, pop the top value from the stack, push the exception' s `value` attribute, and increment the bytecode counter by *delta*.

Added in version 3.11.

HAVE_ARGUMENT

This is not really an opcode. It identifies the dividing line between opcodes in the range [0,255] which don' t use their argument and those that do (`< HAVE_ARGUMENT` and `>= HAVE_ARGUMENT`, respectively).

If your application uses pseudo instructions or specialized instructions, use the `hasarg` collection instead.

버전 3.6에서 변경: 이제 모든 명령어에는 인자가 있지만, `< HAVE_ARGUMENT`인 옵코드는 이를 무시합니다. 이전에는, `>= HAVE_ARGUMENT`인 옵코드에만 인자가 있었습니다.

버전 3.12에서 변경: Pseudo instructions were added to the `dis` module, and for them it is not true that comparison with `HAVE_ARGUMENT` indicates whether they use their arg.

버전 3.13부터 폐지됨: Use `hasarg` instead.

CALL_INTRINSIC_1

Calls an intrinsic function with one argument. Passes `STACK[-1]` as the argument and sets `STACK[-1]` to the result. Used to implement functionality that is not performance critical.

The operand determines which intrinsic function is called:

Operand	Description
INTRINSIC_1_INVAL	Not valid
INTRINSIC_PRINT	Prints the argument to standard out. Used in the REPL.
INTRINSIC_IMPORT_	Performs `import *` for the named module.
INTRINSIC_STOPITE	Extracts the return value from a `StopIteration` exception.
INTRINSIC_ASYNC_G	Wraps an async generator value
INTRINSIC_UNARY_P	Performs the unary + operation
INTRINSIC_LIST_TO	Converts a list to a tuple
INTRINSIC_TYPEVAR	Creates a *typing.TypeVar*
INTRINSIC_PARAMSP	Creates a *typing.ParamSpec*
INTRINSIC_TYPEVAR	Creates a *typing.TypeVarTuple*
INTRINSIC_SUBSCRI	Returns *typing.Generic* subscripted with the argument
INTRINSIC_TYPEALI	Creates a *typing.TypeAliasType*; used in the `type` statement. The argument is a tuple of the type alias' s name, type parameters, and value.

Added in version 3.12.

CALL_INTRINSIC_2

Calls an intrinsic function with two arguments. Used to implement functionality that is not performance critical:

```
arg2 = STACK.pop()
arg1 = STACK.pop()
result = intrinsic2(arg1, arg2)
STACK.append(result)
```

The operand determines which intrinsic function is called:

Operand	Description
INTRINSIC_2_INVALID	Not valid
INTRINSIC_PREP_RERAISE_STAR	Calculates the *ExceptionGroup* to raise from a `try-except*`.
INTRINSIC_TYPEVAR_WITH_BOUND	Creates a *typing.TypeVar* with a bound.
INTRINSIC_TYPEVAR_WITH_CONSTRAIN	Creates a *typing.TypeVar* with constraints.
INTRINSIC_SET_FUNCTION_TYPE_PARA	Sets the `__type_params__` attribute of a function.

Added in version 3.12.

LOAD_SPECIAL

Performs special method lookup on `STACK[-1]`. If `type(STACK[-1]).__xxx__` is a method, leave `type(STACK[-1]).__xxx__; STACK[-1]` on the stack. If `type(STACK[-1]).__xxx__` is not a method, leave `STACK[-1].__xxx__; NULL` on the stack.

Added in version 3.14.

Pseudo-instructions

These opcodes do not appear in Python bytecode. They are used by the compiler but are replaced by real opcodes or removed before bytecode is generated.

SETUP_FINALLY (*target*)

Set up an exception handler for the following code block. If an exception occurs, the value stack level is restored to its current state and control is transferred to the exception handler at `target`.

SETUP_CLEANUP (*target*)

Like `SETUP_FINALLY`, but in case of an exception also pushes the last instruction (`lasti`) to the stack so that `RERAISE` can restore it. If an exception occurs, the value stack level and the last instruction on the frame are restored to their current state, and control is transferred to the exception handler at `target`.

SETUP_WITH (*target*)

Like `SETUP_CLEANUP`, but in case of an exception one more item is popped from the stack before control is transferred to the exception handler at `target`.

This variant is used in `with` and `async with` constructs, which push the return value of the context manager's `__enter__()` or `__aenter__()` to the stack.

POP_BLOCK

Marks the end of the code block associated with the last `SETUP_FINALLY`, `SETUP_CLEANUP` or `SETUP_WITH`.

LOAD_CONST_IMMORTAL (*consti*)

Works as *`LOAD_CONST`*, but is more efficient for immortal objects.

JUMP

JUMP_NO_INTERRUPT

Undirected relative jump instructions which are replaced by their directed (forward/backward) counterparts by the assembler.

JUMP_IF_TRUE

JUMP_IF_FALSE

Conditional jumps which do not impact the stack. Replaced by the sequence `COPY 1`, `TO_BOOL`, `POP_JUMP_IF_TRUE/FALSE`.

LOAD_CLOSURE (*i*)

Pushes a reference to the cell contained in slot `i` of the "fast locals" storage.

Note that `LOAD_CLOSURE` is replaced with `LOAD_FAST` in the assembler.

버전 3.13에서 변경: This opcode is now a pseudo-instruction.

32.10.5 옵코드 모음

이 모음은 바이트 코드 명령어의 자동 검사를 위해 제공됩니다:

버전 3.12에서 변경: The collections now contain pseudo instructions and instrumented instructions as well. These are opcodes with values `>= MIN_PSEUDO_OPCODE` and `>= MIN_INSTRUMENTED_OPCODE`.

`dis.`**opname**

연산 이름의 시퀀스, 바이트 코드를 사용하여 인덱싱할 수 있습니다.

`dis.`**opmap**

연산 이름을 바이트 코드로 매핑하는 딕셔너리.

`dis.`**cmp_op**

모든 비교 연산 이름의 시퀀스.

`dis.`**hasarg**

Sequence of bytecodes that use their argument.

Added in version 3.12.

`dis.`**hasconst**

상수에 액세스하는 바이트 코드의 시퀀스.

`dis.`**hasfree**

Sequence of bytecodes that access a *free (closure) variable*. 'free' in this context refers to names in the current scope that are referenced by inner scopes or names in outer scopes that are referenced from this scope. It does *not* include references to global or builtin scopes.

`dis.`**hasname**

어트리뷰트를 이름으로 액세스하는 바이트 코드의 시퀀스.

dis.**hasjump**

Sequence of bytecodes that have a jump target. All jumps are relative.

Added in version 3.13.

dis.**haslocal**

지역 변수에 액세스하는 바이트 코드의 시퀀스.

dis.**hascompare**

불리언 연산의 바이트 코드의 시퀀스.

dis.**hasexc**

Sequence of bytecodes that set an exception handler.

Added in version 3.12.

dis.**hasjrel**

상대 점프 대상이 있는 바이트 코드의 시퀀스.

버전 3.13부터 폐지됨: All jumps are now relative. Use *hasjump*.

dis.**hasjabs**

절대 점프 대상이 있는 바이트 코드의 시퀀스.

버전 3.13부터 폐지됨: All jumps are now relative. This list is empty.

32.11 pickletools — 피클 개발자를 위한 도구

소스 코드: Lib/pickletools.py

이 모듈은 *pickle* 모듈의 깊은 세부 사항과 관련된 다양한 상수, 구현에 대한 긴 주석, 그리고 피클 된 데이터를 분석하기 위한 몇 가지 유용한 함수를 포함합니다. 이 모듈의 내용은 *pickle*에서 작업하는 파이썬 코어 개발자에게 유용합니다; 아마도 *pickle* 모듈의 일반 사용자는 *pickletools* 모듈을 적절한 용도를 찾지 못할 것입니다.

32.11.1 Command-line usage

Added in version 3.2.

명령 줄에서 호출될 때, `python -m pickletools`는 하나 이상의 피클 파일의 내용을 역 어셈블합니다. 피클 형식의 세부 사항이 아닌 피클에 저장된 파이썬 객체를 보려면, 대신 `-m pickle`을 사용하는 것이 좋습니다. 그러나, 검사하려는 피클 파일이 신뢰할 수 없는 소스에서 왔을 때, 피클 바이트 코드를 실행하지 않으므로 `-m pickletools`가 더 안전한 옵션입니다.

예를 들어, 튜플 `(1, 2)`가 파일 `x.pickle`에 피클 된 경우:

```
$ python -m pickle x.pickle
(1, 2)

$ python -m pickletools x.pickle
    0: \x80 PROTO      3
    2: K    BININT1    1
    4: K    BININT1    2
    6: \x86 TUPLE2
    7: q    BINPUT     0
    9: .    STOP
highest protocol among opcodes = 2
```

Command-line options

-a, --annotate

각 줄에 짧은 opcode 설명으로 주석을 답니다.

-o, --output=<file>

출력이 기록되어야 하는 파일의 이름.

-l, --indentlevel=<num>

새 MARK 수준을 들여쓰기하는 공백의 수.

-m, --memo

여러 객체가 역 어셈블될 때, 역 어셈블리 간에 메모를 보존합니다.

-p, --preamble=<preamble>

When more than one pickle file is specified, print given preamble before each disassembly.

pickle_file

A pickle file to read, or - to indicate reading from standard input.

32.11.2 Programmatic interface

pickletools.**dis**(*pickle*, *out=None*, *memo=None*, *indentlevel=4*, *annotate=0*)

피클의 기호적인 역 어셈블리를 기본값이 `sys.stdout` 인 파일류 객체 *out* 으로 출력합니다. *pickle* 는 문자열이나 파일류 객체가 될 수 있습니다. *memo* 는 피클의 메모로 사용될 파이썬 딕셔너리일 수 있습니다; 같은 피클러로 만들어진 여러 피클에 걸쳐 역 어셈블리를 수행하는 데 사용할 수 있습니다. 스트림의 `MARK` 옵코드로 표시된 연속 수준은 *indentlevel* 개의 스페이스로 들여쓰기 됩니다. 0이 아닌 값이 *annotate* 에 주어지면, 출력의 각 옵코드에 짧은 설명이 주석으로 표시됩니다. *annotate* 값은 주석을 시작해야 하는 열의 힌트로 사용됩니다.

버전 3.2에서 변경: *annotate* 인자를 추가했습니다.

pickletools.**genops**(*pickle*)

피클의 모든 옵코드에 대해 `(opcode, arg, pos)` 트리플을 반환하는 이터레이터를 제공합니다. *opcode* 는 `OpcodeInfo` 클래스의 인스턴스입니다; *arg* 는 옵코드 인자의 파이썬 객체로 디코딩된 값입니다; *pos* 는 이 옵코드의 위치입니다. *pickle* 은 문자열이나 파일류 객체가 될 수 있습니다.

pickletools.**optimize**(*picklestring*)

사용되지 않는 `PUT` 옵코드를 제거한 후 새로운 동등한 피클 문자열을 반환합니다. 최적화된 피클은 더 짧고, 전송 시간이 덜 걸리며, 저장 공간이 덜 필요하고, 역 피클이 더 효율적입니다.

CHAPTER 33

MS 윈도우 특정 서비스

이 장에서는 MS 윈도우 플랫폼에서만 사용 가능한 모듈에 관해 설명합니다.

33.1 msvcrt — MS VC++ 런타임의 유용한 루틴

이 함수들은 윈도우 플랫폼에서 유용한 기능에 대한 액세스를 제공합니다. 일부 고수준 모듈은 이러한 함수를 사용하여 해당 서비스의 윈도우 구현을 구축합니다. 예를 들어, *getpass* 모듈은 *getpass()* 함수를 구현할 때 이를 사용합니다.

이 함수에 대한 자세한 설명은 플랫폼 API 설명서에서 찾을 수 있습니다.

이 모듈은 콘솔 I/O api의 일반과 광폭(wide) 문자 변형을 모두 구현합니다. 일반 API는 ASCII 문자만 다루며 국제화된 응용 프로그램에서는 제한적으로 사용됩니다. 가능하면 광폭 문자 API를 사용해야 합니다.

버전 3.3에서 변경: 이 모듈의 연산은 이제 *IOError*를 발생시키던 곳에서 *OSError*를 발생시킵니다.

33.1.1 파일 연산

msvcrt.**locking**(*fd*, *mode*, *nbytes*)

C 런타임의 파일 기술자 *fd*를 기반으로 파일 일부를 잠급니다. 실패하면 *OSError*를 발생시킵니다. 파일의 잠긴 영역은 현재 파일 위치에서부터 *nbytes* 바이트까지며, 파일 끝을 넘어 계속될 수 있습니다. *mode*는 아래에 나열된 `LK_*` 상수 중 하나여야 합니다. 파일의 여러 영역이 동시에 잠길 수 있지만 겹칠 수는 없습니다. 인접한 영역은 병합되지 않습니다; 개별적으로 잠금을 해제해야 합니다.

인자 `fd`, `mode`, `nbytes`로 감사 이벤트 `msvcrt.locking`을 발생시킵니다.

msvcrt.**LK_LOCK**

msvcrt.**LK_RLCK**

지정된 바이트를 잠급니다. 바이트를 잠글 수 없으면, 프로그램은 1초 후에 즉시 다시 시도합니다. 10 번 시도한 후에도 바이트를 잠글 수 없으면, *OSError*가 발생합니다.

msvcrt.**LK_NBLCK**

msvcrt.**LK_NBRLCK**

지정된 바이트를 잠급니다. 바이트를 잠글 수 없으면, *OSError*가 발생합니다.

msvcrt.**LK_UNLCK**

이전에 잠겨 있어야 하는 지정된 바이트의 잠금을 해제합니다.

msvcrt.**setmode**(*fd*, *flags*)

파일 기술자 *fd*의 줄 종료 변환 모드를 설정합니다. 텍스트 모드로 설정하려면, *flags*가 `os.O_TEXT` 여야 합니다; 바이너리는, `os.O_BINARY` 여야 합니다.

msvcrt.**open_osfhandle**(*handle*, *flags*)

파일 핸들 *handle*에서 C 런타임 파일 기술자를 만듭니다. *flags* 매개 변수는 `os.O_APPEND`, `os.O_RDONLY`, `os.O_TEXT` 및 `os.O_NOINHERIT`의 비트별 OR 여야 합니다. 반환된 파일 기술자는 `os.fdopen()`에 대한 매개 변수로 사용되어 파일 객체를 만들 수 있습니다.

The file descriptor is inheritable by default. Pass `os.O_NOINHERIT` flag to make it non inheritable.

인자 `handle`, `flags`로 감사 이벤트 `msvcrt.open_osfhandle`을 발생시킵니다.

msvcrt.**get_osfhandle**(*fd*)

파일 기술자 *fd*의 파일 핸들을 돌려줍니다. *fd*가 인식되지 않으면 `OSError`를 발생시킵니다.

인자 `fd`로 감사 이벤트 `msvcrt.get_osfhandle`을 발생시킵니다.

33.1.2 콘솔 I/O

msvcrt.**kbhit**()

읽을 수 있는 키 누르기가 대기 중이면 0이 아닌 값을 반환합니다. 그렇지 않으면, 0을 반환합니다.

msvcrt.**getch**()

키 누르기를 읽고 결과 문자를 바이트열로 반환합니다. 콘솔에 아무것도 에코 되지 않습니다. 이 호출은 키 누르기를 아직 사용할 수 없으면 블록하지만, `Enter`가 눌러지기를 기다리지는 않습니다. 누른 키가 특수 기능 키면, `'\000'` 이나 `'\xe0'`를 반환합니다; 다음 호출은 키코드를 반환합니다. 이 함수로 `Control-C` 키 누르기를 읽을 수 없습니다.

msvcrt.**getwch**()

유니코드 값을 반환하는 `getch()`의 광폭 문자 변형.

msvcrt.**getche**()

`getch()`와 비슷하지만, 인쇄 가능한 문자를 나타내는 경우 키 누르기가 에코 됩니다.

msvcrt.**getwche**()

유니코드 값을 반환하는 `getche()`의 광폭 문자 변형.

msvcrt.**putch**(*char*)

버퍼링하지 않고 바이트열 *char*을 콘솔에 인쇄합니다.

msvcrt.**putwch**(*unicode_char*)

유니코드 값을 받아들이는 `putch()`의 광폭 문자 변형.

msvcrt.**ungetch**(*char*)

바이트열 *char*이 콘솔 버퍼로 "푸시백" 되도록 합니다; `getch()` 나 `getche()`가 읽는 다음 문자가 됩니다.

msvcrt.**ungetwch**(*unicode_char*)

유니코드 값을 받아들이는 `ungetch()`의 광폭 문자 변형.

33.1.3 기타 함수

msvcrt.**heapmin**()

강제로 `malloc()` 힙이 자신을 정리하고, 사용하지 않는 블록을 운영 체제로 반환하도록 합니다. 실패하면, `OSError`가 발생합니다.

msvcrt.**set_error_mode**(*mode*)

Changes the location where the C runtime writes an error message for an error that might end the program. *mode* must be one of the `OUT_*` constants listed below or `REPORT_ERRMODE`. Returns the old setting or -1 if an error occurs. Only available in debug build of Python.

msvcrt.**OUT_TO_DEFAULT**

Error sink is determined by the app' s type. Only available in debug build of Python.

msvcrt.**OUT_TO_STDERR**

Error sink is a standard error. Only available in debug build of Python.

msvcrt.**OUT_TO_MSGBOX**

Error sink is a message box. Only available in debug build of Python.

msvcrt.**REPORT_ERRMODE**

Report the current error mode value. Only available in debug build of Python.

msvcrt.**CrtSetReportMode**(*type, mode*)

Specifies the destination or destinations for a specific report type generated by `_CrtDbgReport()` in the MS VC++ runtime. *type* must be one of the `CRT_*` constants listed below. *mode* must be one of the `CRTDBG_*` constants listed below. Only available in debug build of Python.

msvcrt.**CrtSetReportFile**(*type, file*)

After you use `CrtSetReportMode()` to specify `CRTDBG_MODE_FILE`, you can specify the file handle to receive the message text. *type* must be one of the `CRT_*` constants listed below. *file* should be the file handle your want specified. Only available in debug build of Python.

msvcrt.**CRT_WARN**

Warnings, messages, and information that doesn' t need immediate attention.

msvcrt.**CRT_ERROR**

Errors, unrecoverable problems, and issues that require immediate attention.

msvcrt.**CRT_ASSERT**

Assertion failures.

msvcrt.**CRTDBG_MODE_DEBUG**

Writes the message to the debugger' s output window.

msvcrt.**CRTDBG_MODE_FILE**

Writes the message to a user-supplied file handle. `CrtSetReportFile()` should be called to define the specific file or stream to use as the destination.

msvcrt.**CRTDBG_MODE_WNDW**

Creates a message box to display the message along with the `Abort`, `Retry`, and `Ignore` buttons.

msvcrt.**CRTDBG_REPORT_MODE**

Returns current *mode* for the specified *type*.

msvcrt.**CRT_ASSEMBLY_VERSION**

The CRT Assembly version, from the `crtassem.h` header file.

msvcrt.**VC_ASSEMBLY_PUBLICKEYTOKEN**

The VC Assembly public key token, from the `crtassem.h` header file.

msvcrt.**LIBRARIES_ASSEMBLY_NAME_PREFIX**

The Libraries Assembly name prefix, from the `crtassem.h` header file.

33.2 winreg — 윈도우 레지스트리 액세스

이 함수들은 윈도우 레지스트리 API를 파이썬에 노출합니다. 프로그래머가 명시적으로 닫는 것을 무시하더라도 핸들이 올바르게 닫히도록 하기 위해, 레지스트리 핸들로 정수를 사용하는 대신 핸들 객체가 사용됩니다.

버전 3.3에서 변경: 이 모듈의 여러 함수는 `WindowsError`를 발생시켜왔는데, 이제는 `OSError`의 별칭입니다.

33.2.1 함수

이 모듈은 다음 함수를 제공합니다:

winreg.**CloseKey**(*hkey*)

이전에 열린 레지스트리 키를 닫습니다. *hkey* 인자는 이전에 열린 키를 지정합니다.

> **참고**
>
> 이 메서드를 사용하여 (또는 `hkey.Close()`를 통해) *hkey*가 닫히지 않으면, *hkey* 객체가 파이썬에 의해 파괴될 때 닫힙니다.

winreg.**ConnectRegistry**(*computer_name*, *key*)

다른 컴퓨터에 있는 사전 정의된 레지스트리 핸들에 연결하고, 핸들 객체를 반환합니다.

*computer_name*은 `r"\\computername"` 형식의 원격 컴퓨터 이름입니다. `None`이면, 로컬 컴퓨터가 사용됩니다.

*key*는 연결할 사전 정의된 핸들입니다.

반환 값은 열린 키의 핸들입니다. 함수가 실패하면, `OSError` 예외가 발생합니다.

인자 `computer_name`, `key`로 감사 이벤트 `winreg.ConnectRegistry`를 발생시킵니다.

버전 3.3에서 변경: 위를 참조하십시오.

winreg.**CreateKey**(*key*, *sub_key*)

지정된 키를 만들거나 열어, 핸들 객체를 반환합니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_** 상수 중 하나입니다.

*sub_key*는 이 메서드가 열거나 만드는 키의 이름을 지정하는 문자열입니다.

*key*가 사전 정의된 키 중 하나이면, *sub_key*는 `None`일 수 있습니다. 이 경우, 반환된 핸들은 함수에 전달된 것과 같은 키 핸들입니다.

키가 이미 존재하면, 이 함수는 기존 키를 엽니다.

반환 값은 열린 키의 핸들입니다. 함수가 실패하면, `OSError` 예외가 발생합니다.

인자 `key`, `sub_key`, `access`로 감사 이벤트 `winreg.CreateKey`를 발생시킵니다.

인자 `key`로 감사 이벤트 `winreg.OpenKey/result`를 발생시킵니다.

버전 3.3에서 변경: 위를 참조하십시오.

winreg.**CreateKeyEx**(*key*, *sub_key*, *reserved=0*, *access=KEY_WRITE*)

지정된 키를 만들거나 열어, 핸들 객체를 반환합니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_** 상수 중 하나입니다.

*sub_key*는 이 메서드가 열거나 만드는 키의 이름을 지정하는 문자열입니다.

*reserved*는 예약된 정수이며, 0이어야 합니다. 기본값은 0입니다.

*access*는 키에 대한 원하는 보안 액세스를 기술하는 액세스 마스크를 지정하는 정수입니다. 기본값은 `KEY_WRITE`입니다. 허용되는 다른 값은 액세스 권한을 참조하십시오.

*key*가 사전 정의된 키 중 하나이면, *sub_key*는 `None`일 수 있습니다. 이 경우, 반환된 핸들은 함수에 전달된 것과 같은 키 핸들입니다.

키가 이미 존재하면, 이 함수는 기존 키를 엽니다.

반환 값은 열린 키의 핸들입니다. 함수가 실패하면, `OSError` 예외가 발생합니다.

인자 `key`, `sub_key`, `access`로 감사 이벤트 `winreg.CreateKey`를 발생시킵니다.

인자 `key`로 감사 이벤트 `winreg.OpenKey/result`를 발생시킵니다.

Added in version 3.2.

버전 3.3에서 변경: 위를 참조하십시오.

`winreg.`**`DeleteKey`**(*key*, *sub_key*)

지정된 키를 삭제합니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_** 상수 중 하나입니다.

*sub_key*는 *key* 매개 변수로 식별된 키의 서브 키여야 하는 문자열입니다. 이 값은 `None`이 아니어야 하며, 키에 서브 키가 없을 수 있습니다.

이 메서드는 서브 키가 있는 키를 삭제할 수 없습니다.

메서드가 성공하면, 모든 값을 포함하여 전체 키가 제거됩니다. 메서드가 실패하면, `OSError` 예외가 발생합니다.

인자 `key`, `sub_key`, `access`로 감사 이벤트 `winreg.DeleteKey`를 발생시킵니다.

버전 3.3에서 변경: 위를 참조하십시오.

`winreg.`**`DeleteKeyEx`**(*key*, *sub_key*, *access=KEY_WOW64_64KEY*, *reserved=0*)

지정된 키를 삭제합니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_** 상수 중 하나입니다.

*sub_key*는 *key* 매개 변수로 식별된 키의 서브 키여야 하는 문자열입니다. 이 값은 `None`이 아니어야 하며, 키에 서브 키가 없을 수 있습니다.

*reserved*는 예약된 정수이며, 0이어야 합니다. 기본값은 0입니다.

*access*는 키에 대한 원하는 보안 액세스를 기술하는 액세스 마스크를 지정하는 정수입니다. 기본값은 `KEY_WOW64_64KEY`입니다. 32비트 윈도우에서는, WOW64 상수가 무시됩니다. 허용되는 다른 값은 액세스 권한을 참조하십시오.

이 메서드는 서브 키가 있는 키를 삭제할 수 없습니다.

메서드가 성공하면, 모든 값을 포함하여 전체 키가 제거됩니다. 메서드가 실패하면, `OSError` 예외가 발생합니다.

지원되지 않는 윈도우 버전에서는, `NotImplementedError` 가 발생합니다.

인자 `key`, `sub_key`, `access`로 감사 이벤트 `winreg.DeleteKey`를 발생시킵니다.

Added in version 3.2.

버전 3.3에서 변경: 위를 참조하십시오.

`winreg.`**`DeleteValue`**(*key*, *value*)

레지스트리 키에서 명명된 값을 제거합니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_** 상수 중 하나입니다.

*value*는 제거할 값을 식별하는 문자열입니다.

인자 `key`, `value`로 감사 이벤트 `winreg.DeleteValue`를 발생시킵니다.

winreg.**EnumKey**(*key*, *index*)

열린 레지스트리 키의 서브 키를 열거하고, 문자열을 반환합니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_* 상수* 중 하나입니다.

*index*는 꺼낼 키의 인덱스를 식별하는 정수입니다.

이 함수는 호출될 때마다 하나의 서브 키 이름을 꺼냅니다. 일반적으로 더 이상 사용할 수 있는 값이 없음을 나타내는 `OSError` 예외가 발생할 때까지 반복적으로 호출됩니다.

인자 `key`, `index`로 감사 이벤트 `winreg.EnumKey`를 발생시킵니다.

버전 3.3에서 변경: 위를 참조하십시오.

winreg.**EnumValue**(*key*, *index*)

열린 레지스트리 키의 값을 열거하고, 튜플을 반환합니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_* 상수* 중 하나입니다.

*index*는 꺼낼 값의 인덱스를 식별하는 정수입니다.

이 함수는 호출될 때마다 하나의 서브 키 이름을 꺼냅니다. 일반적으로 더는 값이 없음을 표시하는 `OSError` 예외가 발생할 때까지 반복적으로 호출됩니다.

결과는 3개의 항목으로 구성된 튜플입니다:

인덱스	의미
0	값 이름을 식별하는 문자열
1	값 데이터를 담은 객체, 형이 하부 레지스트리 유형에 따라 달라집니다
2	값 데이터의 형을 식별하는 정수 (`SetValueEx()`에 대한 설명서에 있는 표를 참조하십시오)

인자 `key`, `index`로 감사 이벤트 `winreg.EnumValue`를 발생시킵니다.

버전 3.3에서 변경: 위를 참조하십시오.

winreg.**ExpandEnvironmentStrings**(*str*)

*REG_EXPAND_SZ*와 같은 문자열에서 환경 변수 자리 표시자 `%NAME%`을 확장합니다:

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

인자 `str`로 감사 이벤트 `winreg.ExpandEnvironmentStrings`를 발생시킵니다.

winreg.**FlushKey**(*key*)

키의 모든 어트리뷰트를 레지스트리에 씁니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_* 상수* 중 하나입니다.

키를 변경하기 위해 `FlushKey()`를 호출할 필요는 없습니다. 레지스트리 변경은 지연 플러셔를 사용하여 레지스트리에 의해 디스크로 플러시 됩니다. 레지스트리 변경은 시스템 종료 시에도 디스크로 플러시 됩니다. `CloseKey()`와 달리, `FlushKey()` 메서드는 모든 데이터가 레지스트리에 기록될 때만 반환합니다. 응용 프로그램은 레지스트리 변경이 디스크에 있다는 절대적인 확신이 필요할 때만 `FlushKey()`를 호출해야 합니다.

> **참고**
>
> `FlushKey()` 호출이 필요한지 모른다면, 아마도 필요하지 않습니다.

winreg.**LoadKey**(*key*, *sub_key*, *file_name*)

지정된 키 아래에 서브 키를 만들고 지정된 파일에 있는 등록 정보를 그 서브 키에 저장합니다.

*key*는 `ConnectRegistry()`가 반환한 핸들이거나 상수 `HKEY_USERS`나 `HKEY_LOCAL_MACHINE` 중 하나입니다.

*sub_key*는 로드할 서브 키를 식별하는 문자열입니다.

*file_name*은 레지스트리 데이터를 로드할 파일의 이름입니다. 이 파일은 `SaveKey()` 함수로 만들어졌어야 합니다. FAT(file allocation table) 파일 시스템에서, 파일명은 확장자가 없을 수 있습니다.

호출하는 프로세스에 `SE_RESTORE_PRIVILEGE` 권한(privilege)이 없으면 `LoadKey()`에 대한 호출이 실패합니다. 권한(privilege)은 허가(permissions)와 다름에 유의하십시오 – 자세한 내용은 RegLoadKey 설명서를 참조하십시오.

*key*가 `ConnectRegistry()`가 반환한 핸들이면, *file_name*에 지정된 경로는 원격 컴퓨터에 상대적입니다.

인자 `key`, `sub_key`, `file_name`으로 감사 이벤트 `winreg.LoadKey`를 발생시킵니다.

winreg.**OpenKey**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)

winreg.**OpenKeyEx**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)

지정된 키를 열고, 핸들 객체를 반환합니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_** 상수 중 하나입니다.

*sub_key*는 열 서브 키를 식별하는 문자열입니다.

*reserved*는 예약된 정수이며, 0이어야 합니다. 기본값은 0입니다.

*access*는 키에 대한 원하는 보안 액세스를 기술하는 액세스 마스크를 지정하는 정수입니다. 기본값은 `KEY_READ`입니다. 허용되는 다른 값은 액세스 권한을 참조하십시오.

결과는 지정된 키에 대한 새로운 핸들입니다.

함수가 실패하면, `OSError`가 발생합니다.

인자 `key`, `sub_key`, `access`로 감사 이벤트 `winreg.OpenKey`를 발생시킵니다.

인자 `key`로 감사 이벤트 `winreg.OpenKey/result`를 발생시킵니다.

버전 3.2에서 변경: 명명된 인자 사용을 허용합니다.

버전 3.3에서 변경: 위를 참조하십시오.

winreg.**QueryInfoKey**(*key*)

키에 대한 정보를 튜플로 반환합니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_** 상수 중 하나입니다.

결과는 3개의 항목으로 구성된 튜플입니다:

인덱스	의미
0	이 키가 가진 서브 키의 수를 제공하는 정수.
1	이 키가 가진 값의 수를 제공하는 정수.
2	키가 마지막으로 수정된 때(있다면)를 1601년 1월 1일 이후로 지난 100나노초로 제공하는 정수.

인자 `key`로 감사 이벤트 `winreg.QueryInfoKey`를 발생시킵니다.

winreg.**QueryValue**(*key*, *sub_key*)

키의 이름이 없는 값을 문자열로 가져옵니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_** 상수 중 하나입니다.

*sub_key*는 값이 연관된 서브 키의 이름을 담은 문자열입니다. 이 매개 변수가 `None`이거나 비어있으면, 함수는 *key*로 식별된 키에 대해 `SetValue()` 메서드로 설정된 값을 가져옵니다.

레지스트리의 값에는 이름, 형 및 데이터 구성 요소가 있습니다. 이 메서드는 `NULL` 이름을 가진 키의 첫 번째 값에 대한 데이터를 가져옵니다. 그러나 하부 API 호출은 형을 반환하지 않아서, 가능하다면 항상 `QueryValueEx()`를 사용하십시오.

인자 `key`, `sub_key`, `value_name`으로 감사 이벤트 `winreg.QueryValue`를 발생시킵니다.

`winreg.`**`QueryValueEx`**(*key*, *value_name*)

열린 레지스트리 키와 연관된 지정된 값 이름의 형과 데이터를 가져옵니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_** 상수 중 하나입니다.

*value_name*은 조회할 값을 나타내는 문자열입니다.

결과는 2개의 항목으로 구성된 튜플입니다:

인덱스	의미
0	레지스트리 항목의 값.
1	이 값에 대한 레지스트리 유형을 제공하는 정수 (`SetValueEx()`의 설명서에 있는 표를 참조하십시오)

인자 `key`, `sub_key`, `value_name`으로 감사 이벤트 `winreg.QueryValue`를 발생시킵니다.

`winreg.`**`SaveKey`**(*key*, *file_name*)

지정된 키와 그것의 모든 서브 키를 지정된 파일에 저장합니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_** 상수 중 하나입니다.

*file_name*은 레지스트리 데이터를 저장할 파일 이름입니다. 이 파일은 이미 존재할 수 없습니다. 이 파일명에 확장자가 포함되어 있으면, `LoadKey()` 메서드로 FAT(file allocation table) 파일 시스템에서 사용할 수 없습니다.

*key*가 원격 컴퓨터의 키를 나타내면, *file_name*이 기술하는 경로는 원격 컴퓨터에 상대적입니다. 이 메서드의 호출자는 **SeBackupPrivilege** 보안 권한(privilege)을 가지고 있어야 합니다. 권한(privilege)은 허가(permissions)와 다름에 유의하십시오 – 자세한 내용은 사용자 권한과 허가 간의 충돌 설명서를 참조하십시오.

이 함수는 *security_attributes*로 `NULL`을 API로 전달합니다.

인자 `key`, `file_name`으로 감사 이벤트 `winreg.SaveKey`를 발생시킵니다.

`winreg.`**`SetValue`**(*key*, *sub_key*, *type*, *value*)

값을 지정된 키와 연관시킵니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_** 상수 중 하나입니다.

*sub_key*는 값이 연관된 서브 키의 이름을 지정하는 문자열입니다.

*type*은 데이터의 형을 지정하는 정수입니다. 현재 이것은 `REG_SZ` 여야 하는데, 문자열만 지원된다는 뜻입니다. 다른 데이터형을 지원하려면 `SetValueEx()` 함수를 사용하십시오.

*value*는 새 값을 지정하는 문자열입니다.

sub_key 매개 변수로 지정된 키가 존재하지 않으면, SetValue 함수가 이를 만듭니다.

값 길이는 사용 가능한 메모리에 따라 제한됩니다. 긴 값(2048바이트보다 긴)은 구성 레지스트리에 저장된 파일명을 가진 파일로 저장해야 합니다. 이렇게 하면 레지스트리가 효율적으로 수행하는 데 도움을 줍니다.

key 매개 변수로 식별된 키는 `KEY_SET_VALUE` 액세스로 열렸어야 합니다.

인자 `key`, `sub_key`, `type`, `value`로 감사 이벤트 `winreg.SetValue`를 발생시킵니다.

winreg.**SetValueEx**(*key, value_name, reserved, type, value*)

열린 레지스트리 키의 값 필드에 데이터를 저장합니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_* 상수* 중 하나입니다.

*value_name*은 값이 연관된 서브 키의 이름을 지정하는 문자열입니다.

*reserved*는 무엇이든 가능합니다 – 0이 항상 API로 전달됩니다.

*type*은 데이터의 형을 지정하는 정수입니다. 사용 가능한 형은 값 형을 참조하십시오.

*value*는 새 값을 지정하는 문자열입니다.

이 메서드는 지정된 키에 대한 추가 값과 형 정보를 설정할 수도 있습니다. key 매개 변수로 식별된 키는 `KEY_SET_VALUE` 액세스로 열렸어야 합니다.

키를 열려면, `CreateKey()` 나 `OpenKey()` 메서드를 사용하십시오.

값 길이는 사용 가능한 메모리에 따라 제한됩니다. 긴 값(2048바이트보다 긴)은 구성 레지스트리에 저장된 파일명을 가진 파일로 저장해야 합니다. 이렇게 하면 레지스트리가 효율적으로 수행하는 데 도움을 줍니다.

인자 `key, sub_key, type, value`로 감사 이벤트 `winreg.SetValue`를 발생시킵니다.

winreg.**DisableReflectionKey**(*key*)

64비트 운영 체제에서 실행 중인 32비트 프로세스에 대한 레지스트리 리플렉션(reflection)을 비활성화합니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_* 상수* 중 하나입니다.

32비트 운영 체제에서 실행하면 일반적으로 `NotImplementedError` 가 발생합니다.

키가 리플렉션 목록에 없으면, 함수는 성공하지만 아무런 효과가 없습니다. 키에 대한 리플렉션을 비활성화해도 서브 키의 리플렉션에는 영향을 미치지 않습니다.

인자 `key`로 감사 이벤트 `winreg.DisableReflectionKey`를 발생시킵니다.

winreg.**EnableReflectionKey**(*key*)

지정된 비활성화 된 키에 대한 레지스트리 리플렉션(reflection)을 복원합니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_* 상수* 중 하나입니다.

32비트 운영 체제에서 실행하면 일반적으로 `NotImplementedError` 가 발생합니다.

키에 대한 리플렉션을 복원해도 서브 키의 리플렉션에 영향을 미치지 않습니다.

인자 `key`로 감사 이벤트 `winreg.EnableReflectionKey`를 발생시킵니다.

winreg.**QueryReflectionKey**(*key*)

지정된 키의 리플렉션(reflection) 상태를 판단합니다.

*key*는 이미 열린 키이거나, 사전 정의된 *HKEY_* 상수* 중 하나입니다.

리플렉션이 비활성화되었으면 `True`를 반환합니다.

32비트 운영 체제에서 실행하면 일반적으로 `NotImplementedError` 가 발생합니다.

인자 `key`로 감사 이벤트 `winreg.QueryReflectionKey`를 발생시킵니다.

33.2.2 상수

많은 `winreg` 함수에 사용하기 위해 다음 상수가 정의되어 있습니다.

HKEY_* 상수

winreg.**HKEY_CLASSES_ROOT**

이 키에 종속된 레지스트리 항목은 문서의 형(또는 클래스)과 해당 형과 연관된 속성을 정의합니다. 셸과 COM 응용 프로그램은 이 키에 저장된 정보를 사용합니다.

winreg.**HKEY_CURRENT_USER**

이 키에 종속된 레지스트리 항목은 현재 사용자의 환경 설정(preferences)을 정의합니다. 이러한 환경 설정에는 환경 변수 설정, 프로그램 그룹, 색상, 프린터, 네트워크 연결 및 응용 프로그램 환경 설정에 대한 데이터가 포함됩니다.

winreg.**HKEY_LOCAL_MACHINE**

이 키에 종속된 레지스트리 항목은 버스 유형, 시스템 메모리 및 설치된 하드웨어와 소프트웨어에 대한 데이터를 포함하는 컴퓨터의 물리적 상태를 정의합니다.

winreg.**HKEY_USERS**

이 키에 종속된 레지스트리 항목은 로컬 컴퓨터의 새 사용자를 위한 기본 사용자 구성과 현재 사용자의 사용자 구성을 정의합니다.

winreg.**HKEY_PERFORMANCE_DATA**

이 키에 종속된 레지스트리 항목을 사용하면 성능 데이터에 액세스 할 수 있습니다. 데이터는 실제로 레지스트리에 저장되지 않습니다; 레지스트리 함수는 시스템이 소스에서 데이터를 수집하도록 합니다.

winreg.**HKEY_CURRENT_CONFIG**

로컬 컴퓨터 시스템의 현재 하드웨어 프로필에 대한 정보가 들어 있습니다.

winreg.**HKEY_DYN_DATA**

이 키는 98 이후의 윈도우 버전에서는 사용되지 않습니다.

액세스 권한

자세한 내용은 레지스트리 키 보안과 액세스를 참조하십시오.

winreg.**KEY_ALL_ACCESS**

STANDARD_RIGHTS_REQUIRED, *KEY_QUERY_VALUE*, *KEY_SET_VALUE*, *KEY_CREATE_SUB_KEY*, *KEY_ENUMERATE_SUB_KEYS*, *KEY_NOTIFY* 및 *KEY_CREATE_LINK* 액세스 권한을 결합합니다.

winreg.**KEY_WRITE**

STANDARD_RIGHTS_WRITE, *KEY_SET_VALUE* 및 *KEY_CREATE_SUB_KEY* 액세스 권한을 결합합니다.

winreg.**KEY_READ**

STANDARD_RIGHTS_READ, *KEY_QUERY_VALUE*, *KEY_ENUMERATE_SUB_KEYS* 및 *KEY_NOTIFY* 값을 결합합니다.

winreg.**KEY_EXECUTE**

*KEY_READ*와 동등합니다.

winreg.**KEY_QUERY_VALUE**

레지스트리 키의 값을 조회하는 데 필요합니다.

winreg.**KEY_SET_VALUE**

레지스트리 값을 생성, 삭제 또는 설정하는 데 필요합니다.

winreg.**KEY_CREATE_SUB_KEY**

레지스트리 키의 서브 키를 만드는 데 필요합니다.

winreg.**KEY_ENUMERATE_SUB_KEYS**

레지스트리 키의 서브 키를 열거하는 데 필요합니다.

winreg.**KEY_NOTIFY**

레지스트리 키나 레지스트리 키의 서브 키에 대한 변경 알림을 요청하는 데 필요합니다.

winreg.**KEY_CREATE_LINK**

시스템 사용을 위해 예약되어 있습니다.

64비트 특정

자세한 내용은 대체 레지스트리 뷰에 액세스하기를 참조하십시오.

winreg.**KEY_WOW64_64KEY**

64비트 윈도우의 응용 프로그램이 64비트 레지스트리 뷰에서 작동해야 함을 나타냅니다. 32비트 윈도우에서, 이 상수는 무시됩니다.

winreg.**KEY_WOW64_32KEY**

64비트 윈도우의 응용 프로그램이 32비트 레지스트리 뷰에서 작동해야 함을 나타냅니다. 32비트 윈도우에서, 이 상수는 무시됩니다.

값 형

자세한 내용은 레지스트리 값 형을 참조하십시오.

winreg.**REG_BINARY**

모든 형태의 바이너리 데이터.

winreg.**REG_DWORD**

32비트 숫자.

winreg.**REG_DWORD_LITTLE_ENDIAN**

리틀 엔디안 형식의 32비트 숫자. *REG_DWORD*와 동등합니다.

winreg.**REG_DWORD_BIG_ENDIAN**

빅 엔디안 형식의 32비트 숫자.

winreg.**REG_EXPAND_SZ**

환경 변수(%PATH%)에 대한 참조를 포함하는 널 종료 문자열.

winreg.**REG_LINK**

유니코드 심볼릭 링크.

winreg.**REG_MULTI_SZ**

두 개의 널 문자로 끝나는 널 종료 문자열의 시퀀스. (파이썬은 이 종료를 자동으로 처리합니다.)

winreg.**REG_NONE**

정의된 값 형이 없습니다.

winreg.**REG_QWORD**

64비트 숫자.

Added in version 3.6.

winreg.**REG_QWORD_LITTLE_ENDIAN**

리틀 엔디안 형식의 64비트 숫자. *REG_QWORD*와 동등합니다.

Added in version 3.6.

winreg.**REG_RESOURCE_LIST**

장치 드라이버 리소스 목록.

winreg.**REG_FULL_RESOURCE_DESCRIPTOR**

하드웨어 설정.

winreg.**REG_RESOURCE_REQUIREMENTS_LIST**

하드웨어 리소스 목록.

winreg.**REG_SZ**

널 종료 문자열.

33.2.3 레지스트리 핸들 객체

이 객체는 윈도우 HKEY 객체를 감싸서, 객체가 파괴될 때 자동으로 닫습니다. 정리를 보장하기 위해, 객체의 *Close()* 메서드나 *CloseKey()* 함수를 호출할 수 있습니다.

이 모듈의 모든 레지스트리 함수는 이러한 객체 중 하나를 반환합니다.

핸들 객체를 받아들이는 이 모듈의 모든 레지스트리 함수는 정수도 받아들이지만, 핸들 객체의 사용을 권장합니다.

핸들 객체는 `__bool__()` 에 대한 의미를 제공합니다 - 그래서

```
if handle:
    print("Yes")
```

는 핸들이 현재 유효하면 (닫혔거나 분리되지(detached) 않았으면) `Yes`를 인쇄합니다.

객체는 또한 비교 개념을 지원하므로, 핸들 객체가 모두 같은 하부 윈도우 핸들값을 참조하면 참으로 비교됩니다.

핸들 객체는 정수로 변환될 수 있으며 (예를 들어, 내장 *int()* 함수 사용해서), 이 경우 하부 윈도우 핸들값이 반환됩니다. *Detach()* 메서드를 사용하여 정수 핸들을 반환하고 핸들 객체에서 윈도우 핸들을 분리할 수도 있습니다.

`PyHKEY.`**`Close`**`()`

하부 윈도우 핸들을 닫습니다.

핸들이 이미 닫혀 있으면, 에러가 발생하지 않습니다.

`PyHKEY.`**`Detach`**`()`

핸들 객체에서 윈도우 핸들을 분리합니다.

결과는 핸들이 분리되기 전의 핸들 값을 담고 있는 정수입니다. 핸들이 이미 분리되었거나 닫혔으면 0이 반환됩니다.

이 함수를 호출한 후에는, 핸들이 효과적으로 무효가 되지만, 핸들이 닫히지는 않습니다. 하부 Win32 핸들이 핸들 객체의 수명을 넘어 존재해야 할 때 이 함수를 호출합니다.

인자 `key`로 감사 이벤트 `winreg.PyHKEY.Detach`를 발생시킵니다.

`PyHKEY.`**`__enter__`**`()`

`PyHKEY.`**`__exit__`**`(`**exc_info*`)`

HKEY 객체는 `__enter__()` 와 `__exit__()` 를 구현하므로 `with` 문의 컨텍스트 프로토콜을 지원합니다:

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...  # key로 작업합니다
```

는 제어가 `with` 블록을 벗어날 때 *key*를 자동으로 닫습니다.

33.3 winsound — 윈도우용 소리 재생 인터페이스

---

winsound 모듈은 윈도우 플랫폼에서 제공하는 기본 소리 재생 장치에 대한 액세스를 제공합니다. 함수와 여러 상수를 포함합니다.

`winsound.`**`Beep`**`(`*frequency*, *duration*`)`

PC 스피커로 신호음을 울립니다. *frequency* 매개 변수는 소리의 주파수를 헤르츠 단위로 지정하며 37에서 32,767 범위에 있어야 합니다. *duration* 매개 변수는 소리의 지속 시간을 밀리 초로 지정합니다. 시스템이 스피커에서 신호음을 울리지 못하면, *RuntimeError*가 발생합니다.

winsound.**PlaySound**(*sound*, *flags*)

플랫폼 API에서 하부 `PlaySound()` 함수를 호출합니다. *sound* 매개 변수는 파일명, 시스템 소리 별칭, 바이트열류 객체의 오디오 데이터 또는 `None` 일 수 있습니다. 해석은 *flags*의 값에 따라 달라지는데, 아래에 설명된 상수의 비트별 OR 결합이 될 수 있습니다. *sound* 매개 변수가 `None`이면, 현재 재생 중인 파형 소리가 중지됩니다. 시스템이 에러를 표시하면 `RuntimeError`가 발생합니다.

winsound.**MessageBeep**(*type=MB_OK*)

플랫폼 API에서 하부 `MessageBeep()` 함수를 호출합니다. 레지스트리에 지정된 소리를 재생합니다. *type* 인자는 재생할 소리를 지정합니다; 가능한 값은 아래에 설명된 `-1`, `MB_ICONASTERISK`, `MB_ICONEXCLAMATION`, `MB_ICONHAND`, `MB_ICONQUESTION` 및 `MB_OK`입니다. `-1` 값은 "간단한 신호음"을 생성합니다; 이것은 소리를 재생할 수 없을 때 최종 대체가 됩니다. 시스템이 에러를 표시하면 `RuntimeError`가 발생합니다.

winsound.**SND_FILENAME**

sound 매개 변수는 WAV 파일의 이름입니다. `SND_ALIAS`와 함께 사용하지 마십시오.

winsound.**SND_ALIAS**

sound 매개 변수는 레지스트리의 소리 연결 이름입니다. 레지스트리에 그러한 이름이 없으면, `SND_NODEFAULT`도 함께 지정하지 않는 한 시스템 기본 소리를 재생합니다. 기본 소리가 등록되어 있지 않으면, `RuntimeError`를 일으킵니다. `SND_FILENAME`과 함께 사용하지 마십시오.

모든 Win32 시스템은 적어도 다음을 지원합니다; 대부분 시스템은 더 많은 것을 지원합니다:

`PlaySound()` 이름	해당 제어판 소리 이름
`'SystemAsterisk'`	Asterisk
`'SystemExclamation'`	Exclamation
`'SystemExit'`	Exit Windows
`'SystemHand'`	Critical Stop
`'SystemQuestion'`	Question

예를 들면:

```
import winsound
# 윈도우 종료 소리를 재생합니다.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# 아마도 (등록되어 있다면) 윈도우 기본 소리를 재생합니다 (아마도 "*" 는 등록 된 소리
# 이름이 아닐 것이기 때문입니다).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.**SND_LOOP**

소리를 반복해서 재생합니다. 블로킹을 피하고자 `SND_ASYNC` 플래그도 사용해야 합니다. `SND_MEMORY`와 함께 사용할 수 없습니다.

winsound.**SND_MEMORY**

`PlaySound()`에 대한 *sound* 매개 변수는 바이트열류 객체의 WAV 파일의 메모리 이미지입니다.

> **참고**
>
> 이 모듈은 메모리 이미지를 비동기적으로 재생하는 것을 지원하지 않으므로, 이 플래그와 `SND_ASYNC`의 조합은 `RuntimeError`를 발생시킵니다.

winsound.**SND_PURGE**

지정된 소리의 모든 인스턴스 재생을 중지합니다.

> **참고**
>
> 이 플래그는 최신 윈도우 플랫폼에서 지원되지 않습니다.

winsound.**SND_ASYNC**

소리를 비동기적으로 재생할 수 있도록, 즉시 반환합니다.

winsound.**SND_NODEFAULT**

지정된 소리를 찾을 수 없을 때, 시스템 기본 소리를 재생하지 않습니다.

winsound.**SND_NOSTOP**

현재 재생 중인 소리를 중단하지 않습니다.

winsound.**SND_NOWAIT**

사운드 드라이버가 바쁘면 즉시 반환합니다.

> **참고**
>
> 이 플래그는 최신 윈도우 플랫폼에서 지원되지 않습니다.

winsound.**SND_APPLICATION**

The *sound* parameter is an application-specific alias in the registry. This flag can be combined with the *SND_ALIAS* flag to specify an application-defined sound alias.

winsound.**SND_SENTRY**

Triggers a SoundSentry event when the sound is played.

Added in version 3.14.

winsound.**SND_SYNC**

The sound is played synchronously. This is the default behavior.

Added in version 3.14.

winsound.**SND_SYSTEM**

Assign the sound to the audio session for system notification sounds.

Added in version 3.14.

winsound.**MB_ICONASTERISK**

`SystemDefault` 소리를 재생합니다.

winsound.**MB_ICONEXCLAMATION**

`SystemExclamation` 소리를 재생합니다.

winsound.**MB_ICONHAND**

`SystemHand` 소리를 재생합니다.

winsound.**MB_ICONQUESTION**

`SystemQuestion` 소리를 재생합니다.

winsound.**MB_OK**

`SystemDefault` 소리를 재생합니다.

winsound.**MB_ICONERROR**

`SystemHand` 소리를 재생합니다.

Added in version 3.14.

winsound.**MB_ICONINFORMATION**

`SystemDefault` 소리를 재생합니다.

Added in version 3.14.

winsound.**MB_ICONSTOP**

`SystemHand` 소리를 재생합니다.

Added in version 3.14.

winsound.**MB_ICONWARNING**

`SystemExclamation` 소리를 재생합니다.

Added in version 3.14.

CHAPTER 34

Unix-specific services

이 장에서 설명하는 모듈은 유닉스 운영 체제(혹은 때에 따라 일부 혹은 많은 변종)에 고유한 기능에 대한 인터페이스를 제공합니다. 다음은 개요입니다:

34.1 shlex — 간단한 어휘 분석

소스 코드: Lib/shlex.py

---

shlex 클래스를 사용하면 유닉스 셸과 유사한 간단한 구문에 대한 어휘 분석기를 쉽게 작성할 수 있습니다. 이것은 미니 언어를 작성하거나 (예를 들어 파이썬 응용 프로그램을 위한 실행 제어 파일에서), 인용된 문자열을 구문 분석할 때 유용합니다.

shlex 모듈은 다음 함수를 정의합니다:

shlex.**split**(*s*, *comments=False*, *posix=True*)

셸과 비슷한 문법을 사용하여 문자열 *s*를 분할합니다. *comments*가 *False*(기본값)이면, 지정된 문자열의 주석 구문 분석이 비활성화됩니다 (*shlex* 인스턴스의 *commenters* 어트리뷰트를 빈 문자열로 설정합니다). 이 함수는 기본적으로 POSIX 모드로 작동하지만, *posix* 인자가 거짓이면 비 POSIX 모드를 사용합니다.

버전 3.12에서 변경: *s* 인자에 None을 전달하면 이제 *sys.stdin*을 읽는 대신 예외를 일으킵니다.

shlex.**join**(*split_command*)

리스트 *split_command*의 토큰을 이어붙이고 문자열을 반환합니다. 이 함수는 *split()*의 역함수입니다.

```
>>> from shlex import join
>>> print(join(['echo', '-n', 'Multiple words']))
echo -n 'Multiple words'
```

반환된 값은 주입 취약점(injection vulnerabilities)으로부터 보호하기 위해 셸 이스케이프 처리됩니다 (*quote()*를 참조하십시오).

Added in version 3.8.

shlex.**quote**(*s*)

셸 이스케이프 된 문자열 *s*를 반환합니다. 반환된 값은 (리스트를 사용할 수 없는 경우) 셸 명령 줄에서 하나의 토큰으로 안전하게 사용할 수 있는 문자열입니다.

> **경고**
>
> The `shlex` module is **only designed for Unix shells**.
>
> The `quote()` function is not guaranteed to be correct on non-POSIX compliant shells or shells from other operating systems such as Windows. Executing commands quoted by this module on such shells can open up the possibility of a command injection vulnerability.
>
> Consider using functions that pass command arguments with lists such as `subprocess.run()` with `shell=False`.

이 관용구는 안전하지 않습니다:

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

`quote()`를 사용하면 보안 허점을 메꿀 수 있습니다:

```
>>> from shlex import quote
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"'
```

인용(quoting)은 유닉스 셸과 `split()`과 호환됩니다:

```
>>> from shlex import split
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

Added in version 3.3.

`shlex` 모듈은 다음 클래스를 정의합니다:

class shlex.**shlex**(*instream=None, infile=None, posix=False, punctuation_chars=False*)

`shlex` 인스턴스나 서브 클래스 인스턴스는 어휘 분석기 객체입니다. 존재할 때 초기화 인자는 문자를 어디에서 읽을지를 지정합니다. `read()`와 `readline()` 메서드가 있는 파일/스트림류 객체이거나 문자열이어야 합니다. 인자가 없으면 `sys.stdin`에서 입력을 받습니다. 두 번째 선택적 인자는 파일명 문자열이며, `infile` 어트리뷰트의 초깃값을 설정합니다. *instream* 인자가 생략되거나 `sys.stdin`과 같으면, 이 두 번째 인자의 기본값은 "stdin" 입니다. *posix* 인자는 작동 모드를 정의합니다: *posix*가 참이 아닐 때 (기본값), `shlex` 인스턴스는 호환 모드에서 작동합니다. POSIX 모드에서 작동할 때, `shlex`는 가능한 한 POSIX 셸 구문 분석 규칙에 가깝도록 시도합니다. *punctuation_chars* 인자는 동작을 실제 셸이 구문 분석하는 방식에 더 가깝게 만드는 방법을 제공합니다. 이것은 여러 종류의 값을 취할 수 있습니다: 기본값 `False`는 파이썬 3.5와 이전 버전에서의 동작을 유지합니다. `True`로 설정되면, 문자 `();<>|&`의 구문 분석이 변경됩니다: 이러한 문자(구두(punctuation) 문자로 간주합니다)의 모든 연속은 단일 토큰으로 반환됩니다. 비어 있지 않은 문자열로 설정하면, 해당 문자는 구두(punctuation) 문자로 사용됩니다. *punctuation_chars*에 나타나는 `wordchars` 어트리뷰트의 문자는

wordchars에서 제거됩니다. 자세한 정보는 셸과의 호환성 향상을 참조하십시오. *punctuation_chars* 는 shlex 인스턴스 생성 시에만 설정할 수 있으며 나중에 수정할 수 없습니다.

버전 3.6에서 변경: *punctuation_chars* 매개 변수가 추가되었습니다.

더 보기

모듈 configparser
윈도우 .ini 파일과 유사한 구성 파일 구문 분석기.

34.1.1 shlex 객체

shlex 인스턴스에는 다음과 같은 메서드가 있습니다:

shlex.**get_token**()

토큰을 반환합니다. push_token() 을 사용하여 토큰이 스택(stack) 되었으면, 스택에서 토큰을 팝(pop) 합니다. 그렇지 않으면, 입력 스트림에서 하나를 읽습니다. 읽기가 즉시 파일 끝을 만나면, eof가 반환됩니다 (POSIX 모드가 아니면 빈 문자열 (''), POSIX 모드이면 None).

shlex.**push_token**(*str*)

인자를 토큰 스택으로 푸시(push) 합니다.

shlex.**read_token**()

원시 토큰을 읽습니다. 푸시백 스택을 무시하고, 소스 요청(source requests) 을 해석하지 않습니다. (이것은 일반적으로 유용한 진입점이 아니며, 단지 완전성을 위해 여기에서 설명합니다.)

shlex.**sourcehook**(*filename*)

shlex가 소스 요청(아래 source를 참조하십시오)을 감지할 때 이 메서드에는 다음 토큰이 인자로 제공되고 파일명과 열린 파일류 객체로 구성된 튜플을 반환해야 합니다.

일반적으로, 이 메서드는 먼저 인자에서 인용(quotes)을 제거합니다. 결과가 절대 경로명이거나 유효한 이전 소스 요청이 없거나 이전 소스가 스트림(가령 sys.stdin)이면 결과는 그대로 유지됩니다. 그렇지 않으면, 결과가 상대 경로명이면 소스 포함 스택에서 파일 바로 앞에 있는 파일의 이름의 디렉터리 부분을 앞에 붙입니다 (이 동작은 C 전처리기가 #include "file.h"를 처리하는 방식과 유사합니다).

조작 결과는 파일명으로 취급되고, 튜플의 첫 번째 구성 요소로 반환되며, 이것에 open() 이 호출되어 두 번째 구성 요소를 산출합니다. (참고: 이것은 인스턴스 초기화의 인자 순서와 반대입니다!)

이 훅은 디렉터리 검색 경로, 파일 확장자 추가 및 기타 네임 스페이스 해킹을 구현하는 데 사용할 수 있도록 노출됩니다. 해당 '닫기' 훅은 없지만, shlex 인스턴스는 소스 입력 스트림이 EOF를 반환할 때 그것의 close() 메서드를 호출합니다.

소스 스태킹을 더 명시적으로 제어하려면, push_source()와 pop_source() 메서드를 사용하십시오.

shlex.**push_source**(*newstream*, *newfile=None*)

입력 소스 스트림을 입력 스택으로 푸시합니다. filename 인자가 지정되면 나중에 에러 메시지에 사용할 수 있습니다. 이것은 sourcehook() 메서드에 의해 내부적으로 사용되는 것과 같은 메서드입니다.

shlex.**pop_source**()

마지막으로 푸시 된 입력 소스를 입력 스택에서 팝 합니다. 이는 어휘 분석기가 스택 된 입력 스트림에서 EOF에 도달할 때 내부적으로 사용되는 것과 같은 메서드입니다.

shlex.**error_leader**(*infile=None*, *lineno=None*)

이 메서드는 유닉스 C 컴파일러 에러 레이블 형식으로 에러 메시지 리더를 생성합니다; 형식은 '"%s", line %d: '이며, 여기서 %s는 현재 소스 파일의 이름으로 치환되고 %d는 현재 입력 줄 번호로 치환됩니다 (선택적 인자를 사용하여 이를 재정의할 수 있습니다).

이 편리 메서드는 shlex 사용자가 Emacs와 기타 유닉스 도구가 이해할 수 있는 표준의 구문 분석 가능한 형식으로 에러 메시지를 생성하도록 권장하기 위해 제공됩니다.

shlex 서브 클래스의 인스턴스에는 어휘 분석을 제어하거나 디버깅에 사용할 수 있는 일부 공용 인스턴스 변수가 있습니다:

shlex.**commenters**

주석을 시작하는 것으로 인식되는 문자열. 주석 시작부터 줄 끝까지의 모든 문자는 무시됩니다. 기본적으로 `'#'` 만 포함합니다.

shlex.**wordchars**

다중 문자 토큰에 누적될 문자들의 문자열. 기본적으로, 모든 ASCII 영숫자와 밑줄이 포함됩니다. POSIX 모드에서는, 라틴-1 집합의 악센트 부호 문자도 포함됩니다. *punctuation_chars*가 비어 있지 않으면, 파일명 명세와 명령 줄 매개 변수에 나타날 수 있는 문자 `~-./*?=`도 이 어트리뷰트에 포함되며, `punctuation_chars`에 있는 문자는 `wordchars`에 있으면 제거됩니다. *whitespace_split*이 `True`로 설정되면, 이것은 효과가 없습니다.

shlex.**whitespace**

공백으로 간주하여 건너뛸 문자들. 공백은 토큰의 경계를 만듭니다. 기본적으로, 스페이스, 탭, 줄바꿈 및 캐리지 리턴이 포함됩니다.

shlex.**escape**

이스케이프로 간주하는 문자들. POSIX 모드에서만 사용되며, 기본적으로 `'\'` 만 포함합니다.

shlex.**quotes**

문자열 인용으로 간주하는 문자들. 같은 인용을 다시 만날 때까지 토큰이 누적됩니다 (따라서, 다른 인용 유형은 셸에서와같이 서로를 보호합니다). 기본적으로, ASCII 작은따옴표와 큰따옴표가 포함됩니다.

shlex.**escapedquotes**

*escape*에 정의된 이스케이프 문자를 해석하는 *quotes*의 문자들. 이것은 POSIX 모드에서만 사용되며, 기본적으로 `'"'` 만 포함합니다.

shlex.**whitespace_split**

`True`이면, 토큰은 공백으로만 분할됩니다. 예를 들어, *shlex*로 명령 줄을 구문 분석하고 셸 인자와 유사한 방식으로 토큰을 가져오는 데 유용합니다. *punctuation_chars*와 함께 사용하면, 토큰은 그 문자들 외에도 공백으로 분할됩니다.

버전 3.8에서 변경: *punctuation_chars* 어트리뷰트가 *whitespace_split* 어트리뷰트와 호환되었습니다.

shlex.**infile**

클래스 인스턴스화 시점에 처음 설정되거나 이후 소스 요청으로 스택 된 현재 입력 파일의 이름. 에러 메시지를 구성할 때 이를 조사하는 것이 유용할 수 있습니다.

shlex.**instream**

이 *shlex* 인스턴스가 문자를 읽고 있는 입력 스트림.

shlex.**source**

이 어트리뷰트는 기본적으로 `None`입니다. 문자열을 대입하면, 해당 문자열은 여러 셸의 `source` 키워드와 유사한 어휘 수준 포함 요청으로 인식됩니다. 즉, 바로 다음 토큰이 파일명으로 열리고 그 스트림에서 입력을 EOF까지 취합니다, 이 시점에서 해당 스트림의 *close()* 메서드가 호출되고 입력 소스는 다시 원래 입력 스트림이 됩니다. 소스 요청은 임의의 수준 깊이로 스택 될 수 있습니다.

shlex.**debug**

이 어트리뷰트가 숫자이고 `1` 이상이면, *shlex* 인스턴스는 자신의 동작에 대한 자세한 진행 출력을 인쇄합니다. 이것을 사용해야 하면, 세부 사항을 배우기 위해 모듈 소스 코드를 읽을 수 있습니다.

shlex.**lineno**

소스 줄 번호 (지금까지 본 줄 넘김 개수에 1을 더한 것).

shlex.**token**

토큰 버퍼. 예외를 잡을 때 이를 조사하는 것이 유용 할 수 있습니다.

shlex.**eof**

파일 끝을 판단하는 데 사용되는 토큰. POSIX 모드가 아닐 때 빈 문자열 (''), POSIX 모드일 때 None 으로 설정됩니다.

shlex.**punctuation_chars**

읽기 전용 프로퍼티. 구두 부호로 간주할 문자들. 구두 부호 문자들은 단일 토큰으로 반환됩니다. 그러나, 아무런 의미 유효성 검사도 수행되지 않음에 유의하십시오: 예를 들어 '»>' 는 셸에서 인식되지 않더라도 토큰으로 반환될 수 있습니다.

Added in version 3.6.

34.1.2 구문 분석 규칙

비 POSIX 모드에서 작동할 때, *shlex*는 다음 규칙을 따르려고 합니다.

- 인용 문자는 단어 내에서 인식되지 않습니다 (Do"Not"Separate는 단일 단어 Do"Not"Separate로 구문 분석됩니다);
- 이스케이프 문자는 인식되지 않습니다;
- 인용으로 묶인 문자들은 인용 안에 있는 모든 문자의 리터럴 값을 유지합니다;
- 인용을 닫는 것은 단어를 분리합니다 ("Do"Separate는 "Do"와 Separate로 구문 분석됩니다);
- *whitespace_split*가 False이면, 단어 문자, 공백 또는 인용으로 선언되지 않은 모든 문자는 단일 문자 토큰으로 반환됩니다. True이면, *shlex*는 공백으로 만 단어를 분리합니다;
- EOF는 빈 문자열('')로 알립니다;
- 인용된 경우에도 빈 문자열을 구문 분석할 수 없습니다.

POSIX 모드에서 작동할 때, *shlex*는 다음 구문 분석 규칙을 따르려고 합니다.

- 인용은 제거되고, 단어를 분리하지 않습니다 ("Do"Not"Separate"는 단일 단어 DoNotSeparate로 구문 분석됩니다);
- 인용되지 않은 이스케이프 문자(예를 들어 '\')는 다음에 오는 문자의 리터럴 값을 유지합니다;
- *escapedquotes*의 일부가 아닌 인용으로 묶인 문자(예를 들어 "'")는 인용 안에 있는 모든 문자의 리터럴 값을 유지합니다;
- *escapedquotes*의 일부인 인용으로 묶인 문자(예를 들어 '"')는 *escape*에서 언급된 문자를 제외하고 인용 안에 있는 모든 문자의 리터럴 값을 유지합니다. 이스케이프 문자는 사용 중인 인용이나 이스케이프 문자 자체가 뒤에 오는 경우에만 특별한 의미를 유지합니다. 그렇지 않으면 이스케이프 문자는 일반 문자로 간주합니다.
- EOF는 *None* 값으로 알립니다;
- 인용된 빈 문자열('')이 허용됩니다.

34.1.3 셸과의 호환성 향상

Added in version 3.6.

shlex 클래스는 bash, dash 및 sh와 같은 일반적인 유닉스 셸에서 수행하는 구문 분석과 호환됩니다. 이 호환성을 이용하려면, 생성자에서 punctuation_chars 인자를 지정하십시오. 기본값은 False이며, 3.6 이전 동작을 유지합니다. 그러나, True로 설정되면, 문자 ();<>|&의 구문 분석이 변경됩니다: 이러한 문자의 연속은 단일 토큰으로 반환됩니다. 이것은 셸에 대한 전체 파서로는 부족하지만 (여러 종류의 셸이 있음을 고려할 때, 표준 라이브러리의 범위를 벗어납니다), 이것이 없을 때보다 명령 줄 처리를 더 쉽게 수행할 수 있도록 합니다. 예시하기 위해, 다음 코드 조각에서 차이점을 볼 수 있습니다:

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> s = shlex.shlex(text, posix=True)
>>> s.whitespace_split = True
>>> list(s)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
['a', '&&', 'b;', 'c', '&&', 'd', '||', 'e;', 'f', '>abc;', '(def', 'ghi)']
>>> s = shlex.shlex(text, posix=True, punctuation_chars=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', 'abc', ';',
'(', 'def', 'ghi', ')']
```

물론, 셸에 유효하지 않은 토큰이 반환되며, 반환된 토큰에 대해 여러분 자신의 에러 검사를 구현해야 합니다.

punctuation_chars 매개 변수의 값으로 `True`를 전달하는 대신, 특정 문자가 포함된 문자열을 전달하면 구두 부호를 구성하는 문자를 판별하는 데 사용됩니다. 예를 들면:

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

참고

`punctuation_chars`가 지정되면, *wordchars* 어트리뷰트는 문자 `~-./*?=`로 보강됩니다. 이러한 문자는 파일 이름(와일드카드를 포함해서)과 명령 줄 인자(예를 들어 `--color=auto`)에 나타날 수 있기 때문입니다. 그래서:

```
>>> import shlex
>>> s = shlex.shlex('~/a && b-c --color=auto || d *.py?',
...                 punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

그러나, 가능한 한 가깝게 셸과 일치하려면, *punctuation_chars*를 사용할 때 항상 `posix`와 *whitespace_split*를 사용하는 것이 좋습니다, 이는 *wordchars*를 완전히 무효로 합니다.

최상의 효과를 얻으려면, `punctuation_chars`를 `posix=True`와 함께 설정해야 합니다. (`posix=False`가 *shlex*의 기본값임에 유의하십시오.)

34.2 posix — 가장 일반적인 POSIX 시스템 호출

이 모듈은 C 표준과 POSIX 표준(얇게 위장한 유닉스 인터페이스)에 의해 표준화된 운영 체제 기능에 대한 액세스를 제공합니다.

가용성: Unix.

이 모듈을 직접 임포트 하지 마십시오. 대신, 이 인터페이스의 이식성 있는 버전을 제공하는 모듈 *os*를 임포트 하십시오. 유닉스에서, *os* 모듈은 *posix* 인터페이스의 상위집합을 제공합니다. 비 유닉스 운영 체제에서는 *posix* 모듈을 사용할 수 없지만, *os* 인터페이스를 통해 항상 부분집합을 사용할 수 있습니다. 일단 *os*를 임포트하면, *posix* 대신 사용해도 성능 저하가 없습니다. 또한, *os*는 `os.environ`의 항목이 변경될 때 자동으로 *putenv()*를 호출하는 등의 몇 가지 추가 기능을 제공합니다.

에러는 예외로 보고됩니다; 보통 예외는 형 에러로 인한 것입니다만, 시스템 호출 때문에 보고되는 에러는 *OSError*를 발생시킵니다.

34.2.1 대용량 파일 지원

여러 운영 체제(AIX 와 Solaris 포함)는 `int`와 `long`이 32비트 값인 C 프로그래밍 모델로 인한 2 GiB보다 큰 파일에 대한 지원을 제공합니다. 이것은 일반적으로 관련 크기 및 오프셋 형을 64비트 값으로 정의하여 수행됩니다. 이러한 파일을 때로 대용량 파일 (*large files*)이라고 합니다.

`off_t`의 크기가 `long`보다 크고 `long long`이 적어도 `off_t`만큼 크면 파이썬에서 대용량 파일 지원이 활성화됩니다. 이 모드를 활성화하려면 특정 컴파일러 플래그로 파이썬을 구성하고 컴파일해야 할 수도 있습니다. 예를 들어, Solaris 2.6과 2.7에서는 다음과 같은 작업이 필요합니다:

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \
        ./configure
```

대용량 파일을 사용할 수 있는 리눅스 시스템에서, 이렇게 할 수 있습니다:

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \
        ./configure
```

34.2.2 주목할만한 모듈 내용

os 모듈 설명서에서 설명된 많은 함수 외에도, *posix*는 다음 데이터 항목을 정의합니다:

posix.**environ**

인터프리터가 시작될 때 문자열 환경을 나타내는 딕셔너리. 키와 값은 유닉스에서는 바이트열이고 윈도우에서는 str입니다. 예를 들어, `environ[b'HOME']`(윈도우에서는 `environ['HOME']`)은 홈 디렉터리의 경로명이며, C의 `getenv("HOME")`와 동등합니다.

이 딕셔너리를 수정해도 *execv()*, *popen()* 또는 *system()*에 전달되는 문자열 환경에는 영향을 주지 않습니다; 환경을 변경해야 하는 경우 `environ`을 *execve()*로 전달하거나, *system()* 이나 *popen()*의 명령 문자열에 변수 대입과 export 문장을 추가하십시오.

버전 3.2에서 변경: 유닉스에서, 키와 값은 바이트열입니다.

> **참고**
>
> *os* 모듈은 수정 시 환경을 갱신하는 `environ`의 대체 구현을 제공합니다. *os.environ*를 갱신하면 이 딕셔너리를 쓸모없게 만드는 것에 유의하십시오. *os* 모듈 버전을 사용하는 것이 *posix* 모듈에 직접 액세스하는 것보다 권장됩니다.

34.3 pwd — 암호 데이터베이스

이 모듈은 유닉스 사용자 계정과 암호 데이터베이스에 대한 액세스를 제공합니다. 모든 유닉스 버전에서 사용할 수 있습니다.

가용성: Unix, not WASI, not iOS.

암호 데이터베이스 항목은 `passwd` 구조체(아래의 어트리뷰트 필드, `<pwd.h>`를 보세요)의 멤버에 해당하는 어트리뷰트를 가진 튜플류 객체로 보고됩니다.:

인덱스	어트리뷰트	의미
0	`pw_name`	로그인 이름
1	`pw_passwd`	선택적 암호화된 암호
2	`pw_uid`	숫자 사용자 ID
3	`pw_gid`	숫자 그룹 ID
4	`pw_gecos`	사용자 이름이나 주석 필드
5	`pw_dir`	사용자 홈 디렉터리
6	`pw_shell`	사용자 명령 인터프리터

uid 및 gid 항목은 정수이고, 다른 모든 항목은 문자열입니다. 요청된 항목을 찾을 수 없으면 *KeyError*가 발생합니다.

> **참고**
>
> 전통적인 유닉스에서 필드 `pw_passwd`는 대개 DES 파생 알고리즘으로 암호화된 암호를 포함합니다. 그러나 대부분의 현대 유닉스는 소위 섀도 암호 시스템을 사용합니다. 이러한 유닉스에서 *pw_passwd* 필드는 별표 (`'*'`) 또는 문자 `'x'` 만 포함하고, 암호화된 암호는 세계(world)가 읽을 수 없는 파일 `/etc/shadow`에 저장됩니다. *pw_passwd* 필드에 유용한 것이 포함되어 있는지는 시스템에 따라 다릅니다.

다음 항목을 정의합니다:

`pwd.`**`getpwuid`**(*uid*)

주어진 숫자 사용자 ID에 대한 암호 데이터베이스 항목을 반환합니다.

`pwd.`**`getpwnam`**(*name*)

주어진 사용자 이름에 대한 암호 데이터베이스 항목을 반환합니다.

`pwd.`**`getpwall`**()

사용 가능한 모든 암호 데이터베이스 항목의 리스트를 임의의 순서로 반환합니다.

> **더 보기**
>
> **모듈 *grp***
> 그룹 데이터베이스에 대한 인터페이스, 이것과 유사합니다.

34.4 `grp` — 그룹 데이터베이스

이 모듈은 유닉스 그룹 데이터베이스에 대한 액세스를 제공합니다. 모든 유닉스 버전에서 사용할 수 있습니다.

가용성: Unix, not WASI, not Android, not iOS.

그룹 데이터베이스 항목은 `group` 구조체(아래의 어트리뷰트 필드, `<grp.h>`를 보세요)의 멤버에 해당하는 어트리뷰트를 가진 튜플류 객체로 보고됩니다.:

인덱스	어트리뷰트	의미
0	gr_name	그룹의 이름
1	gr_passwd	(암호화된) 그룹 암호; 종종 비어있습니다
2	gr_gid	숫자 그룹 ID
3	gr_mem	모든 그룹 구성원의 사용자 이름

gid는 정수고, 이름과 암호는 문자열이며, 구성원 목록은 문자열 리스트입니다. (대부분 사용자는 암호 데이터베이스에 따라 속한 그룹의 구성원으로 명시적으로 나열되지 않습니다. 완전한 멤버십 정보를 얻으려면 두 데이터베이스를 모두 확인하십시오. `+` 나 `-`로 시작하는 `gr_name`은 YP/NIS 참조일 수 있고 *getgrnam()* 이나 *getgrgid()*로 액세스하지 못할 수 있습니다.)

다음 항목을 정의합니다:

`grp.`**`getgrgid`**(*id*)

주어진 숫자 그룹 ID에 대한 그룹 데이터베이스 항목을 반환합니다. 요청된 항목을 찾을 수 없으면 *KeyError*가 발생합니다.

버전 3.10에서 변경: float나 문자열과 같은 정수가 아닌 인자에 대해 *TypeError*를 일으킵니다.

grp.**getgrnam**(*name*)

지정된 그룹 이름에 대한 그룹 데이터베이스의 항목을 반환합니다. 요청된 항목을 찾을 수 없으면 *KeyError*가 발생합니다.

grp.**getgrall**()

사용 가능한 모든 그룹 항목의 리스트를 임의의 순서로 반환합니다.

더 보기

모듈 ***pwd***
사용자 데이터베이스와의 인터페이스, 이것과 유사합니다.

34.5 termios — POSIX 스타일 tty 제어

이 모듈은 tty I/O 제어를 위한 POSIX 호출에 대한 인터페이스를 제공합니다. 이 호출에 대한 자세한 설명은 *termios(3)* 유닉스 매뉴얼 페이지를 참조하십시오. 설치 중에 구성된 POSIX *termios* 스타일 tty I/O 제어를 지원하는 유닉스 버전에서만 사용할 수 있습니다.

가용성: Unix.

이 모듈의 모든 함수는 첫 번째 인자로 파일 기술자 *fd*를 받아들입니다. `sys.stdin.fileno()`에 의해 반환된 것과 같은 정수 파일 기술자이거나, `sys.stdin` 자체와 같은 파일 객체 일 수 있습니다.

이 모듈은 여기에 제공된 함수로 작업하는 데 필요한 모든 상수도 정의합니다; 이것들은 C에 있는 것들과 같은 이름을 가집니다. 이 터미널 제어 인터페이스의 사용에 대한 자세한 내용은 시스템 설명서를 참조하십시오.

모듈은 다음 함수를 정의합니다:

termios.**tcgetattr**(*fd*)

다음과 같이 파일 기술자 *fd*에 대한 tty 어트리뷰트를 포함하는 리스트를 반환합니다: `[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]`. 여기서 *cc*는 tty 특수 문자 리스트입니다 (각기 길이 1인 문자열인데, 인덱스가 `VMIN` 과 `VTIME` 인 항목은 예외인데, 이 필드가 정의될 때 정수입니다). *cc* 배열의 인덱싱뿐만 아니라 플래그와 속도의 해석은 *termios* 모듈에 정의된 기호 상수를 사용해서 이루어져야 합니다.

termios.**tcsetattr**(*fd*, *when*, *attributes*)

파일 기술자 *fd*에 대한 tty 어트리뷰트를 *attributes*로 설정합니다. *attributes*는 *tcgetattr()*에 의해 반환된 것과 같은 리스트입니다. *when* 인자는 언제 어트리뷰트가 변경되는지를 결정합니다:

termios.**TCSANOW**

어트리뷰트를 즉시 변경합니다.

termios.**TCSADRAIN**

계류 중인 모든 출력을 전송한 후에 어트리뷰트를 변경합니다.

termios.**TCSAFLUSH**

계류 중인 모든 출력을 전송하고 계류 중인 모든 입력을 버린 후에 어트리뷰트를 변경합니다.

termios.**tcsendbreak**(*fd*, *duration*)

파일 기술자 *fd*에 브레이크(break)를 보냅니다. 0 *duration*은 0.25–0.5 초 동안 브레이크를 보냅니다; 0이 아닌 *duration*은 시스템 종속적인 의미가 있습니다.

termios.**tcdrain**(*fd*)

파일 기술자 *fd*에 기록된 모든 출력이 전송될 때까지 기다립니다.

termios.**tcflush**(*fd*, *queue*)

파일 기술자 *fd*에 계류 중인 데이터를 버립니다. *queue* 선택기는 어떤 큐인지를 지정합니다: 입력 큐는 TCIFLUSH, 출력 큐는 TCOFLUSH 또는 두 큐 모두는 TCIOFLUSH.

termios.**tcflow**(*fd*, *action*)

파일 기술자 *fd*에서 입력 또는 출력을 일시 중단하거나 다시 시작합니다. *action* 인자는 출력을 일시 중단하는 TCOOFF, 출력을 다시 시작하는 TCOON, 입력을 일시 중단하는 TCIOFF 또는 입력을 다시 시작하는 TCION이 될 수 있습니다.

termios.**tcgetwinsize**(*fd*)

Return a tuple (ws_row, ws_col) containing the tty window size for file descriptor *fd*. Requires termios.TIOCGWINSZ or termios.TIOCGSIZE.

Added in version 3.11.

termios.**tcsetwinsize**(*fd*, *winsize*)

Set the tty window size for file descriptor *fd* from *winsize*, which is a two-item tuple (ws_row, ws_col) like the one returned by *tcgetwinsize()*. Requires at least one of the pairs (termios.TIOCGWINSZ, termios.TIOCSWINSZ); (termios.TIOCGSIZE, termios.TIOCSSIZE) to be defined.

Added in version 3.11.

더 보기

모듈 *tty*
공통 터미널 제어 연산을 위한 편리 함수.

34.5.1 예제

이것은 에코가 꺼진 상태에서 암호를 묻는 함수입니다. 별도의 *tcgetattr()* 호출과 try … finally 문을 사용하여 이전 tty 어트리뷰트가 어떤 일이 발생하든 정확하게 복원되도록 하는 것에 유의하십시오:

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

34.6 tty — 터미널 제어 함수

소스 코드: Lib/tty.py

---

tty 모듈은 tty를 cbreak 및 raw 모드로 설정하는 함수를 정의합니다.

가용성: Unix.

termios 모듈이 필요하기 때문에 유닉스에서만 작동합니다.

tty 모듈은 다음 함수를 정의합니다.:

tty.**cfmakeraw**(*mode*)

Convert the tty attribute list *mode*, which is a list like the one returned by `termios.tcgetattr()`, to that of a tty in raw mode.

Added in version 3.12.

tty.**cfmakecbreak**(*mode*)

Convert the tty attribute list *mode*, which is a list like the one returned by `termios.tcgetattr()`, to that of a tty in cbreak mode.

This clears the `ECHO` and `ICANON` local mode flags in *mode* as well as setting the minimum input to 1 byte with no delay.

Added in version 3.12.

버전 3.12.2에서 변경: The `ICRNL` flag is no longer cleared. This matches Linux and macOS `stty cbreak` behavior and what `setcbreak()` historically did.

tty.**setraw**(*fd*, *when=termios.TCSAFLUSH*)

파일 기술자 *fd*의 모드를 raw로 변경합니다. *when*이 생략되면, 기본값은 `termios.TCSAFLUSH`이며 `termios.tcsetattr()`로 전달됩니다. *fd*를 raw 모드로 설정하기 전에 `termios.tcgetattr()`의 반환값을 보관합니다; 이 값을 반환합니다.

버전 3.12에서 변경: The return value is now the original tty attributes, instead of `None`.

tty.**setcbreak**(*fd*, *when=termios.TCSAFLUSH*)

파일 기술자 *fd*의 모드를 cbreak로 변경합니다. *when*이 생략되면, 기본값은 `termios.TCSAFLUSH`이며 `termios.tcsetattr()`로 전달됩니다. *fd*를 cbreak 모드로 설정하기 전에 `termios.tcgetattr()`의 반환값을 보관합니다; 이 값을 반환합니다.

This clears the `ECHO` and `ICANON` local mode flags as well as setting the minimum input to 1 byte with no delay.

버전 3.12에서 변경: The return value is now the original tty attributes, instead of `None`.

버전 3.12.2에서 변경: The `ICRNL` flag is no longer cleared. This restores the behavior of Python 3.11 and earlier as well as matching what Linux, macOS, & BSDs describe in their `stty(1)` man pages regarding cbreak mode.

더 보기

모듈 ***termios***
저수준 터미널 제어 인터페이스.

34.7 pty — 의사 터미널 유틸리티

소스 코드: Lib/pty.py

`pty` 모듈은 의사 터미널 개념을 처리하기 위한 연산을 정의합니다: 다른 프로세스를 시작하고, 그것의 제어 터미널에 프로그래밍 방식으로 쓰고 읽습니다.

가용성: Unix.

의사 터미널 처리는 플랫폼에 따라 매우 다릅니다. 이 코드는 주로 리눅스, FreeBSD, macOS 에서 테스트 되었습니다 (다른 POSIX 플랫폼에서도 작동하리라고 기대되지만, 충분히 테스트 되지는 않았습니다).

`pty` 모듈은 다음 함수를 정의합니다:

pty.**fork**()

포크. 자식의 제어 터미널을 의사 터미널에 연결합니다. 반환 값은 (pid, fd) 입니다. 자식은 *pid* 0을 받고, *fd*는 유효하지 않음에 유의하십시오. 부모의 반환 값은 자식의 *pid*이고, *fd*는 자식의 제어 터미널(또한, 자식의 표준 입력과 출력)에 연결된 파일 기술자입니다.

> **경고**
>
> On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using `urllib.request`.

pty.**openpty**()

가능하면 `os.openpty()`를 사용하고, 그렇지 않으면 일반 유닉스 시스템을 위한 에뮬레이션 코드를 사용해서 새로운 의사 터미널 쌍을 엽니다. 각각 마스터와 슬레이브인 파일 기술자 쌍 (master, slave)를 반환합니다.

pty.**spawn**(*argv*[, *master_read*[, *stdin_read*]])

프로세스를 스폰하고, 그것의 제어 터미널을 현재 프로세스의 표준 입출력과 연결합니다. 이것은 종종 제어 터미널에서 읽으려고 하는 프로그램을 조절하는 데 사용됩니다. pty 뒤에 스폰된 프로세스가 결국 종료할 것으로 기대하고, 그 때 *spawn*이 반환됩니다.

A loop copies STDIN of the current process to the child and data received from the child to STDOUT of the current process. It is not signaled to the child if STDIN of the current process closes down.

함수 *master_read*와 *stdin_read*는 그들이 읽어야 할 파일 기술자를 전달받고, 항상 바이트열을 반환해야 합니다. 자식 프로세스가 종료하기 전에 spawn이 강제로 반환되게 하려면 파일의 끝을 알리기 위해 빈 바이트열을 반환해야 합니다.

두 함수의 기본 구현은 함수가 호출될 때마다 최대 1024바이트를 읽고 반환합니다. *master_read* 콜백으로 의사 터미널의 마스터 파일 기술자가 전달되어 자식 프로세스의 출력을 읽으며, *stdin_read*는 파일 기술자 0을 전달받아, 부모 프로세스의 표준 입력을 읽습니다.

두 콜백 중 하나가 빈 바이트열을 반환하는 것은 파일 끝(EOF) 조건으로 해석되며, 그 이후로 해당 콜백은 호출되지 않습니다. *stdin_read*가 EOF 신호를 보내면 제어 터미널은 더는 부모 프로세스나 자식 프로세스와 통신할 수 없습니다. 자식 프로세스가 입력 없이 종료하지 않는 한, *spawn*은 영원히 반복됩니다. *master_read*가 EOF 신호를 보내면 같은 동작으로 이어집니다 (적어도 리눅스에서는).

자식 프로세스에 대한 `os.waitpid()`로부터 온 종료 상태 값을 반환합니다.

`os.waitstatus_to_exitcode()`를 사용하여 종료 상태를 종료 코드로 변환할 수 있습니다.

인자 argv로 감사 이벤트 pty.spawn을 발생시킵니다.

버전 3.4에서 변경: 이제 `spawn()`은 자식 프로세스에 대한 `os.waitpid()`로부터 온 상태 값을 반환합니다.

34.7.1 예제

다음 프로그램은 유닉스 명령 `script(1)`과 유사하게 동작하며, 의사 터미널을 사용하여 터미널 세션의 모든 입력과 출력을 "typescript" 에 기록합니다.

```
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
parser.add_argument('-a', dest='append', action='store_true')
parser.add_argument('-p', dest='use_python', action='store_true')
parser.add_argument('filename', nargs='?', default='typescript')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
options = parser.parse_args()

shell = sys.executable if options.use_python else os.environ.get('SHELL', 'sh')
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
        data = os.read(fd, 1024)
        script.write(data)
        return data

    print('Script started, file is', filename)
    script.write(('Script started on %s\n' % time.asctime()).encode())

    pty.spawn(shell, read)

    script.write(('Script done on %s\n' % time.asctime()).encode())
    print('Script done, file is', filename)
```

34.8 fcntl — fcntl과 ioctl 시스템 호출

이 모듈은 파일 기술자에 대한 파일과 I/O 제어를 수행합니다. `fcntl()` 과 `ioctl()` 유닉스 루틴에 대한 인터페이스입니다. 자세한 설명은 *fcntl(2)* 과 *ioctl(2)* 유닉스 매뉴얼 페이지를 참조하십시오.

가용성: Unix, not WASI.

이 모듈의 모든 함수는 첫 번째 인자로 파일 기술자 *fd* 를 받아들입니다. 이것은 `sys.stdin.fileno()` 에 의해 반환된 것과 같은 정수 파일 기술자이거나 `sys.stdin` 자체와 같은 *io.IOBase* 객체일 수 있습니다. 이 객체는 실제 파일 기술자를 반환하는 *fileno()* 를 제공합니다.

버전 3.3에서 변경: 이 모듈의 연산은 *IOError* 를 발생시켰는데, 이제는 *OSError* 를 발생시킵니다.

버전 3.8에서 변경: `fcntl` 모듈에는 이제 *os.memfd_create()* 파일 기술자를 봉인(seal)하기 위한 `F_ADD_SEALS`, `F_GET_SEALS` 및 `F_SEAL_*` 상수가 포함됩니다.

버전 3.9에서 변경: macOS에서, `fcntl` 모듈은 파일 기술자에서 파일의 경로를 얻는 `F_GETPATH` 상수를 노출합니다. 리눅스(>=3.15)에서, `fcntl` 모듈은 열린 파일 기술 록(open file description locks)으로 작업할 때 사용되는 `F_OFD_GETLK`, `F_OFD_SETLK` 및 `F_OFD_SETLKW` 상수를 노출합니다.

버전 3.10에서 변경: On Linux >= 2.6.11, the `fcntl` module exposes the `F_GETPIPE_SZ` and `F_SETPIPE_SZ` constants, which allow to check and modify a pipe' s size respectively.

버전 3.11에서 변경: On FreeBSD, the `fcntl` module exposes the `F_DUP2FD` and `F_DUP2FD_CLOEXEC` constants, which allow to duplicate a file descriptor, the latter setting `FD_CLOEXEC` flag in addition.

버전 3.12에서 변경: On Linux >= 4.5, the *fcntl* module exposes the `FICLONE` and `FICLONERANGE` constants, which allow to share some data of one file with another file by reflinking on some filesystems (e.g., btrfs, OCFS2, and XFS). This behavior is commonly referred to as "copy-on-write".

버전 3.13에서 변경: On Linux >= 2.6.32, the `fcntl` module exposes the `F_GETOWN_EX`, `F_SETOWN_EX`, `F_OWNER_TID`, `F_OWNER_PID`, `F_OWNER_PGRP` constants, which allow to direct I/O availability signals to a specific thread, process, or process group. On Linux >= 4.13, the `fcntl` module exposes the `F_GET_RW_HINT`, `F_SET_RW_HINT`, `F_GET_FILE_RW_HINT`, `F_SET_FILE_RW_HINT`, and `RWH_WRITE_LIFE_*` constants, which allow to inform the kernel about the relative expected lifetime of writes on a given inode or via a particular open file description. On Linux >= 5.1 and NetBSD, the `fcntl` module exposes the `F_SEAL_FUTURE_WRITE` constant for use with `F_ADD_SEALS` and `F_GET_SEALS` operations. On FreeBSD, the `fcntl` module exposes the `F_READAHEAD`, `F_ISUNIONSTACK`, and `F_KINFO` constants. On macOS and FreeBSD, the `fcntl` module exposes the `F_RDAHEAD`

constant. On NetBSD and AIX, the `fcntl` module exposes the `F_CLOSEM` constant. On NetBSD, the `fcntl` module exposes the `F_MAXFD` constant. On macOS and NetBSD, the `fcntl` module exposes the `F_GETNOSIGPIPE` and `F_SETNOSIGPIPE` constant.

버전 3.14에서 변경: On Linux >= 6.1, the `fcntl` module exposes the `F_DUPFD_QUERY` to query a file descriptor pointing to the same file.

이 모듈은 다음 함수를 정의합니다:

`fcntl.`**`fcntl`**(*fd, cmd, arg=0, /*)

Perform the operation *cmd* on file descriptor *fd* (file objects providing a `fileno()` method are accepted as well). The values used for *cmd* are operating system dependent, and are available as constants in the `fcntl` module, using the same names as used in the relevant C header files. The argument *arg* can either be an integer value, a *bytes-like object*, or a string. The type and size of *arg* must match the type and size of the argument of the operation as specified in the relevant C documentation.

When *arg* is an integer, the function returns the integer return value of the C `fcntl()` call.

When the argument is bytes-like object, it represents a binary structure, for example, created by `struct.pack()`. A string value is encoded to binary using the UTF-8 encoding. The binary data is copied to a buffer whose address is passed to the C `fcntl()` call. The return value after a successful call is the contents of the buffer, converted to a `bytes` object. The length of the returned object will be the same as the length of the *arg* argument. This is limited to 1024 bytes.

`fcntl()` 호출이 실패하면, `OSError`가 발생합니다.

> **참고**
>
> If the type or the size of *arg* does not match the type or size of the argument of the operation (for example, if an integer is passed when a pointer is expected, or the information returned in the buffer by the operating system is larger than 1024 bytes), this is most likely to result in a segmentation violation or a more subtle data corruption.

인자 `fd`, `cmd`, `arg`로 감사 이벤트 `fcntl.fcntl`을 발생시킵니다.

버전 3.14에서 변경: Add support of arbitrary *bytes-like objects*, not only `bytes`.

`fcntl.`**`ioctl`**(*fd, request, arg=0, mutate_flag=True, /*)

이 함수는 인자 처리가 훨씬 더 복잡하다는 점을 제외하면, `fcntl()` 함수와 같습니다.

The *request* parameter is limited to values that can fit in 32-bits or 64-bits, depending on the platform. Additional constants of interest for use as the *request* argument can be found in the `termios` module, under the same names as used in the relevant C header files.

The parameter *arg* can be an integer, a *bytes-like object*, or a string. The type and size of *arg* must match the type and size of the argument of the operation as specified in the relevant C documentation.

If *arg* does not support the read-write buffer interface or the *mutate_flag* is false, behavior is as for the `fcntl()` function.

If *arg* supports the read-write buffer interface (like `bytearray`) and *mutate_flag* is true (the default), then the buffer is (in effect) passed to the underlying `ioctl()` system call, the latter' s return code is passed back to the calling Python, and the buffer' s new contents reflect the action of the `ioctl()`. This is a slight simplification, because if the supplied buffer is less than 1024 bytes long it is first copied into a static buffer 1024 bytes long which is then passed to `ioctl()` and copied back into the supplied buffer.

`ioctl()` 호출이 실패하면, `OSError` 예외가 발생합니다.

> **참고**
>
> If the type or size of *arg* does not match the type or size of the operation' s argument (for example, if an integer is passed when a pointer is expected, or the information returned in the buffer by the operating

system is larger than 1024 bytes, or the size of the mutable bytes-like object is too small), this is most likely to result in a segmentation violation or a more subtle data corruption.

예제:

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "  "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```

인자 `fd`, `request`, `arg`로 감사 이벤트 `fcntl.ioctl`을 발생시킵니다.

버전 3.14에서 변경: The GIL is always released during a system call. System calls failing with EINTR are automatically retried.

`fcntl.`**`flock`**(*fd*, *operation*, /)

파일 기술자 *fd*(`fileno()` 메서드를 제공하는 파일 객체도 허용됩니다)에 대한 잠금 연산 *operation*을 수행합니다. 자세한 내용은 유닉스 매뉴얼 `flock(2)`를 참조하십시오. (일부 시스템에서는, 이 함수가 `fcntl()`를 사용하여 에뮬레이트됩니다.)

`flock()` 호출이 실패하면, `OSError` 예외가 발생합니다.

인자 `fd`, `operation`으로 감사 이벤트 `fcntl.flock`을 발생시킵니다.

`fcntl.`**`lockf`**(*fd*, *cmd*, *len=0*, *start=0*, *whence=0*, /)

이것은 본질에서 `fcntl()` 잠금 호출에 대한 래퍼입니다. *fd*는 잠그거나 잠금 해제할 파일의 파일 기술자이고 (`fileno()` 메서드를 제공하는 파일 객체도 허용됩니다), *cmd*는 다음 값 중 하나입니다:

`fcntl.`**`LOCK_UN`**

기존 잠금 해제.

`fcntl.`**`LOCK_SH`**

공유 잠금 획득.

`fcntl.`**`LOCK_EX`**

배타적 잠금 획득.

`fcntl.`**`LOCK_NB`**

다른 세 개의 `LOCK_*` 상수와 비트별 OR 하여 요청을 비 블로킹으로 만듭니다.

`LOCK_NB`가 사용되고 잠금을 얻을 수 없을 때, `OSError`가 발생하고 *errno* 어트리뷰트가 `EACCES`나 `EAGAIN`으로 설정됩니다 (운영 체제에 따라 다릅니다; 이식성을 위해서 두 값을 모두 확인하십시오). 적어도 일부 시스템에서, `LOCK_EX`는 파일 기술자가 쓰기 위해 열린 파일을 참조할 때만 사용할 수 있습니다.

*len*은 잠글 바이트 수, *start*는 *whence*가 정의하는 기준으로 잠금이 시작되는 바이트 오프셋이며 *whence*는 `io.IOBase.seek()`에서와 같은데, 구체적으로 다음과 같습니다:

- `0` – 파일의 시작에 상대적 (`os.SEEK_SET`)
- `1` – 현재 버퍼 위치에 상대적 (`os.SEEK_CUR`)
- `2` – 파일의 끝에 상대적 (`os.SEEK_END`)

*start*의 기본값은 파일 시작 부분에서 시작한다는 의미인 0입니다. *len*의 기본값은 파일 끝까지 잠그는 것을 의미하는 0입니다. *whence*의 기본값도 0입니다.

인자 `fd`, `cmd`, `len`, `start`, `whence`로 감사 이벤트 `fcntl.lockf`를 발생시킵니다.

예제 (모두 SVR4 호환 시스템에서):

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

첫 번째 예제에서 반환 값 변수 *rv*는 정숫값을 저장합니다; 두 번째 예제에서는 `bytes` 객체를 저장합니다. *lockdata* 변수에 대한 구조체 배치는 시스템 종속적입니다 — 그래서 `flock()` 호출을 사용하는 것이 더 좋을 수 있습니다.

더 보기

모듈 *os*
잠금 플래그 `O_SHLOCK`과 `O_EXLOCK`이 `os` 모듈에 있으면 (BSD에만 해당합니다), `os.open()` 함수는 `lockf()`와 `flock()` 함수의 대안을 제공합니다.

34.9 resource — 자원 사용 정보

이 모듈은 프로그램에서 사용하는 시스템 자원을 측정하고 제어하기 위한 기본 메커니즘을 제공합니다.

가용성: Unix, not WASI.

기호 상수는 특정 시스템 자원을 지정하고 현재 프로세스나 그 자식들에 대한 사용 정보를 요청하는 데 사용됩니다.

시스템 호출(syscall) 실패 시 `OSError`가 발생합니다.

exception resource.**error**
폐지된 `OSError`의 별칭.

버전 3.3에서 변경: **PEP 3151**에 따라, 이 클래스는 `OSError`의 별칭이 되었습니다.

34.9.1 자원 제한

아래 설명된 `setrlimit()` 함수를 사용하여 자원 사용량을 제한 할 수 있습니다. 각 자원은 제한의 쌍으로 제어됩니다: 소프트 제한과 하드 제한. 소프트 제한은 현재 제한이며, 시간이 지남에 따라 프로세스에 의해 낮아지거나 높아질 수 있습니다. 소프트 제한은 하드 제한을 초과할 수 없습니다. 하드 제한은 소프트 제한보다 큰 값으로 낮출 수 있지만, 높일 수는 없습니다. (슈퍼 유저의 유효 UID를 갖는 프로세스만 하드 제한을 높일 수 있습니다.)

제한될 수 있는 구체적인 자원은 시스템에 따라 다릅니다. `getrlimit(2)` 매뉴얼 페이지에 설명되어 있습니다. 아래에 나열된 자원은 하부 운영 체제에서 지원할 때 지원됩니다; 운영 체제에서 검사하거나 제어할 수 없는 자원은 해당 플랫폼에서는 이 모듈에서 정의되지 않습니다.

resource.**RLIM_INFINITY**
무제한 자원의 제한을 나타내는 데 사용되는 상수.

resource.**getrlimit**(*resource*)
*resource*의 현재 소프트와 하드 제한인 튜플 `(soft, hard)`를 반환합니다. 유효하지 않은 resource가 지정되면 `ValueError`가 발생하고, 하부 시스템 호출이 예기치 않게 실패하면 `error`가 발생합니다.

resource.**setrlimit**(*resource*, *limits*)

*resource*의 새로운 소비 제한을 설정합니다. *limits* 인자는 새로운 제한을 설명하는 두 정수의 튜플 (soft, hard) 이어야 합니다. *RLIM_INFINITY* 값을 사용하여 무제한 제한을 요청할 수 있습니다.

유효하지 않은 resource가 지정되거나, 새 소프트 제한이 하드 제한을 초과하거나, 프로세스가 하드 제한을 높이려고 시도하면 *ValueError*가 발생합니다. 해당 자원의 하드나 시스템 제한이 무제한이 아닐 때 *RLIM_INFINITY* 제한을 지정하면 *ValueError*가 발생합니다. 슈퍼 유저의 유효 UID를 갖는 프로세스는 무제한을 포함하여 임의의 유효한 제한 값을 요청할 수 있지만, 요청된 제한이 시스템이 부과한 제한을 초과하면 *ValueError*가 여전히 발생합니다.

하부 시스템 호출이 실패하면 setrlimit도 *error*를 발생시킬 수 있습니다.

VxWorks는 *RLIMIT_NOFILE* 설정만 지원합니다.

인자 resource, limits로 감사 이벤트 resource.setrlimit를 발생시킵니다.

resource.**prlimit**(*pid*, *resource*[, *limits*])

하나의 함수에서 *setrlimit()*와 *getrlimit()*를 결합하고 임의 프로세스의 자원 제한을 가져오고 설정하도록 지원합니다. *pid*가 0이면, 호출은 현재 프로세스에 적용됩니다. *resource*와 *limits*는 *limits*가 선택적이라는 점을 제외하고 *setrlimit()*와 같은 의미입니다.

*limits*가 제공되지 않으면 함수는 프로세스 *pid*의 *resource* 제한을 반환합니다. *limits*가 제공되면 프로세스의 *resource* 제한이 설정되고 이전 자원 제한이 반환됩니다.

*pid*를 찾을 수 없으면 *ProcessLookupError*가 발생하고 사용자가 프로세스에 대해 CAP_SYS_RESOURCE가 없으면 *PermissionError*가 발생합니다.

인자 pid, resource, limits로 감사 이벤트 resource.prlimit를 발생시킵니다.

가용성: Linux >= 2.6.36 with glibc >= 2.13.

Added in version 3.4.

이 기호들은 *setrlimit()*와 *getrlimit()* 함수를 사용하여 소비를 제어할 수 있는 아래 설명된 자원을 정의합니다. 이 기호의 값은 정확히 C 프로그램에서 사용하는 상수입니다.

*getrlimit(2)*에 관한 유닉스 매뉴얼 페이지는 사용 가능한 자원을 나열합니다. 모든 시스템이 같은 자원을 나타내는 데 같은 기호나 같은 값을 사용하는 것은 아닙니다. 이 모듈은 플랫폼 차이를 감추려고 시도하지 않습니다 — 플랫폼에서 정의되지 않은 기호는 해당 플랫폼에서 이 모듈에서 제공되지 않습니다.

resource.**RLIMIT_CORE**

현재 프로세스가 만들 수 있는 코어(core) 파일의 최대 크기 (바이트). 전체 프로세스 이미지를 담기 위해 더 큰 코어가 필요할 때 부분 코어 파일이 생성될 수 있습니다.

resource.**RLIMIT_CPU**

프로세스가 사용할 수 있는 최대 프로세서 시간 (초). 이 제한을 초과하면, SIGXCPU 시그널이 프로세스로 전송됩니다. (이 시그널을 포착하고, 열려있는 파일을 디스크로 플러시 하는 등 유용한 작업을 수행하는 방법에 대한 정보는 *signal* 모듈 설명서를 참조하십시오.)

resource.**RLIMIT_FSIZE**

프로세스가 만들 수 있는 파일의 최대 크기.

resource.**RLIMIT_DATA**

프로세스 힙(heap)의 최대 크기 (바이트).

resource.**RLIMIT_STACK**

현재 프로세스에 대한 호출 스택의 최대 크기 (바이트). 이것은 다중 스레드 프로세스에서 메인 스레드의 스택에만 영향을 줍니다.

resource.**RLIMIT_RSS**

프로세스에서 사용할 수 있는 최대 상주 집합(resident set) 크기.

resource.**RLIMIT_NPROC**

현재 프로세스가 만들 수 있는 최대 프로세스 수.

resource.**RLIMIT_NOFILE**
현재 프로세스에 대한 열린 파일 기술자의 최대 수.

resource.**RLIMIT_OFILE**
*RLIMIT_NOFILE*의 BSD 이름.

resource.**RLIMIT_MEMLOCK**
메모리에 잠겨 있을 수 있는 최대 주소 공간.

resource.**RLIMIT_VMEM**
프로세스가 차지할 수 있는 가장 큰 매핑된 메모리(mapped memory) 영역.

가용성: FreeBSD >= 11.

resource.**RLIMIT_AS**
프로세스에서 사용할 수 있는 주소 공간의 최대 영역 (바이트).

resource.**RLIMIT_MSGQUEUE**
POSIX 메시지 큐에 할당할 수 있는 바이트 수.

가용성: Linux >= 2.6.8.

Added in version 3.4.

resource.**RLIMIT_NICE**
프로세스의 나이스(nice) 수준의 상한 (20 - rlim_cur로 계산됩니다).

가용성: Linux >= 2.6.12.

Added in version 3.4.

resource.**RLIMIT_RTPRIO**
실시간 우선순위의 상한.

가용성: Linux >= 2.6.12.

Added in version 3.4.

resource.**RLIMIT_RTTIME**
프로세스가 블로킹 시스템 호출 없이 실시간 스케줄링 하에서 소비할 수 있는 CPU 시간의 시간제한 (마이크로초).

가용성: Linux >= 2.6.25.

Added in version 3.4.

resource.**RLIMIT_SIGPENDING**
프로세스가 큐에 넣을 수 있는 시그널 수입니다.

가용성: Linux >= 2.6.8.

Added in version 3.4.

resource.**RLIMIT_SBSIZE**
이 사용자의 소켓 버퍼 사용량의 최대 크기 (바이트). 이것은 이 사용자가 모든 시점에 보유할 수 있는 네트워크 메모리양과 mbuf들의 양을 제한합니다.

가용성: FreeBSD.

Added in version 3.4.

resource.**RLIMIT_SWAP**
이 사용자 ID의 모든 프로세스에서 예약하거나 사용할 수 있는 스와프 공간의 최대 크기 (바이트). 이 제한은 vm.overcommit sysctl의 비트 1이 설정되었을 때만 적용됩니다. 이 sysctl에 대한 자세한 설명은 tuning(7)를 참조하십시오.

가용성: FreeBSD.

Added in version 3.4.

resource.**RLIMIT_NPTS**

이 사용자 ID로 만들어지는 최대 의사 터미널(pseudo-terminal) 수.

가용성: FreeBSD.

Added in version 3.4.

resource.**RLIMIT_KQUEUES**

이 사용자 ID로 만들 수 있는 최대 kqueue 수.

가용성: FreeBSD >= 11.

Added in version 3.10.

34.9.2 자원 사용량

이 함수는 자원 사용량 정보를 조회하는 데 사용됩니다:

resource.**getrusage**(*who*)

이 함수는 *who* 매개 변수에 지정된 대로 현재 프로세스나 그 자식이 소비한 자원을 설명하는 객체를 반환합니다. *who* 매개 변수는 아래에 설명된 `RUSAGE_*` 상수 중 하나를 사용하여 지정해야 합니다.

간단한 예:

```
from resource import *
import time

# CPU 병목이 아닌 작업
time.sleep(3)
print(getrusage(RUSAGE_SELF))

# CPU 병목 작업
for i in range(10 ** 8):
    _ = 1 + 1
print(getrusage(RUSAGE_SELF))
```

반환 값의 필드는 각각 특정 시스템 자원이 어떻게 사용되었는지를 설명합니다. 예를 들어, 사용자 모드로 실행에 든 시간이나 프로세스가 주 메모리에서 스와프된 횟수. 일부 값은 클록 틱(clock tick) 내부에 의존합니다, 예를 들어, 프로세스에서 사용 중인 메모리양.

이전 버전과의 호환성을 위해, 반환 값은 16개 요소의 튜플로 액세스 할 수도 있습니다.

반환 값의 필드 `ru_utime`과 `ru_stime`은 각각 사용자 모드에서 실행된 시간과 시스템 모드에서 실행된 시간을 나타내는 부동 소수점 값입니다. 나머지 값은 정수입니다. 이러한 값에 대한 자세한 내용은 *getrusage(2)* 매뉴얼 페이지를 참조하십시오. 간략한 요약은 다음과 같습니다:

인덱스	필드	자원
0	ru_utime	사용자 모드의 시간 (float 초)
1	ru_stime	시스템 모드의 시간 (float 초)
2	ru_maxrss	최대 상주 집합(resident set) 크기
3	ru_ixrss	공유 메모리 크기
4	ru_idrss	비공유 메모리 크기
5	ru_isrss	비공유 스택 크기
6	ru_minflt	I/O가 필요 없는 페이지 폴트(page fault)
7	ru_majflt	I/O가 필요한 페이지 폴트(page fault)
8	ru_nswap	스와프(swap out) 수
9	ru_inblock	블록 입력 연산(block input operations)
10	ru_oublock	블록 출력 연산(block output operations)
11	ru_msgsnd	보낸 메시지
12	ru_msgrcv	받은 메시지
13	ru_nsignals	받은 시그널
14	ru_nvcsw	자발적 컨텍스트 전환(voluntary context switches)
15	ru_nivcsw	비자발적 컨텍스트 전환(involuntary context switches)

유효하지 않은 *who* 매개 변수가 지정되면 이 함수는 `ValueError`를 발생시킵니다. 비정상적인 상황에서 `error` 예외가 발생할 수도 있습니다.

`resource.`**`getpagesize`**`()`

시스템 페이지의 바이트 수를 반환합니다. (하드웨어 페이지 크기와 같을 필요는 없습니다.)

다음 `RUSAGE_*` 기호는 `getrusage()` 함수에 전달되어 제공할 프로세스 정보를 지정합니다.

`resource.`**`RUSAGE_SELF`**

호출하는 프로세스가 소비한 자원을 요청하기 위해 `getrusage()`로 전달합니다. 이는 프로세스의 모든 스레드가 사용하는 자원의 합계입니다.

`resource.`**`RUSAGE_CHILDREN`**

호출하는 프로세스의 종료되어 기다리고 있는 자식 프로세스에서 소비한 자원을 요청하기 위해 `getrusage()`로 전달합니다.

`resource.`**`RUSAGE_BOTH`**

현재 프로세스와 자식 프로세스 모두에서 소비한 자원을 요청하기 위해 `getrusage()`로 전달합니다. 모든 시스템에서 사용 가능한 것은 아닙니다.

`resource.`**`RUSAGE_THREAD`**

현재 스레드가 소비한 자원을 요청하기 위해 `getrusage()`에 전달합니다. 모든 시스템에서 사용 가능한 것은 아닙니다.

Added in version 3.2.

34.10 `syslog` — 유닉스 syslog 라이브러리 루틴

이 모듈은 유닉스 `syslog` 라이브러리 루틴에 대한 인터페이스를 제공합니다. `syslog` 기능에 대한 자세한 설명은 유닉스 매뉴얼 페이지를 참조하십시오.

가용성: Unix, not WASI, not iOS.

이 모듈은 시스템 `syslog` 계열의 루틴을 감쌉니다. syslog 서버와 통신할 수 있는 순수한 파이썬 라이브러리는 `logging.handlers` 모듈에서 `SysLogHandler`로 제공됩니다.

모듈은 다음 함수를 정의합니다:

`syslog.`**`syslog`**`(`*message*`)`

syslog.**syslog**(*priority*, *message*)

message 문자열을 시스템 로거로 보냅니다. 필요하면 끝에 줄 넘김이 추가됩니다. 각 메시지에는 시설(*facility*) 과 수준(*level*)으로 구성된 우선순위(priority)로 꼬리표가 붙습니다. 선택적 *priority* 인자(기본값은 `LOG_INFO`)는 메시지 우선순위를 결정합니다. 시설이 논리합(`LOG_INFO | LOG_USER`)을 사용하여 *priority*로 인코딩되지 않으면, `openlog()` 호출에 지정된 값이 사용됩니다.

`syslog()` 호출 이전에 `openlog()`가 호출되지 않았으면, 인자 없이 `openlog()`가 호출됩니다.

인자 `priority`, `message`로 감사 이벤트 `syslog.syslog`를 발생시킵니다.

버전 3.2에서 변경: In previous versions, `openlog()` would not be called automatically if it wasn' t called prior to the call to `syslog()`, deferring to the syslog implementation to call `openlog()`.

버전 3.12에서 변경: This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) `openlog()` must be called in the main interpreter before `syslog()` may be used in a subinterpreter. Otherwise it will raise `RuntimeError`.

syslog.**openlog**([*ident*[, *logoption*[, *facility*]]])

후속 `syslog()` 호출의 로깅 옵션은 `openlog()`를 호출하여 설정할 수 있습니다. 로그가 현재 열려 있지 않으면 `syslog()`는 인자 없이 `openlog()`를 호출합니다.

선택적 *ident* 키워드 인자는 모든 메시지 앞에 추가되는 문자열이며, 기본값은 선행 경로 구성 요소가 제거된 `sys.argv[0]` 입니다. 선택적 *logoption* 키워드 인자(기본값은 0)는 비트 필드입니다 – 결합할 수 있는 가능한 값은 아래를 보십시오. 선택적 *facility* 키워드 인자(기본값은 `LOG_USER`)는 명시적으로 인코드 된 시설이 없는 메시지에 대한 기본 시설을 설정합니다.

인자 `ident`, `logoption`, `facility`로 감사 이벤트 `syslog.openlog`를 발생시킵니다.

버전 3.2에서 변경: 이전 버전에서는, 키워드 인자가 허용되지 않았고, *ident*가 필수였습니다.

버전 3.12에서 변경: This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) This may only be called in the main interpreter. It will raise `RuntimeError` if called in a subinterpreter.

syslog.**closelog**()

syslog 모듈값을 재설정하고 시스템 라이브러리 `closelog()`를 호출합니다.

이것은 모듈이 처음 임포트될 때처럼 작동하도록 합니다. 예를 들어, `openlog()`는 첫 번째 `syslog()` 호출에서 호출되며(`openlog()`가 아직 호출되지 않았다면), *ident* 와 기타 `openlog()` 매개 변수가 기본값으로 재설정됩니다.

인자 없이 감사 이벤트 `syslog.closelog`를 발생시킵니다.

버전 3.12에서 변경: This function is restricted in subinterpreters. (Only code that runs in multiple interpreters is affected and the restriction is not relevant for most users.) This may only be called in the main interpreter. It will raise `RuntimeError` if called in a subinterpreter.

syslog.**setlogmask**(*maskpri*)

우선순위 마스크를 *maskpri*로 설정하고 이전 마스크값을 반환합니다. *maskpri*에 설정되지 않은 우선순위 수준으로 `syslog()`를 호출하면 무시됩니다. 기본값은 모든 우선순위를 로그 하는 것입니다. 함수 `LOG_MASK(pri)`는 개별 우선순위 *pri*에 대한 마스크를 계산합니다. 함수 `LOG_UPTO(pri)`는 *pri* 까지의 모든 우선순위에 대한 마스크를 계산합니다.

인자 `maskpri`로 감사 이벤트 `syslog.setlogmask`를 발생시킵니다.

모듈은 다음 상수를 정의합니다:

syslog.**LOG_EMERG**

syslog.**LOG_ALERT**

syslog.**LOG_CRIT**

syslog.**LOG_ERR**

syslog.**LOG_WARNING**

syslog.**LOG_NOTICE**

syslog.**LOG_INFO**

syslog.**LOG_DEBUG**

우선순위 수준 (높음에서 낮음 순서).

syslog.**LOG_AUTH**

syslog.**LOG_AUTHPRIV**

syslog.**LOG_CRON**

syslog.**LOG_DAEMON**

syslog.**LOG_FTP**

syslog.**LOG_INSTALL**

syslog.**LOG_KERN**

syslog.**LOG_LAUNCHD**

syslog.**LOG_LPR**

syslog.**LOG_MAIL**

syslog.**LOG_NETINFO**

syslog.**LOG_NEWS**

syslog.**LOG_RAS**

syslog.**LOG_REMOTEAUTH**

syslog.**LOG_SYSLOG**

syslog.**LOG_USER**

syslog.**LOG_UUCP**

syslog.**LOG_LOCAL0**

syslog.**LOG_LOCAL1**

syslog.**LOG_LOCAL2**

syslog.**LOG_LOCAL3**

syslog.**LOG_LOCAL4**

syslog.**LOG_LOCAL5**

syslog.**LOG_LOCAL6**

syslog.**LOG_LOCAL7**

시설, *LOG_AUTHPRIV*, *LOG_FTP*, *LOG_NETINFO*, *LOG_REMOTEAUTH*, *LOG_INSTALL* 및 *LOG_RAS*는 `<syslog.h>`에 정의되었을 때만 제공됩니다.

버전 3.13 에서 변경: *LOG_FTP*, *LOG_NETINFO*, *LOG_REMOTEAUTH*, *LOG_INSTALL*, *LOG_RAS* 및 *LOG_LAUNCHD*를 추가했습니다.

syslog.**LOG_PID**

syslog.**LOG_CONS**

syslog.**LOG_NDELAY**

syslog.**LOG_ODELAY**

syslog.**LOG_NOWAIT**

syslog.**LOG_PERROR**

로그 옵션, *LOG_ODELAY*, *LOG_NOWAIT* 및 *LOG_PERROR*는 `<syslog.h>`에 정의되었을 때만 제공됩니다.

34.10.1 예제

간단한 예제

간단한 예제 집합:

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

일부 로그 옵션 설정의 예, 이것은 로그 메시지에 프로세스 ID를 포함하며, 메일 로깅에 사용되는 대상 시설로 메시지를 기록합니다:

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

CHAPTER 35

Modules command-line interface (CLI)

The following modules have a command-line interface.

- *ast*
- *asyncio*
- `base64`
- *calendar*
- `code`
- *compileall*
- `cProfile`: see *profile*
- *dis*
- *doctest*
- `encodings.rot_13`
- `ensurepip`
- `filecmp`
- `fileinput`
- `ftplib`
- *gzip*
- *http.server*
- `idlelib`
- *inspect*
- *json*
- *mimetypes*
- `pdb`
- *pickle*
- *pickletools*

- *platform*
- `poplib`
- *profile*
- `pstats`
- *py_compile*
- `pyclbr`
- `pydoc`
- `quopri`
- *random*
- `runpy`
- *site*
- *sqlite3*
- *symtable*
- *sysconfig*
- `tabnanny`
- *tarfile*
- `this`
- *timeit*
- *tokenize*
- *trace*
- `turtledemo`
- *unittest*
- *uuid*
- `venv`
- `webbrowser`
- *zipapp*
- *zipfile*

See also the Python command-line interface.

CHAPTER 36

대체된 모듈

이 장에서 설명하는 모듈은 대부분의 사용 사례에서 다른 모듈로 대체되었으며, 주로 하위 호환성을 유지하기 위해 유지됩니다.

Modules may appear in this chapter because they only cover a limited subset of a problem space, and a more generally applicable solution is available elsewhere in the standard library (for example, `getopt` covers the very specific task of "mimic the C `getopt()` API in Python", rather than the broader command line option parsing and argument parsing capabilities offered by `optparse` and `argparse`).

Alternatively, modules may appear in this chapter because they are deprecated outright, and awaiting removal in a future release, or they are *soft deprecated* and their use is actively discouraged in new projects. With the removal of various obsolete modules through **PEP 594**, there are currently no modules in this latter category.

36.1 `getopt` — 명령 줄 옵션용 C 스타일 구문 분석기

소스 코드: Lib/getopt.py

> **참고**
>
> This module is considered feature complete. A more declarative and extensible alternative to this API is provided in the `optparse` module. Further functional enhancements for command line parameter processing are provided either as third party modules on PyPI, or else as features in the `argparse` module.

이 모듈은 스크립트가 `sys.argv`에 있는 명령 줄 인자를 구문 분석하는 데 도움이 됩니다. 유닉스 `getopt()` 함수와 같은 규칙을 지원합니다 ('`-`' 와 '`--`' 형식의 인자의 특수한 의미를 포함합니다). 선택적인 세 번째 인자를 통해 GNU 소프트웨어가 지원하는 것과 유사한 긴 옵션을 사용할 수 있습니다.

Users who are unfamiliar with the Unix `getopt()` function should consider using the `argparse` module instead. Users who are familiar with the Unix `getopt()` function, but would like to get equivalent behavior while writing less code and getting better help and error messages should consider using the `optparse` module. See *Choosing an argument parsing library* for additional details.

이 모듈은 두 가지 함수와 예외를 제공합니다:

getopt.**getopt**(*args*, *shortopts*, *longopts*=[])

Parses command line options and parameter list. *args* is the argument list to be parsed, without the leading reference to the running program. Typically, this means `sys.argv[1:]`. *shortopts* is the string of option letters that the script wants to recognize, with options that require an argument followed by a colon (`':'`) and options that accept an optional argument followed by two colons (`'::'`); i.e., the same format that Unix `getopt()` uses.

> **참고**
>
> GNU `getopt()` 와는 달리, 옵션이 아닌 인자 다음에 오는 모든 인자는 옵션이 아닌 것으로 간주합니다. 이는 비 GNU 유닉스 시스템이 작동하는 방식과 비슷합니다.

longopts, if specified, must be a list of strings with the names of the long options which should be supported. The leading `'--'` characters should not be included in the option name. Long options which require an argument should be followed by an equal sign (`'='`). Long options which accept an optional argument should be followed by an equal sign and question mark (`'=?'`). To accept only long options, *shortopts* should be an empty string. Long options on the command line can be recognized so long as they provide a prefix of the option name that matches exactly one of the accepted options. For example, if *longopts* is `['foo', 'frob']`, the option `--fo` will match as `--foo`, but `--f` will not match uniquely, so *GetoptError* will be raised.

반환 값은 두 요소로 구성됩니다: 첫 번째는 `(option, value)` 쌍의 리스트입니다; 두 번째는 옵션 리스트가 제거된 후 남겨진 프로그램 인자 리스트입니다 (이것은 *args*의 후행 슬라이스입니다). 반환된 각 옵션-값 쌍은 첫 번째 요소로 옵션을 가지며, 짧은 옵션(예를 들어, `'-x'`)은 하이픈이, 긴 옵션(예를 들어, `'--long-option'`)은 두 개의 하이픈이 접두사로 붙고, 두 번째 요소는 옵션 인자나 옵션에 인자가 없으면 빈 문자열입니다. 옵션은 발견된 순서와 같은 순서로 리스트에 나타나므로, 여러 번 나오는 것을 허용합니다. 긴 옵션과 짧은 옵션은 혼합될 수 있습니다.

버전 3.14에서 변경: Optional arguments are supported.

getopt.**gnu_getopt**(*args*, *shortopts*, *longopts*=[])

이 함수는 기본적으로 GNU 스타일 스캔 모드가 사용된다는 점을 제외하고는 *getopt()* 처럼 작동합니다. 이것은 옵션과 옵션이 아닌 인자가 섞일 수 있음을 뜻합니다. *getopt()* 함수는 옵션이 아닌 인자가 발견되자마자 옵션 처리를 중지합니다.

옵션 문자열의 첫 번째 문자가 `'+'` 이거나, 환경 변수 `POSIXLY_CORRECT` 가 설정되면, 옵션이 아닌 인자를 만나자마자 옵션 처리가 중지됩니다.

If the first character of the option string is `'-'`, non-option arguments that are followed by options are added to the list of option-and-value pairs as a pair that has `None` as its first element and the list of non-option arguments as its second element. The second element of the `gnu_getopt()` result is a list of program arguments after the last option.

버전 3.14에서 변경: Support for returning intermixed options and non-option arguments in order.

exception getopt.**GetoptError**

인자 목록에 인식할 수 없는 옵션이 있거나 인자가 필요한 옵션에 아무것도 주어지지 않으면 발생합니다. 예외에 대한 인자는 에러의 원인을 나타내는 문자열입니다. 긴 옵션의 경우, 인자를 요구하지 않는 옵션에 인자가 주어질 때도 이 예외를 발생시킵니다. 어트리뷰트 `msg` 와 `opt` 는 에러 메시지와 관련 옵션을 제공합니다; 예외와 관련된 특정 옵션이 없으면 `opt` 는 빈 문자열입니다.

exception getopt.**error**

*GetoptError*의 별칭; 과거 호환성을 위한 것입니다.

유닉스 스타일 옵션만 사용하는 예제:

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

긴 옵션 이름을 사용하는 것도 똑같이 간단합니다:

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', '
↪')]
>>> args
['a1', 'a2']
```

Optional arguments should be specified explicitly:

```
>>> s = '-Con -C --color=off --color a1 a2'
>>> args = s.split()
>>> args
['-Con', '-C', '--color=off', '--color', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'C::', ['color=?'])
>>> optlist
[('-C', 'on'), ('-C', ''), ('--color', 'off'), ('--color', '')]
>>> args
['a1', 'a2']
```

The order of options and non-option arguments can be preserved:

```
>>> s = 'a1 -x a2 a3 a4 --long a5 a6'
>>> args = s.split()
>>> args
['a1', '-x', 'a2', 'a3', 'a4', '--long', 'a5', 'a6']
>>> optlist, args = getopt.gnu_getopt(args, '-x:', ['long='])
>>> optlist
[(None, ['a1']), ('-x', 'a2'), (None, ['a3', 'a4']), ('--long', 'a5')]
>>> args
['a6']
```

스크립트에서, 일반적인 사용법은 다음과 같습니다:

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    except getopt.GetoptError as err:
        # 도움말 정보를 출력하고 종료합니다:
        print(err)  # "option -a not recognized" 같은 것을 출력합니다
        usage()
        sys.exit(2)
    output = None
    verbose = False
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"
    process(args, output=output, verbose=verbose)

if __name__ == "__main__":
    main()
```

optparse 모듈을 사용하면 더 적은 코드로, 더욱 유용한 도움말과 에러 메시지를 제공하는 동등한 명령 줄 인터페이스를 만들 수 있습니다:

```
import optparse

if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option('-o', '--output')
    parser.add_option('-v', dest='verbose', action='store_true')
    opts, args = parser.parse_args()
    process(args, output=opts.output, verbose=opts.verbose)
```

이 경우와 거의 동등한 명령 줄 인터페이스를 *argparse* 모듈을 사용해서도 만들 수 있습니다:

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    parser.add_argument('rest', nargs='*')
    args = parser.parse_args()
    process(args.rest, output=args.output, verbose=args.verbose)
```

See *Choosing an argument parsing library* for details on how the `argparse` version of this code differs in behaviour from the `optparse` (and `getopt`) version.

더 보기

모듈 ***optparse***
선언적 명령 줄 옵션 구문 분석.

모듈 ***argparse***
더 견해가 강한 명령 줄 옵션과 인자 구문 분석 라이브러리.

CHAPTER 37

Removed Modules

The modules described in this chapter have been removed from the Python standard library. They are documented here to help people find replacements.

37.1 aifc — AIFF와 AIFC 파일 읽고 쓰기

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

aifc 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.2 asynchat — 비동기 소켓 명령/응답 처리기

Deprecated since version 3.6, removed in version 3.12.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.6 에서 폐지된 후 파이썬 3.12 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

응용은 대신 *asyncio* 모듈을 사용해야 합니다.

asynchat 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.11 입니다.

37.3 asyncore — 비동기 소켓 처리기

Deprecated since version 3.6, removed in version 3.12.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.6 에서 폐지된 후 파이썬 3.12 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

응용은 대신 *asyncio* 모듈을 사용해야 합니다.

asyncore 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.11 입니다.

37.4 audioop — 원시 오디오 데이터 조작

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

audioop 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.5 cgi — 공통 게이트웨이 인터페이스 지원

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

PyPI의 모듈 포크를 대신 사용할 수 있습니다: legacy-cgi. 이것은 cgi 모듈의 복사본인데, 더 이상 파이썬 코어 팀에서 유지 관리하거나 지원하지 않습니다.

cgi 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.6 cgitb — CGI 스크립트를 위한 트레이스백 관리자

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

PyPI의 모듈 포크를 대신 사용할 수 있습니다: legacy-cgi. 이것은 cgi 모듈의 복사본인데, 더 이상 파이썬 코어 팀에서 유지 관리하거나 지원하지 않습니다.

cgitb 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.7 chunk — IFF 청크된 데이터 읽기

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

chunk 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.8 crypt — 유닉스 비밀번호 확인 함수

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

응용은 표준 라이브러리의 *hashlib* 모듈을 사용할 수 있습니다. 다른 가능한 대체품은 PyPI 에 있는 타사 라이브러리들입니다: legacycrypt, bcrypt, argon2-cffi 또는 passlib. 이것들은 파이썬 코어 팀에서 지원하거나 유지 관리하지 않습니다.

crypt 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.9 distutils — 파이썬 모듈 빌드와 설치

Deprecated since version 3.10, removed in version 3.12.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.10 에서 폐지된 후 파이썬 3.12 에서 제거되었습니다. 제거는 **PEP 632** 에서 결정되었고, 마이그레이션 지침이 있습니다.

distutils 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.11 입니다.

37.10 imghdr — 이미지 유형 판단

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

가능한 대체품은 PyPI 에 있는 타사 라이브러리들입니다: filetype, puremagic 또는 python-magic. 이 라이브러리는 파이썬 코어 팀에서 지원하거나 유지 관리하지 않습니다.

imghdr 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.11 imp — 임포트 내부 액세스

Deprecated since version 3.4, removed in version 3.12.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.4 에서 폐지된 후 파이썬 3.12 에서 제거되었습니다.

제거 공지에는 코드를 imp 에서 *importlib* 로 마이그레이션하는 방법에 대한 지침이 포함되어 있습니다.

imp 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.11 입니다.

37.12 mailcap — Mailcap 파일 처리

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

mailcap 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.13 msilib — 마이크로소프트 설치 파일 읽고 쓰기

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

msilib 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.14 nis — Sun의 NIS(옐로 페이지)에 대한 인터페이스

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

nis 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.15 nntplib — NNTP 프로토콜 클라이언트

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

nntplib 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.16 ossaudiodev — OSS 호환 오디오 장치 액세스

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

ossaudiodev 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.17 pipes — 셸 파이프라인에 대한 인터페이스

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

응용은 대신 *subprocess* 모듈을 사용해야 합니다.

pipes 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.18 smtpd — SMTP 서버

Deprecated since version 3.6, removed in version 3.12.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.6 에서 폐지된 후 파이썬 3.12 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

가능한 대체품은 타사 aiosmtpd 라이브러리입니다. 이 라이브러리는 파이썬 코어 팀에서 유지 관리하거나 지원하지 않습니다.

smtpd 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.11 입니다.

37.19 sndhdr — 음향 파일 유형 판단

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

가능한 대체품은 PyPI 에 있는 타사 라이브러리들입니다: filetype, puremagic 또는 python-magic. 이 라이브러리는 파이썬 코어 팀에서 지원하거나 유지 관리하지 않습니다.

sndhdr 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.20 spwd — 섀도 암호 데이터베이스

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

가능한 대체품은 타사 python-pam 라이브러리입니다. 이 라이브러리는 파이썬 코어 팀에서 지원하거나 유지 관리하지 않습니다.

spwd 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.21 sunau — Sun AU 파일 읽고 쓰기

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

sunau 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.22 telnetlib — 텔넷 클라이언트

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

가능한 대체품은 PyPI 에 있는 타사 라이브러리들입니다: telnetlib3 이나 Exscript. 이 라이브러리는 파이썬 코어 팀에서 지원하거나 유지 관리하지 않습니다.

telnetlib 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.23 uu — uuencode 파일 인코딩과 디코딩

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

uu 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

37.24 xdrlib — XDR 데이터 인코딩과 디코딩

Deprecated since version 3.11, removed in version 3.13.

이 모듈은 더는 파이썬 표준 라이브러리에 포함되지 않습니다. 파이썬 3.11 에서 폐지된 후 파이썬 3.13 에서 제거되었습니다. 제거는 **PEP 594** 에서 결정되었습니다.

xdrlib 모듈을 제공한 마지막 파이썬 버전은 파이썬 3.12 입니다.

CHAPTER 38

Security Considerations

The following modules have specific security considerations:

- `base64`: *base64 security considerations* in **RFC 4648**
- `hashlib`: *all constructors take a "usedforsecurity" keyword-only argument disabling known insecure and blocked algorithms*
- `http.server` is not suitable for production use, only implementing basic security checks. See the *security considerations*.
- `logging`: *Logging configuration uses eval()*
- `multiprocessing`: *Connection.recv() uses pickle*
- `pickle`: *Restricting globals in pickle*
- `random` shouldn' t be used for security purposes, use `secrets` instead
- `shelve`: *shelve is based on pickle and thus unsuitable for dealing with untrusted sources*
- `ssl`: *SSL/TLS security considerations*
- `subprocess`: *Subprocess security considerations*
- `tempfile`: *mktemp is deprecated due to vulnerability to race conditions*
- `xml`: *XML security*
- `zipfile`: *maliciously prepared .zip files can cause disk volume exhaustion*

The `-I` command line option can be used to run Python in isolated mode. When it cannot be used, the `-P` option or the `PYTHONSAFEPATH` environment variable can be used to not prepend a potentially unsafe path to `sys.path` such as the current directory, the script' s directory or an empty string.

APPENDIX A

용어집

>>>

대화형 셸의 기본 파이썬 프롬프트. 인터프리터에서 대화형으로 실행될 수 있는 코드 예에서 자주 볼 수 있습니다.

...

다음과 같은 것들을 가리킬 수 있습니다:

- 들여쓰기 된 코드 블록의 코드를 입력할 때, 쌍을 이루는 구분자 (괄호, 대괄호, 중괄호) 안에 코드를 입력할 때, 데코레이터 지정 후의 대화형 셸의 기본 파이썬 프롬프트.
- The three dots form of the *Ellipsis* object.

abstract base class (추상 베이스 클래스)

추상 베이스 클래스는 `hasattr()` 같은 다른 테크닉들이 불편하거나 미묘하게 잘못된 (예를 들어, 매직 메서드) 경우, 인터페이스를 정의하는 방법을 제공함으로써 덕 타이핑 을 보완합니다. ABC는 가상 서브 클래스를 도입하는데, 클래스를 계승하지 않으면서도 `isinstance()` 와 `issubclass()` 에 의해 감지될 수 있는 클래스들입니다; `abc` 모듈 설명서를 보세요. 파이썬에는 많은 내장 ABC 들이 따라오는데 다음과 같은 것들이 있습니다: 자료 구조 (`collections.abc` 모듈에서), 숫자 (`numbers` 모듈에서), 스트림 (`io` 모듈에서), 임포트 파인더와 로더 (`importlib.abc` 모듈에서). `abc` 모듈을 사용해서 자신만의 ABC를 만들 수도 있습니다.

annotate function

A function that can be called to retrieve the *annotations* of an object. This function is accessible as the `__annotate__` attribute of functions, classes, and modules. Annotate functions are a subset of *evaluate functions*.

annotation (어노테이션)

관습에 따라 형 힌트 로 사용되는 변수, 클래스 어트리뷰트 또는 함수 매개변수 나 반환 값과 연결된 레이블입니다.

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions can be retrieved by calling `annotationlib.get_annotations()` on modules, classes, and functions, respectively.

See *variable annotation*, *function annotation*, **PEP 484**, **PEP 526**, and **PEP 649**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

argument (인자)

함수를 호출할 때 함수 (또는 메서드) 로 전달되는 값. 두 종류의 인자가 있습니다:

- 키워드 인자 (*keyword argument*): 함수 호출 때 식별자가 앞에 붙은 인자 (예를 들어, `name=`) 또는 `**` 를 앞에 붙인 딕셔너리로 전달되는 인자. 예를 들어, 다음과 같은 `complex()` 호출에서 `3` 과 `5` 는 모두 키워드 인자입니다:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- 위치 인자 (*positional argument*): 키워드 인자가 아닌 인자. 위치 인자들은 인자 목록의 처음에 나오거나 이터러블 의 앞에 `*` 를 붙여 전달할 수 있습니다. 예를 들어, 다음과 같은 호출에서 `3` 과 `5` 는 모두 위치 인자입니다.

```
complex(3, 5)
complex(*(3, 5))
```

인자는 함수 바디의 이름 붙은 지역 변수에 대입됩니다. 이 대입에 적용되는 규칙들에 대해서는 calls 절을 보세요. 문법적으로, 어떤 표현식이건 인자로 사용될 수 있습니다; 구해진 값이 지역 변수에 대입됩니다.

용어집의 매개변수 항목과 FAQ 질문 인자와 매개변수의 차이 와 **PEP 362**도 보세요.

asynchronous context manager (비동기 컨텍스트 관리자)
`__aenter__()` 와 `__aexit__()` 메서드를 정의함으로써 `async with` 문에서 보이는 환경을 제어하는 객체. **PEP 492**로 도입되었습니다.

asynchronous generator (비동기 제너레이터)
비동기 제너레이터 이터레이터 를 돌려주는 함수. `async def` 로 정의되는 코루틴 함수처럼 보이는데, `async for` 루프가 사용할 수 있는 일련의 값들을 만드는 `yield` 표현식을 포함한다는 점이 다릅니다.

보통 비동기 제너레이터 함수를 가리키지만, 어떤 문맥에서는 비동기 제너레이터 이터레이터 를 가리킵니다. 의도하는 의미가 명확하지 않은 경우는, 완전한 용어를 써서 모호함을 없앱니다.

비동기 제너레이터 함수는 `await` 표현식과, `async for` 문과, `async with` 문을 포함할 수 있습니다.

asynchronous generator iterator (비동기 제너레이터 이터레이터)
An object created by an *asynchronous generator* function.

비동기 이터레이터 인데 `__anext__()` 를 호출하면 어웨이터블 객체를 돌려주고, 이것은 다음 `yield` 표현식 까지 비동기 제너레이터 함수의 바디를 실행합니다.

각 `yield`는 일시적으로 처리를 중단하고, (지역 변수들과 대기 중인 try-문들을 포함하는) 실행 상태를 기억합니다. 비동기 제너레이터 이터레이터 가 `__anext__()` 가 돌려주는 또 하나의 어웨이터블로 재개되면, 떠난 곳으로 복귀합니다. **PEP 492**와 **PEP 525**를 보세요.

asynchronous iterable (비동기 이터러블)
`async for` 문에서 사용될 수 있는 객체. `__aiter__()` 메서드는 비동기 이터레이터 를 돌려줘야 합니다. **PEP 492** 로 도입되었습니다.

asynchronous iterator (비동기 이터레이터)
`__aiter__()` 와 `__anext__()` 메서드를 구현하는 객체. `__anext__()` 는 어웨이터블 객체를 돌려줘야 합니다. `async for`는 `StopAsyncIteration` 예외가 발생할 때까지 비동기 이터레이터의 `__anext__()` 메서드가 돌려주는 어웨이터블을 풉니다. **PEP 492**로 도입되었습니다.

attached thread state
A *thread state* that is active for the current OS thread.

When a *thread state* is attached, the OS thread has access to the full Python C API and can safely invoke the bytecode interpreter.

Unless a function explicitly notes otherwise, attempting to call the C API without an attached thread state will result in a fatal error or undefined behavior. A thread state can be attached and detached explicitly by the user through the C API, or implicitly by the runtime, including during blocking C calls and by the bytecode interpreter in between calls.

On most builds of Python, having an attached thread state implies that the caller holds the *GIL* for the current interpreter, so only one OS thread can have an attached thread state at a given moment. In *free-threaded* builds

of Python, threads can concurrently hold an attached thread state, allowing for true parallelism of the bytecode interpreter.

attribute (어트리뷰트)
흔히 점표현식을 사용하는 이름으로 참조되는 객체와 결합한 값. 예를 들어, 객체 *o*가 어트리뷰트 *a*를 가지면, *o.a*처럼 참조됩니다.

It is possible to give an object an attribute whose name is not an identifier as defined by identifiers, for example using `setattr()`, if the object allows it. Such an attribute will not be accessible using a dotted expression, and would instead need to be retrieved with `getattr()`.

awaitable (어웨이터블)
`await` 표현식에 사용할 수 있는 객체. 코루틴 이나 `__await__()` 메서드를 가진 객체가 될 수 있습니다. **PEP 492**를 보세요.

BDFL
자비로운 종신 독재자 (Benevolent Dictator For Life), 즉 Guido van Rossum, 파이썬의 창시자.

binary file (바이너리 파일)
바이트열류 객체들을 읽고 쓸 수 있는 파일 객체. 바이너리 파일의 예로는 바이너리 모드 (`'rb'`, `'wb'` 또는 `'rb+'`) 로 열린 파일, `sys.stdin.buffer`, `sys.stdout.buffer`, `io.BytesIO` 와 `gzip.GzipFile` 의 인스턴스를 들 수 있습니다.

`str` 객체를 읽고 쓸 수 있는 파일 객체에 대해서는 텍스트 파일 도 참조하세요.

borrowed reference (빌린 참조)
In Python' s C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Calling `Py_INCREF()` on the *borrowed reference* is recommended to convert it to a *strong reference* in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The `Py_NewRef()` function can be used to create a new *strong reference*.

bytes-like object (바이트열류 객체)
bufferobjects 를 지원하고 C-연속 버퍼를 익스포트 할 수 있습니다. 여러 공통 `memoryview` 객체들은 물론이고 `bytes`, `bytearray`, `array.array` 객체들을 포함합니다. 바이트열류 객체들은 바이너리 데이터를 다루는 여러 가지 연산들에 사용될 수 있습니다; 압축, 바이너리 파일로 저장, 소켓을 통한 전송 같은 것들이 있습니다.

어떤 연산들은 바이너리 데이터가 가변적일 필요가 있습니다. 이런 경우에 설명서는 종종 "읽고-쓰기 바이트열류 객체"라고 표현합니다. 가변 버퍼 객체의 예로는 `bytearray` 와 `bytearray` 의 `memoryview` 가 있습니다. 다른 연산들은 바이너리 데이터가 불변 객체 ("읽기 전용 바이트열류 객체")에 저장되도록 요구합니다; 이런 것들의 예로는 `bytes`와 `bytes` 객체의 `memoryview` 가 있습니다.

bytecode (바이트 코드)
파이썬 소스 코드는 바이트 코드로 컴파일되는데, CPython 인터프리터에서 파이썬 프로그램의 내부 표현입니다. 바이트 코드는 `.pyc` 파일에 캐시 되어, 같은 파일을 두 번째 실행할 때 더 빨라지게 만듭니다 (소스에서 바이트 코드로의 재컴파일을 피할 수 있습니다). 이 "중간 언어" 는 각 바이트 코드에 대응하는 기계를 실행하는 가상 기계 에서 실행된다고 말합니다. 바이트 코드는 서로 다른 파이썬 가상 기계에서 작동할 것으로 기대하지도, 파이썬 배포 간에 안정적이지도 않다는 것에 주의해야 합니다.

바이트 코드 명령어들의 목록은 *dis* 모듈 설명서에 나옵니다.

callable (콜러블)
A callable is an object that can be called, possibly with a set of arguments (see *argument*), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A *function*, and by extension a *method*, is a callable. An instance of a class that implements the `__call__()` method is also a callable.

callback (콜백)
인자로 전달되는 미래의 어느 시점에서 실행될 서브 루틴 함수.

class (클래스)
사용자 정의 객체들을 만들기 위한 주형. 클래스 정의는 보통 클래스의 인스턴스를 대상으로 연산하는 메서드 정의들을 포함합니다.

class variable (클래스 변수)
클래스에서 정의되고 클래스 수준 (즉, 클래스의 인스턴스에서가 아니라) 에서만 수정되는 변수.

closure variable (클로저 변수)
A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable*:

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

complex number (복소수)
익숙한 실수 시스템의 확장인데, 모든 숫자가 실수부와 허수부의 합으로 표현됩니다. 허수부는 실수에 허수 단위(`-1`의 제곱근)를 곱한 것인데, 종종 수학에서는 `i`로, 공학에서는 `j`로 표기합니다. 파이썬은 후자의 표기법을 쓰는 복소수를 기본 지원합니다; 허수부는 `j` 접미사를 붙여서 표기합니다, 예를 들어, `3+1j`. `math` 모듈의 복소수 버전이 필요하면, `cmath`를 사용합니다. 복소수의 활용은 꽤 수준 높은 수학적 기능입니다. 필요하다고 느끼지 못한다면, 거의 확실히 무시해도 좋습니다.

context (컨텍스트)
This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a `with` statement.
- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
- A `contextvars.Context` object. Also see *current context*.

context management protocol (컨텍스트 관리 프로토콜)
The `__enter__()` and `__exit__()` methods called by the `with` statement. See **PEP 343**.

context manager (컨텍스트 관리자)
컨텍스트 관리 프로토콜을 구현하고 `with` 문에서 보이는 환경을 제어하는 객체. **PEP 343**을 참조하십시오.

context variable (컨텍스트 변수)
A variable whose value depends on which context is the *current context*. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contiguous (연속)
버퍼는 정확히 *C*-연속(*C-contiguous*)이거나 포트란 연속(*Fortran contiguous*)일 때 연속이라고 여겨집니다. 영차원 버퍼는 C-연속이면서 포트란 연속입니다. 일차원 배열에서, 항목들은 서로에 인접하고, 0에서 시작하는 오름차순 인덱스의 순서대로 메모리에 배치되어야 합니다. 다차원 C-연속 배열에서, 메모리 주소의 순서대로 항목들을 방문할 때 마지막 인덱스가 가장 빨리 변합니다. 하지만, 포트란 연속 배열에서는, 첫 번째 인덱스가 가장 빨리 변합니다.

coroutine (코루틴)
코루틴은 서브루틴의 더 일반화된 형태입니다. 서브루틴은 한 지점에서 진입하고 다른 지점에서 탈출합니다. 코루틴은 여러 다른 지점에서 진입하고, 탈출하고, 재개할 수 있습니다. 이것들은 `async def` 문으로 구현할 수 있습니다. **PEP 492**를 보세요.

coroutine function (코루틴 함수)
코루틴 객체를 돌려주는 함수. 코루틴 함수는 `async def` 문으로 정의될 수 있고, `await` 와 `async for`와 `async with` 키워드를 포함할 수 있습니다. 이것들은 **PEP 492** 에 의해 도입되었습니다.

CPython
파이썬 프로그래밍 언어의 규범적인 구현인데, python.org에서 배포됩니다. 이 구현을 Jython 이나 IronPython 과 같은 다른 것들과 구별할 필요가 있을 때 용어 "CPython" 이 사용됩니다.

current context (현재 컨텍스트)
The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

cyclic isolate
A subgroup of one or more objects that reference each other in a reference cycle, but are not referenced by objects outside the group. The goal of the *cyclic garbage collector* is to identify these groups and break the reference cycles so that the memory can be reclaimed.

decorator (데코레이터)
다른 함수를 돌려주는 함수인데, 보통 `@wrapper` 문법을 사용한 함수 변환으로 적용됩니다. 데코레이터의 흔한 예는 `classmethod()` 과 `staticmethod()` 입니다.

데코레이터 문법은 단지 편의 문법일 뿐입니다. 다음 두 함수 정의는 의미상으로 동등합니다:

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

같은 개념이 클래스에도 존재하지만, 덜 자주 쓰입니다. 데코레이터에 대한 더 자세한 내용은 함수 정의 와 클래스 정의 의 설명서를 보면 됩니다.

descriptor (디스크립터)
메서드 `__get__()` 이나 `__set__()` 이나 `__delete__()` 를 정의하는 객체. 클래스 어트리뷰트가 디스크립터일 때, 어트리뷰트 조회는 특별한 연결 작용을 일으킵니다. 보통, *a.b*를 읽거나, 쓰거나, 삭제하는데 사용할 때, *a*의 클래스 딕셔너리에서 *b* 라고 이름 붙여진 객체를 찾습니다. 하지만 *b*가 디스크립터면, 해당하는 디스크립터 메서드가 호출됩니다. 디스크립터를 이해하는 것은 파이썬에 대한 깊은 이해의 열쇠인데, 함수, 메서드, 프로퍼티, 클래스 메서드, 스태틱 메서드, 슈퍼 클래스 참조 등의 많은 기능의 기초를 이루고 있기 때문입니다.

디스크립터의 메서드들에 대한 자세한 내용은 descriptors나 디스크립터 사용법 안내서에 나옵니다.

dictionary (딕셔너리)
임의의 키를 값에 대응시키는 연관 배열 (associative array). 키는 `__hash__()` 와 `__eq__()` 메서드를 갖는 모든 객체가 될 수 있습니다. 펄에서 해시라고 부릅니다.

dictionary comprehension (딕셔너리 컴프리헨션)
이터러블에 있는 요소 전체나 일부를 처리하고 결과를 담은 딕셔너리를 반환하는 간결한 방법. `results = {n: n ** 2 for n in range(10)}`은 값 `n ** 2`에 매핑된 키 `n`을 포함하는 딕셔너리를 생성합니다. comprehensions을 참조하십시오.

dictionary view (딕셔너리 뷰)
`dict.keys()`, `dict.values()`, `dict.items()` 메서드가 돌려주는 객체들을 딕셔너리 뷰라고 부릅니다. 이것들은 딕셔너리 항목들에 대한 동적인 뷰를 제공하는데, 딕셔너리가 변경될 때, 뷰가 이 변

화를 반영한다는 뜻입니다. 딕셔너리 뷰를 완전한 리스트로 바꾸려면 `list(dictview)` 를 사용하면 됩니다. 딕셔너리 뷰 객체를 보세요.

docstring (독스트링)
클래스, 함수, 모듈에서 첫 번째 표현식으로 나타나는 문자열 리터럴. 스위트가 실행될 때는 무시되지만, 컴파일러에 의해 인지되어 둘러싼 클래스, 함수, 모듈의 `__doc__` 어트리뷰트로 삽입됩니다. 인트로스펙션을 통해 사용할 수 있으므로, 객체의 설명서를 위한 규범적인 장소입니다.

duck-typing (덕 타이핑)
올바른 인터페이스를 가졌는지 판단하는데 객체의 형을 보지 않는 프로그래밍 스타일; 대신, 단순히 메서드나 어트리뷰트가 호출되거나 사용됩니다 ("오리처럼 보이고 오리처럼 꽥꽥댄다면, 그것은 오리다.") 특정한 형 대신에 인터페이스를 강조함으로써, 잘 설계된 코드는 다형적인 치환을 허락함으로써 유연성을 개선할 수 있습니다. 덕 타이핑은 `type()` 이나 `isinstance()` 을 사용한 검사를 피합니다. (하지만, 덕 타이핑이 추상 베이스 클래스 로 보완될 수 있음에 유의해야 합니다.) 대신에, `hasattr()` 검사나 *EAFP* 프로그래밍을 씁니다.

dunder
An informal short-hand for "double underscore", used when talking about a *special method*. For example, `__init__` is often pronounced "dunder init".

EAFP
허락보다는 용서를 구하기가 쉽다 (Easier to ask for forgiveness than permission). 이 흔히 볼 수 있는 파이썬 코딩 스타일은, 올바른 키나 어트리뷰트의 존재를 가정하고, 그 가정이 틀리면 예외를 잡습니다. 이 깔끔하고 빠른 스타일은 많은 `try` 와 `except` 문의 존재로 특징지어집니다. 이 테크닉은 C와 같은 다른 많은 언어에서 자주 사용되는 *LBYL* 스타일과 대비됩니다.

evaluate function
A function that can be called to evaluate a lazily evaluated attribute of an object, such as the value of type aliases created with the `type` statement.

expression (표현식)
어떤 값으로 구해질 수 있는 문법적인 조각. 다른 말로 표현하면, 표현식은 리터럴, 이름, 어트리뷰트 액세스, 연산자, 함수들과 같은 값을 돌려주는 표현 요소들을 쌓아 올린 것입니다. 다른 많은 언어와 대조적으로, 모든 언어 구성물들이 표현식인 것은 아닙니다. `while` 처럼, 표현식으로 사용할 수 없는 문장 들이 있습니다. 대입 또한 문장이고, 표현식이 아닙니다.

extension module (확장 모듈)
C 나 C++로 작성된 모듈인데, 파이썬의 C API를 사용해서 핵심이나 사용자 코드와 상호 작용합니다.

f-string (f-문자열)
f-strings
String literals prefixed with `f` or `F` are commonly called "f-strings" which is short for formatted string literals. See also **PEP 498**.

file object (파일 객체)
하부 자원에 대해 파일 지향적 API(`read()` 나 `write()` 같은 메서드들)를 드러내는 객체. 만들어진 방법에 따라, 파일 객체는 실제 디스크 상의 파일이나 다른 저장 장치나 통신 장치 (예를 들어, 표준 입출력, 인-메모리 버퍼, 소켓, 파이프, 등등)에 대한 액세스를 중계할 수 있습니다. 파일 객체는 파일류 객체 (*file-like objects*) 나 스트림 (*streams*) 이라고도 불립니다.

실제로는 세 부류의 파일 객체들이 있습니다. 날(raw) 바이너리 파일, 버퍼드(buffered) 바이너리 파일, 텍스트 파일. 이들의 인터페이스는 `io` 모듈에서 정의됩니다. 파일 객체를 만드는 규범적인 방법은 `open()` 함수를 쓰는 것입니다.

file-like object (파일류 객체)
파일 객체 의 비슷한 말.

filesystem encoding and error handler (파일시스템 인코딩과 에러 처리기)
Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

The filesystem encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise `UnicodeError`.

The `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()` functions can be used to get the filesystem encoding and error handler.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

로케일 인코딩 도 보세요.

finder (파인더)
임포트될 모듈을 위한 로더 를 찾으려고 시도하는 객체.

두 종류의 파인더가 있습니다: `sys.meta_path` 와 함께 사용하는 메타 경로 파인더 와 `sys.path_hooks` 과 함께 사용하는 경로 엔트리 파인더.

더 자세한 내용은 finders-and-loaders 와 `importlib`를 참조하십시오.

floor division (정수 나눗셈)
가장 가까운 정수로 내림하는 수학적 나눗셈. 정수 나눗셈 연산자는 `//` 다. 예를 들어, 표현식 `11 // 4` 의 값은 `2`가 되지만, 실수 나눗셈은 `2.75`를 돌려줍니다. `(-11) // 4` 가 `-2.75`를 내림 한 `-3`이 됨에 유의해야 합니다. **PEP 238**을 보세요.

free threading (자유 스레딩)
A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

free variable (자유 변수)
Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

function (함수)
호출자에게 어떤 값을 돌려주는 일련의 문장들. 없거나 그 이상의 인자 가 전달될 수 있는데, 바디의 실행에 사용될 수 있습니다. 매개변수 와 메서드 와 function 섹션도 보세요.

function annotation (함수 어노테이션)
함수 매개변수나 반환 값의 어노테이션.

함수 어노테이션은 일반적으로 형 힌트 로 사용됩니다: 예를 들어, 이 함수는 두 개의 `int` 인자를 받아들일 것으로 기대되고, 동시에 `int` 반환 값을 줄 것으로 기대됩니다:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

함수 어노테이션 문법은 function 절에서 설명합니다.

이 기능을 설명하는 변수 어노테이션 과 **PEP 484**를 참조하세요. 또한 어노테이션에 대한 모범 사례는 annotations-howto를 참조하세요.

__future__
A future statement, `from __future__ import <feature>`, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The `__future__` module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection (가비지 수거)
더 사용되지 않는 메모리를 반납하는 절차. 파이썬은 참조 횟수 추적과 참조 순환을 감지하고 끊을 수 있는 순환 가비지 수거기를 통해 가비지 수거를 수행합니다. 가비지 수거기는 `gc` 모듈을 사용해서 제어할 수 있습니다.

generator (제너레이터)
제너레이터 이터레이터 를 돌려주는 함수. 일반 함수처럼 보이는데, 일련의 값들을 만드는 `yield` 표현식을 포함한다는 점이 다릅니다. 이 값들은 for-루프로 사용하거나 *next()* 함수로 한 번에 하나씩 꺼낼 수 있습니다.

보통 제너레이터 함수를 가리키지만, 어떤 문맥에서는 제너레이터 이터레이터 를 가리킵니다. 의도하는 의미가 명확하지 않은 경우는, 완전한 용어를 써서 모호함을 없앱니다.

generator iterator (제너레이터 이터레이터)
제너레이터 함수가 만드는 객체.

각 `yield`는 일시적으로 처리를 중단하고, (지역 변수들과 대기 중인 try-문들을 포함하는) 실행 상태를 기억합니다. 제너레이터 이터레이터 가 재개되면, 떠난 곳으로 복귀합니다 (호출마다 새로 시작하는 함수와 대비됩니다).

generator expression (제너레이터 표현식)
이터레이터를 돌려주는 표현식. 루프 변수와 범위를 정의하는 `for` 절과 생략 가능한 `if` 절이 뒤에 붙는 일반 표현식 처럼 보입니다. 결합한 표현식은 둘러싼 함수를 위한 값들을 만들어냅니다:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

generic function (제네릭 함수)
같은 연산을 서로 다른 형들에 대해 구현한 여러 함수로 구성된 함수. 호출 때 어떤 구현이 사용될지는 디스패치 알고리즘에 의해 결정됩니다.

싱글 디스패치 용어집 항목과 *functools.singledispatch()* 데코레이터와 **PEP 443**도 보세요.

generic type (제네릭 형)
매개 변수화 할 수 있는 형; 일반적으로 *list* 나 *dict*와 같은 컨테이너 클래스. 형 힌트와 어노테이션에 사용됩니다.

For more details, see *generic alias types*, **PEP 483**, **PEP 484**, **PEP 585**, and the *typing* module.

GIL
전역 인터프리터 록 을 보세요.

global interpreter lock (전역 인터프리터 록)
한 번에 오직 하나의 스레드가 파이썬 바이트 코드 를 실행하도록 보장하기 위해 *CPython* 인터프리터가 사용하는 메커니즘. (*dict*와 같은 중요한 내장형들을 포함하는) 객체 모델이 묵시적으로 동시 액세스에 대해 안전하도록 만들어서 CPython 구현을 단순하게 만듭니다. 인터프리터 전체를 잠그는 것은 인터프리터를 다중스레드화하기 쉽게 만드는 대신, 다중 프로세서 기계가 제공하는 병렬성의 많은 부분을 희생합니다.

하지만, 어떤 확장 모듈들은, 표준이나 제삼자 모두, 압축이나 해싱 같은 계산 집약적인 작업을 수행할 때는 GIL을 반납하도록 설계되었습니다. 또한, I/O를 할 때는 항상 GIL을 반납합니다.

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

In prior versions of Python' s C API, a function might declare that it requires the GIL to be held in order to use it. This refers to having an *attached thread state*.

hash-based pyc (해시 기반 pyc)
유효성을 판별하기 위해 해당 소스 파일의 최종 수정 시간이 아닌 해시를 사용하는 바이트 코드 캐시 파일. pyc-invalidation을 참조하세요.

hashable (해시 가능)
객체가 일생 그 값이 변하지 않는 해시값을 갖고 (`__hash__()` 메서드가 필요합니다), 다른 객체와 비교될 수 있으면 (`__eq__()` 메서드가 필요합니다), 해시 가능하다고 합니다. 같다고 비교되는 해시 가능한 객체들의 해시값은 같아야 합니다.

해시 가능성은 객체를 딕셔너리의 키나 집합의 멤버로 사용할 수 있게 하는데, 이 자료 구조들이 내부적으로 해시값을 사용하기 때문입니다.

대부분 파이썬의 불변 내장 객체들은 해시 가능합니다; (리스트나 딕셔너리 같은) 가변 컨테이너들은 그렇지 않습니다; (튜플이나 frozenset 같은) 불변 컨테이너들은 그들의 요소들이 해시 가능할 때만 해시 가능합니다. 사용자 정의 클래스의 인스턴스 객체들은 기본적으로 해시 가능합니다. (자기 자신을 제외하고는) 모두 다르다고 비교되고, 해시값은 `id()` 로 부터 만들어집니다.

IDLE
파이썬을 위한 통합 개발 및 학습 환경 (Integrated Development and Learning Environment). *IDLE — 파이썬 편집기와 셸* 은 파이썬의 표준 배포판에 따라오는 기초적인 편집기와 인터프리터 환경입니다.

immortal (불멸)
Immortal objects are a CPython implementation detail introduced in **PEP 683**.

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

Immortal objects can be identified via `sys._is_immortal()`, or via `PyUnstable_IsImmortal()` in the C API.

immutable (불변)
고정된 값을 갖는 객체. 불변 객체는 숫자, 문자열, 튜플을 포함합니다. 이런 객체들은 변경될 수 없습니다. 새 값을 저장하려면 새 객체를 만들어야 합니다. 변하지 않는 해시값이 있어야 하는 곳에서 중요한 역할을 합니다, 예를 들어, 딕셔너리의 키.

import path (임포트 경로)
경로 기반 파인더 가 임포트 할 모듈을 찾기 위해 검색하는 장소들 (또는 경로 엔트리) 의 목록. 임포트 하는 동안, 이 장소들의 목록은 보통 `sys.path` 로부터 옵니다, 하지만 서브 패키지의 경우 부모 패키지의 `__path__` 어트리뷰트로부터 올 수도 있습니다.

importing (임포팅)
한 모듈의 파이썬 코드가 다른 모듈의 파이썬 코드에서 사용될 수 있도록 하는 절차.

importer (임포터)
모듈을 찾기도 하고 로드 하기도 하는 객체; 동시에 파인더 이자 로더 객체입니다.

interactive (대화형)
파이썬은 대화형 인터프리터를 갖고 있는데, 인터프리터 프롬프트에서 문장과 표현식을 입력할 수 있고, 즉각 실행된 결과를 볼 수 있다는 뜻입니다. 인자 없이 단지 `python`을 실행하세요 (컴퓨터의 주메뉴에서 선택하는 것도 가능할 수 있습니다). 새 아이디어를 검사하거나 모듈과 패키지를 들여다 보는 매우 강력한 방법입니다 (`help(x)` 를 기억하세요). 대화형 모드에 대한 자세한 내용은 tut-interac 를 보세요.

interpreted (인터프리티드)
바이트 코드 컴파일러의 존재 때문에 그 구분이 흐릿해지기는 하지만, 파이썬은 컴파일 언어가 아니라 인터프리터 언어입니다. 이것은 명시적으로 실행 파일을 만들지 않고도, 소스 파일을 직접 실행할 수 있다는 뜻입니다. 그 프로그램이 좀 더 천천히 실행되기는 하지만, 인터프리터 언어는 보통 컴파일 언어보다 짧은 개발/디버깅 주기를 갖습니다. 대화형 도 보세요.

interpreter shutdown (인터프리터 종료)
종료하라는 요청을 받을 때, 파이썬 인터프리터는 특별한 시기에 진입하는데, 모듈이나 여러 가지 중요한 내부 구조들과 같은 모든 할당된 자원들을 단계적으로 반납합니다. 또한, 가비지 수거기 를 여러 번 호출합니다. 사용자 정의 파괴자나 weakref 콜백에 있는 코드들의 실행을 시작시킬 수 있습니다. 종료 시기 동안 실행되는 코드는 다양한 예외들을 만날 수 있는데, 그것이 의존하는 자원들이 더 기능하지 않을 수 있기 때문입니다 (흔한 예는 라이브러리 모듈이나 경고 장치들입니다).

인터프리터 종료의 주된 원인은 실행되는 `__main__` 모듈이나 스크립트가 실행을 끝내는 것입니다.

iterable (이터러블)
멤버들을 한 번에 하나씩 돌려줄 수 있는 객체. 이터러블의 예로는 모든 (`list`, `str`, `tuple` 같은) 시퀀스 형들, `dict` 같은 몇몇 비 시퀀스 형들, 파일 객체들, `__iter__()` 나 시퀀스 개념을 구현하는 `__getitem__()` 메서드를 써서 정의한 모든 클래스의 객체들이 있습니다.

이터러블은 `for` 루프에 사용될 수 있고, 시퀀스를 필요로 하는 다른 많은 곳 (`zip()`, `map()`, …) 에 사용될 수 있습니다. 이터러블 객체가 내장 함수 `iter()` 에 인자로 전달되면, 그 객체의 이터레이터를 돌려줍니다. 이 이터레이터는 값들의 집합을 한 번 거치는 동안 유효합니다. 이터러블을 사용할 때, 보통은 `iter()` 를 호출하거나, 이터레이터 객체를 직접 다룰 필요는 없습니다. `for` 문은 이것들을

여러분을 대신해서 자동으로 해주는데, 루프를 도는 동안 이터레이터를 잡아둘 이름 없는 변수를 만듭니다. 이터레이터, 시퀀스, 제너레이터 도 보세요.

iterator (이터레이터)
데이터의 스트림을 표현하는 객체. 이터레이터의 *__next__()* 메서드를 반복적으로 호출하면 (또는 내장 함수 *next()* 로 전달하면) 스트림에 있는 항목들을 차례대로 돌려줍니다. 더 이상의 데이터가 없을 때는 대신 *StopIteration* 예외를 일으킵니다. 이 지점에서, 이터레이터 객체는 소진되고, 이후의 모든 `__next__()` 메서드 호출은 *StopIteration* 예외를 다시 일으키기만 합니다. 이터레이터는 이터레이터 객체 자신을 돌려주는 *__iter__()* 메서드를 가질 것이 요구되기 때문에, 이터레이터는 이터러블이기도 하고 다른 이터러블들을 받아들이는 대부분의 곳에서 사용될 수 있습니다. 중요한 예외는 여러 번의 이터레이션을 시도하는 코드입니다. (*list* 같은) 컨테이너 객체는 *iter()* 함수로 전달하거나 `for` 루프에 사용할 때마다 새 이터레이터를 만듭니다. 이런 것을 이터레이터에 대해서 수행하려고 하면, 지난 이터레이션에 사용된 이미 소진된 이터레이터를 돌려줘서, 빈 컨테이너처럼 보이게 만듭니다.

이터레이터 형 에 더 자세한 내용이 있습니다.

CPython 구현 상세: CPython does not consistently apply the requirement that an iterator define *__iter__()*. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

key function (키 함수)
키 함수 또는 콜레이션(collation) 함수는 정렬(sorting)이나 배열(ordering)에 사용되는 값을 돌려주는 콜러블입니다. 예를 들어, *locale.strxfrm()* 은 로케일 특정 방식을 따르는 정렬 키를 만드는 데 사용됩니다.

파이썬의 많은 도구가 요소들이 어떻게 순서 지어지고 묶이는지를 제어하기 위해 키 함수를 받아들입니다. 이런 것들에는 *min()*, *max()*, *sorted()*, *list.sort()*, *heapq.merge()*, *heapq.nsmallest()*, *heapq.nlargest()*, *itertools.groupby()* 이 있습니다.

키 함수를 만드는 데는 여러 방법이 있습니다. 예를 들어, *str.lower()* 메서드는 케이스 구분 없는 정렬을 위한 키 함수로 사용될 수 있습니다. 대안적으로, 키 함수는 `lambda` 표현식으로 만들 수도 있는데, 이런 식입니다: `lambda r: (r[0], r[2])`. 또한, *operator.attrgetter()*, *operator.itemgetter()*, *operator.methodcaller()* 가 세 개의 키 함수 생성자입니다. 키 함수를 만들고 사용하는 법에 대한 예로 Sorting HOW TO 를 보세요.

keyword argument (키워드 인자)
인자 를 보세요.

lambda (람다)
호출될 때 값이 구해지는 하나의 표현식 으로 구성된 이름 없는 인라인 함수. 람다 함수를 만드는 문법은 `lambda [parameters]: expression` 입니다.

LBYL
뛰기 전에 보라 (Look before you leap). 이 코딩 스타일은 호출이나 조회를 하기 전에 명시적으로 사전 조건들을 검사합니다. 이 스타일은 *EAFP* 접근법과 대비되고, 많은 `if` 문의 존재로 특징지어집니다.

다중 스레드 환경에서, LBYL 접근법은 "보기" 와 "뛰기" 간에 경쟁 조건을 만들게 될 위험이 있습니다. 예를 들어, 코드 `if key in mapping: return mapping[key]` 는 검사 후에, 하지만 조회 전에, 다른 스레드가 *key*를 *mapping*에서 제거하면 실패할 수 있습니다. 이런 이슈는 록이나 EAFP 접근법을 사용함으로써 해결될 수 있습니다.

lexical analyzer (어휘 분석기)
Formal name for the *tokenizer*; see *token*.

list (리스트)
내장 파이썬 시퀀스. 그 이름에도 불구하고, 원소에 대한 액세스가 $O(1)$ 이기 때문에, 연결 리스트 (linked list) 보다는 다른 언어의 배열과 유사합니다.

list comprehension (리스트 컴프리헨션)
시퀀스의 요소들 전부 또는 일부를 처리하고 그 결과를 리스트로 돌려주는 간결한 방법. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 는 0에서 255 사이에 있는 짝수들의 16진수 (0x..) 들을 포함하는 문자열의 리스트를 만듭니다. `if` 절은 생략할 수 있습니다. 생략하면, `range(256)` 에 있는 모든 요소가 처리됩니다.

loader (로더)
An object that loads a module. It must define the `exec_module()` and `create_module()` methods to implement the `Loader` interface. A loader is typically returned by a *finder*. See also:

- finders-and-loaders
- `importlib.abc.Loader`
- **PEP 302**

locale encoding (로케일 인코딩)
On Unix, it is the encoding of the LC_CTYPE locale. It can be set with `locale.setlocale(locale.LC_CTYPE, new_locale)`.

On Windows, it is the ANSI code page (ex: `"cp1252"`).

On Android and VxWorks, Python uses `"utf-8"` as the locale encoding.

`locale.getencoding()` can be used to get the locale encoding.

See also the *filesystem encoding and error handler*.

magic method (매직 메서드)
특수 메서드 의 비공식적인 비슷한 말.

mapping (매핑)
임의의 키 조회를 지원하고 `collections.abc.Mapping` 이나 `collections.abc.MutableMapping` 추상 베이스 클래스 에 지정된 메서드들을 구현하는 컨테이너 객체. 예로는 `dict`, `collections.defaultdict`, `collections.OrderedDict`, `collections.Counter` 를 들 수 있습니다.

meta path finder (메타 경로 파인더)
`sys.meta_path` 의 검색이 돌려주는 파인더. 메타 경로 파인더는 경로 엔트리 파인더 와 관련되어 있기는 하지만 다릅니다.

메타 경로 파인더가 구현하는 메서드들에 대해서는 `importlib.abc.MetaPathFinder` 를 보면 됩니다.

metaclass (메타 클래스)
클래스의 클래스. 클래스 정의는 클래스 이름, 클래스 딕셔너리, 베이스 클래스들의 목록을 만듭니다. 메타 클래스는 이 세 인자를 받아서 클래스를 만드는 책임을 집니다. 대부분의 객체 지향형 프로그래밍 언어들은 기본 구현을 제공합니다. 파이썬을 특별하게 만드는 것은 커스텀 메타 클래스를 만들 수 있다는 것입니다. 대부분 사용자에게는 이 도구가 전혀 필요 없지만, 필요가 생길 때, 메타 클래스는 강력하고 우아한 해법을 제공합니다. 어트리뷰트 액세스의 로깅(logging), 스레드 안전성의 추가, 객체 생성 추적, 싱글톤 구현과 많은 다른 작업에 사용됐습니다.

metaclasses 에서 더 자세한 내용을 찾을 수 있습니다.

method (메서드)
클래스 바디 안에서 정의되는 함수. 그 클래스의 인스턴스의 어트리뷰트로서 호출되면, 그 메서드는 첫 번째 인자 (보통 `self` 라고 불린다) 로 인스턴스 객체를 받습니다. 함수 와 중첩된 스코프 를 보세요.

method resolution order (메서드 결정 순서)
메서드 결정 순서는 조회하는 동안 멤버를 검색하는 베이스 클래스들의 순서입니다. 2.3 릴리스부터 파이썬 인터프리터에 사용된 알고리즘의 상세한 내용은 python_2.3_mro를 보세요.

module (모듈)
파이썬 코드의 조직화 단위를 담당하는 객체. 모듈은 임의의 파이썬 객체들을 담는 이름 공간을 갖습니다. 모듈은 임포팅 절차에 의해 파이썬으로 로드됩니다.

패키지 도 보세요.

module spec (모듈 스펙)
모듈을 로드하는데 사용되는 임포트 관련 정보들을 담고 있는 이름 공간. `importlib.machinery.ModuleSpec` 의 인스턴스.

module-specs 도 보세요.

MRO
메서드 결정 순서 를 보세요.

mutable (가변)
가변 객체는 값이 변할 수 있지만 `id()` 는 일정하게 유지합니다. 불변 도 보세요.

named tuple (네임드 튜플)
"named tuple(네임드 튜플)" 이라는 용어는 튜플에서 상속하고 이름 붙은 어트리뷰트를 사용하여 인덱스 할 수 있는 요소에 액세스 할 수 있는 모든 형이나 클래스에 적용됩니다. 형이나 클래스에는 다른 기능도 있을 수 있습니다.

`time.localtime()`과 `os.stat()`가 반환한 값을 포함하여, 여러 내장형이 네임드 튜플입니다. 또 다른 예는 `sys.float_info`입니다:

```
>>> sys.float_info[1]                    # indexed access
1024
>>> sys.float_info.max_exp               # named field access
1024
>>> isinstance(sys.float_info, tuple)    # kind of tuple
True
```

일부 네임드 튜플은 내장형(위의 예)입니다. 또는, `tuple`에서 상속하고 이름 붙은 필드를 정의하는 일반 클래스 정의로 네임드 튜플을 만들 수 있습니다. 이러한 클래스는 직접 작성하거나, `typing.NamedTuple`를 계승하거나 팩토리 함수 `collections.namedtuple()`로 만들 수 있습니다. 후자의 기법은 직접 작성하거나 내장 네임드 튜플에서는 찾을 수 없는 몇 가지 추가 메서드를 추가하기도 합니다.

namespace (이름 공간)
변수가 저장되는 장소. 이름 공간은 딕셔너리로 구현됩니다. 객체에 중첩된 이름 공간(메서드 에서) 뿐만 아니라 지역, 전역, 내장 이름 공간이 있습니다. 이름 공간은 이름 충돌을 방지해서 모듈성을 지원합니다. 예를 들어, 함수 `builtins.open`과 `os.open()` 은 그들의 이름 공간에 의해 구별됩니다. 또한, 이름 공간은 어떤 모듈이 함수를 구현하는지를 분명하게 만들어서 가독성과 유지 보수성에 도움을 줍니다. 예를 들어, `random.seed()` 또는 `itertools.islice()` 라고 쓰면 그 함수들이 각각 `random`과 `itertools` 모듈에 의해 구현되었음이 명확해집니다.

namespace package (이름 공간 패키지)
오직 서브 패키지들의 컨테이너로만 기능하는 패키지. 이름 공간 패키지는 물리적인 실체가 없을 수도 있고, 특히 `__init__.py` 파일이 없으므로 정규 패키지 와는 다릅니다.

Namespace packages allow several individually installable packages to have a common parent package. Otherwise, it is recommended to use a *regular package*.

For more information, see **PEP 420** and reference-namespace-package.

모듈 도 보세요.

nested scope (중첩된 스코프)
둘러싼 정의에서 변수를 참조하는 능력. 예를 들어, 다른 함수 내부에서 정의된 함수는 바깥 함수에 있는 변수들을 참조할 수 있습니다. 중첩된 스코프는 기본적으로는 참조만 가능할 뿐, 대입은 되지 않는다는 것에 주의해야 합니다. 지역 변수들은 가장 내부의 스코프에서 읽고 씁니다. 마찬가지로, 전역 변수들은 전역 이름 공간에서 읽고 씁니다. `nonlocal` 은 바깥 스코프에 쓰는 것을 허락합니다.

new-style class (뉴스타일 클래스)
지금은 모든 클래스 객체에 사용되고 있는 클래스 버전의 예전 이름. 초기의 파이썬 버전에서는, 오직 뉴스타일 클래스만 `__slots__`, 디스크립터, 프라퍼티, `__getattribute__()`, 클래스 메서드, 스태틱 메서드와 같은 파이썬의 새롭고 다양한 기능들을 사용할 수 있었습니다.

object (객체)
상태 (어트리뷰트나 값) 를 갖고 동작 (메서드) 이 정의된 모든 데이터. 또한, 모든 뉴스타일 클래스의 최종적인 베이스 클래스입니다.

optimized scope (최적화된 스코프)
A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators,

coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package (패키지)
서브 모듈들이나, 재귀적으로 서브 패키지들을 포함할 수 있는 파이썬 모듈. 기술적으로, 패키지는 `__path__` 어트리뷰트가 있는 파이썬 모듈입니다.

정규 패키지 와 이름 공간 패키지 도 보세요.

parameter (매개변수)
함수 (또는 메서드) 정의에서 함수가 받을 수 있는 인자 (또는 어떤 경우 인자들) 를 지정하는 이름 붙은 엔티티. 다섯 종류의 매개변수가 있습니다:

- 위치-키워드 (*positional-or-keyword*): 위치 인자 나 키워드 인자 로 전달될 수 있는 인자를 지정합니다. 이것이 기본 형태의 매개변수입니다, 예를 들어 다음에서 *foo* 와 *bar*:

```
def func(foo, bar=None): ...
```

- 위치-전용 (*positional-only*): 위치로만 제공될 수 있는 인자를 지정합니다. 위치 전용 매개변수는 함수 정의의 매개변수 목록에 / 문자를 포함하고 그 뒤에 정의할 수 있습니다, 예를 들어 다음에서 *posonly1* 과 *posonly2*:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- 키워드-전용 (*keyword-only*): 키워드로만 제공될 수 있는 인자를 지정합니다. 키워드-전용 매개변수는 함수 정의의 매개변수 목록에서 앞에 하나의 가변-위치 매개변수나 * 를 그대로 포함해서 정의할 수 있습니다. 예를 들어, 다음에서 *kw_only1* 와 *kw_only2*:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- 가변-위치 (*var-positional*): (다른 매개변수들에 의해서 이미 받아들여진 위치 인자들에 더해) 제공될 수 있는 위치 인자들의 임의의 시퀀스를 지정합니다. 이런 매개변수는 매개변수 이름에 * 를 앞에 붙여서 정의될 수 있습니다, 예를 들어 다음에서 *args*:

```
def func(*args, **kwargs): ...
```

- 가변-키워드 (*var-keyword*): (다른 매개변수들에 의해서 이미 받아들여진 키워드 인자들에 더해) 제공될 수 있는 임의의 개수 키워드 인자들을 지정합니다. 이런 매개변수는 매개변수 이름에 ** 를 앞에 붙여서 정의될 수 있습니다, 예를 들어 위의 예에서 *kwargs*.

매개변수는 선택적 인자들을 위한 기본값뿐만 아니라 선택적이거나 필수 인자들을 지정할 수 있습니다.

인자 용어집 항목, 인자와 매개변수의 차이에 나오는 FAQ 질문, `inspect.Parameter` 클래스, function 절, **PEP 362**도 보세요.

path entry (경로 엔트리)
경로 기반 파인더 가 임포트 할 모듈들을 찾기 위해 참고하는 임포트 경로 상의 하나의 장소.

path entry finder (경로 엔트리 파인더)
`sys.path_hooks` 에 있는 콜러블 (즉, 경로 엔트리 훅) 이 돌려주는 파인더 인데, 주어진 경로 엔트리 로 모듈을 찾는 방법을 알고 있습니다.

경로 엔트리 파인더들이 구현하는 메서드들은 `importlib.abc.PathEntryFinder` 에 나옵니다.

path entry hook (경로 엔트리 훅)
`sys.path_hooks` 리스트에 있는 콜러블인데, 특정 경로 엔트리 에서 모듈을 찾는 법을 알고 있다면 경로 엔트리 파인더 를 돌려줍니다.

path based finder (경로 기반 파인더)
기본 메타 경로 파인더들 중 하나인데, 임포트 경로 에서 모듈을 찾습니다.

path-like object (경로류 객체)
파일 시스템 경로를 나타내는 객체. 경로류 객체는 경로를 나타내는 `str` 나 `bytes` 객체이거

나 `os.PathLike` 프로토콜을 구현하는 객체입니다. `os.PathLike` 프로토콜을 지원하는 객체는 `os.fspath()` 함수를 호출해서 `str` 나 `bytes` 파일 시스템 경로로 변환될 수 있습니다; 대신 `os.fsdecode()` 와 `os.fsencode()` 는 각각 `str` 나 `bytes` 결과를 보장하는데 사용될 수 있습니다. **PEP 519**로 도입되었습니다.

PEP

파이썬 개선 제안. PEP는 파이썬 커뮤니티에 정보를 제공하거나 파이썬 또는 그 프로세스 또는 환경에 대한 새로운 기능을 설명하는 설계 문서입니다. PEP는 제안된 기능에 대한 간결한 기술 사양 및 근거를 제공해야 합니다.

PEP는 주요 새로운 기능을 제안하고 문제에 대한 커뮤니티 입력을 수집하며 파이썬에 들어간 설계 결정을 문서로 만들기 위한 기본 메커니즘입니다. PEP 작성자는 커뮤니티 내에서 합의를 구축하고 반대 의견을 문서화 할 책임이 있습니다.

PEP 1 참조하세요.

portion (포션)

PEP 420 에서 정의한 것처럼, 이름 공간 패키지에 이바지하는 하나의 디렉터리에 들어있는 파일들의 집합 (zip 파일에 저장되는 것도 가능합니다).

positional argument (위치 인자)

인자 를 보세요.

provisional API (잠정 API)

잠정 API는 표준 라이브러리의 과거 호환성 보장으로부터 신중히 제외된 것입니다. 인터페이스의 큰 변화가 예상되지는 않지만, 잠정적이라고 표시되는 한, 코어 개발자들이 필요하다고 생각한다면 과거 호환성이 유지되지 않는 변경이 일어날 수 있습니다. 그런 변경은 불필요한 방식으로 일어나지는 않을 것입니다 — API를 포함하기 전에 놓친 중대하고 근본적인 결함이 발견된 경우에만 일어날 것입니다.

잠정 API에서조차도, 과거 호환성이 유지되지 않는 변경은 "최후의 수단" 으로 여겨집니다 - 모든 식별된 문제들에 대해 과거 호환성을 유지하는 해법을 찾으려는 모든 시도가 선행됩니다.

이 절차는 표준 라이브러리가 오랜 시간 동안 잘못된 설계 오류에 발목 잡히지 않고 발전할 수 있도록 만듭니다. 더 자세한 내용은 **PEP 411**을 보면 됩니다.

provisional package (잠정 패키지)

잠정 *API* 를 보세요.

Python 3000 (파이썬 3000)

파이썬 3.x 배포 라인의 별명 (버전 3의 배포가 먼 미래의 이야기던 시절에 만들어진 이름이다.) 이것을 "Py3k" 로 줄여 쓰기도 합니다.

Pythonic (파이썬다운)

다른 언어들에서 일반적인 개념들을 사용해서 코드를 구현하는 대신, 파이썬 언어에서 가장 자주 사용되는 이디엄들을 가까이 따르는 아이디어나 코드 조각. 예를 들어, 파이썬에서 자주 쓰는 이디엄은 `for` 문을 사용해서 이터러블의 모든 요소로 루핑하는 것입니다. 다른 많은 언어에는 이런 종류의 구성물이 없으므로, 파이썬에 익숙하지 않은 사람들은 대신에 숫자 카운터를 사용하기도 합니다:

```
for i in range(len(food)):
    print(food[i])
```

더 깔끔한, 파이썬다운 방법은 이렇습니다:

```
for piece in food:
    print(piece)
```

qualified name (정규화된 이름)

모듈의 전역 스코프에서 모듈에 정의된 클래스, 함수, 메서드에 이르는 "경로" 를 보여주는 점으로 구분된 이름. **PEP 3155** 에서 정의됩니다. 최상위 함수와 클래스의 경우에, 정규화된 이름은 객체의 이름과 같습니다:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

모듈을 가리키는데 사용될 때, 완전히 정규화된 이름(*fully qualified name*)은 모든 부모 패키지들을 포함해서 모듈로 가는 점으로 분리된 이름을 의미합니다, 예를 들어, `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count (참조 횟수)
객체에 대한 참조의 개수. 객체의 참조 횟수가 0으로 떨어지면, 메모리가 반납됩니다. 일부 객체는 불멸이며 참조 횟수가 수정되지 않아서, 객체가 할당 해제되지 않습니다. 참조 횟수 추적은 일반적으로 파이썬 코드에 노출되지는 않지만, *CPython* 구현의 핵심 요소입니다. 프로그래머는 특정 객체의 참조 횟수를 돌려주는 `sys.getrefcount()` 함수를 호출할 수 있습니다.

In *CPython*, reference counts are not considered to be stable or well-defined values; the number of references to an object, and how that number is affected by Python code, may be different between versions.

regular package (정규 패키지)
`__init__.py` 파일을 포함하는 디렉터리와 같은 전통적인 패키지.

이름 공간 패키지 도 보세요.

REPL
An acronym for the "read–eval–print loop", another name for the *interactive* interpreter shell.

__slots__
클래스 내부의 선언인데, 인스턴스 어트리뷰트들을 위한 공간을 미리 선언하고 인스턴스 딕셔너리를 제거함으로써 메모리를 절감하는 효과를 줍니다. 인기 있기는 하지만, 이 테크닉은 올바르게 사용하기가 좀 까다로운 편이라서, 메모리에 민감한 응용 프로그램에서 많은 수의 인스턴스가 있는 특별한 경우로 한정하는 것이 좋습니다.

sequence (시퀀스)
`__getitem__()` 특수 메서드를 통해 정수 인덱스를 사용한 빠른 요소 액세스를 지원하고, 시퀀스의 길이를 돌려주는 `__len__()` 메서드를 정의하는 이터러블. 몇몇 내장 시퀀스들을 나열해보면, `list`, `str`, `tuple`, `bytes` 가 있습니다. `dict` 또한 `__getitem__()` 과 `__len__()` 을 지원하지만, 조회에 정수 대신 임의의 해시 가능 키를 사용하기 때문에 시퀀스가 아니라 매핑으로 취급된다는 것에 주의해야 합니다.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see *Common Sequence Operations*.

set comprehension (집합 컴프리헨션)
이터러블에 있는 요소 전체나 일부를 처리하고 결과를 담은 집합을 반환하는 간결한 방법. `results = {c for c in 'abracadabra' if c not in 'abc'}`는 문자열의 집합 `{'r', 'd'}`를 생성합니다. comprehensions을 참조하십시오.

single dispatch (싱글 디스패치)
구현이 하나의 인자의 형에 기초해서 결정되는 제네릭 함수 디스패치의 한 형태.

slice (슬라이스)
보통 시퀀스 의 일부를 포함하는 객체. 슬라이스는 서브 스크립트 표기법을 사용해서 만듭니다. `variable_name[1:3:5]` 처럼, `[]` 안에서 여러 개의 숫자를 콜론으로 분리합니다. 대괄호 (서브 스크립트) 표기법은 내부적으로 *`slice`* 객체를 사용합니다.

soft deprecated (약하게 폐지된)
A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See PEP 387: Soft Deprecation.

special method (특수 메서드)
파이썬이 형에 어떤 연산을, 덧셈 같은, 실행할 때 묵시적으로 호출되는 메서드. 이런 메서드는 두 개의 밑줄로 시작하고 끝나는 이름을 갖고 있습니다. 특수 메서드는 specialnames 에 문서로 만들어져 있습니다.

standard library
The collection of *packages*, *modules* and *extension modules* distributed as a part of the official Python interpreter package. The exact membership of the collection may vary based on platform, available system libraries, or other criteria. Documentation can be found at 파이썬 표준 라이브러리.

See also *`sys.stdlib_module_names`* for a list of all possible standard library module names.

statement (문장)
문장은 스위트 (코드의 "블록(block)") 를 구성하는 부분입니다. 문장은 표현식 이거나 키워드를 사용하는 여러 가지 구조물 중의 하나입니다. 가령 `if`, `while`, `for`.

static type checker (정적 형 검사기)
An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the *`typing`* module.

stdlib
An abbreviation of *standard library*.

strong reference (강한 참조)
In Python' s C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

The `Py_NewRef()` function can be used to create a strong reference to an object. Usually, the `Py_DECREF()` function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

빌린 참조 도 보세요.

t-string
t-strings
String literals prefixed with `t` or `T` are commonly called "t-strings" which is short for template string literals.

text encoding (텍스트 인코딩)
A string in Python is a sequence of Unicode code points (in range `U+0000-U+10FFFF`). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as "encoding", and recreating the string from the sequence of bytes is known as "decoding".

There are a variety of different text serialization *codecs*, which are collectively referred to as "text encodings".

text file (텍스트 파일)
`str` 객체를 읽고 쓸 수 있는 파일 객체. 종종, 텍스트 파일은 실제로는 바이트 지향 데이터스트림을 액세스하고 텍스트 인코딩 을 자동 처리합니다. 텍스트 파일의 예로는 텍스트 모드 (`'r'` 또는 `'w'`) 로 열린 파일, *`sys.stdin`*, *`sys.stdout`*, *`io.StringIO`* 의 인스턴스를 들 수 있습니다.

바이트열류 객체 를 읽고 쓸 수 있는 파일 객체에 대해서는 바이너리 파일 도 참조하세요.

thread state

The information used by the *CPython* runtime to run in an OS thread. For example, this includes the current exception, if any, and the state of the bytecode interpreter.

Each thread state is bound to a single OS thread, but threads may have many thread states available. At most, one of them may be *attached* at once.

An *attached thread state* is required to call most of Python' s C API, unless a function explicitly documents otherwise. The bytecode interpreter only runs under an attached thread state.

Each thread state belongs to a single interpreter, but each interpreter may have many thread states, including multiple for the same OS thread. Thread states from multiple interpreters may be bound to the same thread, but only one can be *attached* in that thread at any given moment.

See Thread State and the Global Interpreter Lock for more information.

token (토큰)

A small unit of source code, generated by the lexical analyzer (also called the *tokenizer*). Names, numbers, strings, operators, newlines and similar are represented by tokens.

The `tokenize` module exposes Python' s lexical analyzer. The `token` module contains information on the various types of tokens.

triple-quoted string (삼중 따옴표 된 문자열)

따옴표 (") 나 작은따옴표 (') 세 개로 둘러싸인 문자열. 그냥 따옴표 하나로 둘러싸인 문자열에 없는 기능을 제공하지는 않지만, 여러 가지 이유에서 쓸모가 있습니다. 이스케이프 되지 않은 작은따옴표나 큰따옴표를 문자열 안에 포함할 수 있도록 하고, 연결 문자를 쓰지 않고도 여러 줄에 걸칠 수 있는데, 독스트링을 쓸 때 특히 쓸모 있습니다.

type (형)

파이썬 객체의 형은 그것이 어떤 종류의 객체인지를 결정합니다; 모든 객체는 형이 있습니다. 객체의 형은 `__class__` 어트리뷰트로 액세스할 수 있거나 `type(obj)` 로 얻을 수 있습니다.

type alias (형 에일리어스)

형을 식별자에 대입하여 만들어지는 형의 동의어.

형 에일리어스는 형 힌트를 단순화하는 데 유용합니다. 예를 들면:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

는 다음과 같이 더 읽기 쉽게 만들 수 있습니다:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

이 기능을 설명하는 `typing`과 **PEP 484**를 참조하세요.

type hint (형 힌트)

변수, 클래스 어트리뷰트 및 함수 매개변수 나 반환 값의 기대되는 형을 지정하는 어노테이션.

형 힌트는 선택 사항이고 파이썬에서 강제되지는 않지만, 정적 형 검사기에 유용합니다. 또한 IDE의 코드 완성 및 리팩토링을 돕습니다.

지역 변수를 제외하고, 전역 변수, 클래스 어트리뷰트 및 함수의 형 힌트는 `typing.get_type_hints()`를 사용하여 액세스할 수 있습니다.

이 기능을 설명하는 `typing`과 **PEP 484**를 참조하세요.

universal newlines (유니버설 줄 넘김)

다음과 같은 것들을 모두 줄의 끝으로 인식하는, 텍스트 스트림을 해석하는 태도: 유닉스 개행 문자 관례 `'\n'`, 윈도우즈 관례 `'\r\n'`, 예전의 매킨토시 관례 `'\r'`. 추가적인 사용에 관해서는 `bytes.splitlines()` 뿐만 아니라 **PEP 278** 와 **PEP 3116** 도 보세요.

variable annotation (변수 어노테이션)
변수 또는 클래스 어트리뷰트의 어노테이션.

변수 또는 클래스 어트리뷰트에 어노테이션을 달 때 대입은 선택 사항입니다:

```
class C:
    field: 'annotation'
```

변수 어노테이션은 일반적으로 형 힌트로 사용됩니다: 예를 들어, 이 변수는 *int* 값을 가질 것으로 기대됩니다:

```
count: int = 0
```

변수 어노테이션 문법은 섹션 annassign 에서 설명합니다.

이 기능을 설명하는 함수 어노테이션, **PEP 484** 및 **PEP 526**을 참조하세요. 또한 어노테이션 작업에 대한 모범 사례는 annotations-howto를 참조하세요.

virtual environment (가상 환경)
파이썬 사용자와 응용 프로그램이, 같은 시스템에서 실행되는 다른 파이썬 응용 프로그램들의 동작에 영향을 주지 않으면서, 파이썬 배포 패키지들을 설치하거나 업그레이드하는 것을 가능하게 하는, 협력적으로 격리된 실행 환경.

venv 도 보세요.

virtual machine (가상 기계)
소프트웨어만으로 정의된 컴퓨터. 파이썬의 가상 기계는 바이트 코드 컴파일러가 출력하는 바이트 코드를 실행합니다.

walrus operator
A light-hearted way to refer to the assignment expression operator `:=` because it looks a bit like a walrus if you turn your head.

Zen of Python (파이썬 젠)
파이썬 디자인 원리와 철학들의 목록인데, 언어를 이해하고 사용하는 데 도움이 됩니다. 이 목록은 대화형 프롬프트에서 "`import this`" 를 입력하면 보입니다.

APPENDIX B

이 설명서에 관하여

파이썬 설명서는 reStructuredText 소스에서 만들어진 것으로, 원래 파이썬을 위해 제작되었고 이제는 독립 프로젝트로 유지 관리되는 설명서 생성기인 Sphinx 를 사용했습니다.

설명서와 이를 위한 툴체인 개발은 파이썬 자체와 마찬가지로 전적으로 자원봉사자의 노력입니다. 기여하고 싶다면, 참여 방법에 대한 정보는 reporting-bugs 페이지를 참고하십시오. 새로운 자원봉사자는 언제나 환영합니다!

다음 분들에게 많은 감사를 드립니다:

- Fred L. Drake, Jr., 원래 파이썬 설명서 도구 집합의 작성자이자 많은 콘텐츠의 저자;
- reStructuredText와 Docutils 스위트를 만드는 Docutils 프로젝트.
- Fredrik Lundh, 그의 대안 파이썬 참조(Alternative Python Reference) 프로젝트에서 Sphinx가 많은 아이디어를 얻었습니다.

B.1 파이썬 설명서의 공헌자들

많은 사람이 파이썬 언어, 파이썬 표준 라이브러리 및 파이썬 설명서에 기여했습니다. 기여자의 부분적인 목록은 파이썬 소스 배포판의 Misc/ACKS 를 참조하십시오.

파이썬이 이런 멋진 설명서를 갖게 된 것은 파이썬 커뮤니티의 입력과 기여 때문입니다 – 감사합니다!

APPENDIX C

역사와 라이센스

C.1 소프트웨어의 역사

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl) in the Netherlands as a successor of a language called ABC. Guido remains Python' s principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations, which became Zope Corporation. In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation was a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

배포판	파생된 곳	해	소유자	GPL-compatible? (1)
0.9.0 ~ 1.2	n/a	1991-1995	CWI	yes
1.3 ~ 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	yes (2)
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 이상	2.1.1	2001-현재	PSF	yes

참고

(1) GPL-compatible doesn' t mean that we' re distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-

compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don' t.

(2) According to Richard Stallman, 1.6.1 is not GPL-compatible, because its license has a choice of law clause. According to CNRI, however, Stallman' s lawyer has told CNRI' s lawyer that 1.6.1 is "not incompatible" with the GPL.

Guido의 지도하에 이 배포를 가능하게 만든 많은 외부 자원봉사자들에게 감사드립니다.

C.2 파이썬에 액세스하거나 사용하기 위한 이용 약관

Python software and documentation are licensed under the Python Software Foundation License Version 2.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Version 2 and the *Zero-Clause BSD license*.

파이썬에 통합된 일부 소프트웨어에는 다른 라이센스가 적용됩니다. 라이센스는 해당 라이센스에 해당하는 코드와 함께 나열됩니다. 이러한 라이센스의 불완전한 목록은 포함된 소프트웨어에 대한 라이센스 및 승인를 참조하십시오.

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001 Python Software Foundation; All Rights
   Reserved" are retained in Python alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python.

4. PSF is making Python available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This License
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   Agreement does not grant permission to use PSF trademarks or trade name in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.

8. By copying, installing or otherwise using Python, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 포함된 소프트웨어에 대한 라이센스 및 승인

이 섹션은 파이썬 배포판에 포함된 제삼자 소프트웨어에 대한 불완전하지만 늘어나고 있는 라이센스와 승인의 목록입니다.

C.3.1 메르센 트위스터

The `_random` C extension underlying the *random* module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 소켓

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, https://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 비동기 소켓 서비스

The `test.support.asynchat` and `test.support.asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                         All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 쿠키 관리

http.cookies 모듈은 다음과 같은 주의 사항을 포함합니다:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 실행 추적

trace 모듈은 다음과 같은 주의 사항을 포함합니다:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode 및 UUdecode 함수

The `uu` codec contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                      All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 원격 프로시저 호출

xmlrpc.client 모듈은 다음과 같은 주의 사항을 포함합니다:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

The `test.test_epoll` module contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

`select` 모듈은 kqueue 인터페이스에 대해 다음과 같은 주의 사항을 포함합니다:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

파일 `Python/pyhash.c` 에는 Dan Bernstein의 SipHash24 알고리즘의 Marek Majkowski의 구현이 포함되어 있습니다. 여기에는 다음과 같은 내용이 포함되어 있습니다:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 와 dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

The modules *hashlib*, *posix* and *ssl* use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```
                              Apache License
                        Version 2.0, January 2004
                     https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

The *pyexpat* extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

The `_ctypes` C extension underlying the *ctypes* module is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

zlib 확장은 시스템에서 발견된 zlib 버전이 너무 오래되어서 빌드에 사용될 수 없으면, 포함된 zlib 소스 사본을 사용하여 빌드됩니다:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

tracemalloc 에 의해 사용되는 해시 테이블의 구현은 cfuhash 프로젝트를 기반으로 합니다:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

The `_decimal` C extension underlying the *decimal* module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N 테스트 스위트

The C14N 2.0 test suite in the *test* package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT License:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

C.3.20 asyncio

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD' s "Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.22 Zstandard bindings

Zstandard bindings in `Modules/_zstd` and `Lib/compression/zstd` are based on code from the pyzstd library, copyright Ma Lin and contributors. The pyzstd code is distributed under the 3-Clause BSD License:

```
Copyright (c) 2020-present, Ma Lin and contributors.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
   contributors may be used to endorse or promote products derived from
   this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

APPENDIX D

저작권

파이썬과 이 설명서는:

Copyright © 2001 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

전체 라이센스 및 사용 권한 정보는 역사와 라이센스 에서 제공합니다.

Bibliography

[Frie09] Friedl, Jeffrey. Mastering Regular Expressions. 3rd ed., O' Reilly Media, 2009. 이 책의 세 번째 판은 더는 파이썬을 다루지 않지만, 초판은 훌륭한 정규식 패턴 작성을 아주 자세하게 다루었습니다.

[C99] ISO/IEC 9899:1999. "Programming languages – C." 이 표준의 공개 초안은 https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf에 있습니다.

Python 모듈 목록

_

__future__, 2075
__main__, 2020
_thread, 1105
_tkinter, 1630

a

abc, 2058
aifc, 2293
annotationlib, 2100
argparse, 881
array, 301
ast, 2161
asynchat, 2293
asyncio, 1109
asyncore, 2293
atexit, 2063
audioop, 2294

b

base64, 1394
bdb, 1919
binascii, 1398
bisect, 297
builtins, 2019
bz2, 601

c

calendar, 261
cgi, 2294
cgitb, 2294
chunk, 2294
cmath, 360
cmd, 988
code, 2113
codecs, 198
codeop, 2115
collections, 269
collections.abc, 287
colorsys, 1608
compileall, 2218
compression.zstd, 583
concurrent.futures, 1063
concurrent.interpreters, 1072
configparser, 649
contextlib, 2044
contextvars, 1101
copy, 317
copyreg, 540
cProfile, 1941
crypt, 2294
csv, 641
ctypes, 841
curses (*Unix*), 957
curses.ascii, 984
curses.panel, 987
curses.textpad, 982

d

dataclasses, 2033
datetime, 217
dbm, 545
dbm.dumb, 550
dbm.gnu (*Unix*), 547
dbm.ndbm (*Unix*), 549
dbm.sqlite3 (*All*), 547
decimal, 364
difflib, 164
dis, 2222
distutils, 2294
doctest, 1777

e

email, 1303
email.charset, 1354
email.contentmanager, 1332
email.encoders, 1356
email.errors, 1325
email.generator, 1315
email.header, 1351
email.headerregistry, 1326
email.iterators, 1359
email.message, 1304
email.mime, 1349
email.mime.application, 1350
email.mime.audio, 1350
email.mime.base, 1349

email.mime.image, 1350
email.mime.message, 1351
email.mime.multipart, 1349
email.mime.nonmultipart, 1349
email.mime.text, 1351
email.parser, 1312
email.policy, 1318
email.utils, 1357
encodings, 214
encodings.idna, 215
encodings.mbcs, 216
encodings.utf_8_sig, 216
ensurepip, 1965
enum, 327
errno, 833

f

faulthandler, 1925
fcntl (*Unix*), 2275
filecmp, 499
fileinput, 954
fnmatch, 509
fractions, 393
ftplib, 1517
functools, 440

g

gc, 2077
getopt, 2289
getpass, 953
gettext, 1609
glob, 506
graphlib, 343
grp (*Unix*), 2270
gzip, 598

h

hashlib, 673
heapq, 293
hmac, 685
html, 1401
html.entities, 1406
html.parser, 1402
http, 1506
http.client, 1510
http.cookiejar, 1567
http.cookies, 1563
http.server, 1556

i

idlelib, 1677
imaplib, 1527
imghdr, 2295
imp, 2295
importlib, 2125
importlib.abc, 2128
importlib.machinery, 2135
importlib.metadata, 2150
importlib.resources, 2145
importlib.resources.abc, 2148
importlib.util, 2140
inspect, 2081
io, 757
ipaddress, 1589
itertools, 423

j

json, 1360
json.tool, 1369

k

keyword, 2210

l

linecache, 510
locale, 1617
logging, 784
logging.config, 802
logging.handlers, 814
lzma, 606

m

mailbox, 1370
mailcap, 2295
marshal, 543
math, 350
mimetypes, 1389
mmap, 1297
modulefinder, 2122
msilib, 2295
msvcrt (*Windows*), 2247
multiprocessing, 1012
multiprocessing.connection, 1044
multiprocessing.dummy, 1048
multiprocessing.managers, 1035
multiprocessing.pool, 1041
multiprocessing.shared_memory, 1057
multiprocessing.sharedctypes, 1033

n

netrc, 669
nis, 2295
nntplib, 2295
numbers, 347

o

operator, 451
optparse, 926
os, 689
os.path, 486
ossaudiodev, 2296

p

pathlib, 459
pathlib.types, 485
pdb, 1928

pickle, 523
pickletools, 2245
pipes, 2296
pkgutil, 2119
platform, 828
plistlib, 670
poplib, 1524
posix (*Unix*), 2268
pprint, 319
profile, 1941
pstats, 1943
pty (*Unix*), 2273
pwd (*Unix*), 2269
py_compile, 2217
pyclbr, 2215
pydoc, 1773

q

queue, 1098
quopri, 1400

r

random, 397
re, 142
readline (*Unix*), 184
reprlib, 324
resource (*Unix*), 2278
rlcompleter, 189
runpy, 2123

s

sched, 1096
secrets, 686
select, 1277
selectors, 1284
shelve, 540
shlex, 2263
shutil, 511
signal, 1288
site, 2108
sitecustomize, 2110
smtpd, 2296
smtplib, 1534
sndhdr, 2296
socket, 1212
socketserver, 1547
spwd, 2296
sqlite3, 551
ssl, 1241
stat, 493
statistics, 406
string, 125
string.templatelib, 136
stringprep, 183
struct, 191
subprocess, 1077
sunau, 2296
symtable, 2200
sys, 1981
sys.monitoring, 2008
sysconfig, 2013
syslog (*Unix*), 2282

t

tabnanny, 2214
tarfile, 623
telnetlib, 2297
tempfile, 501
termios (*Unix*), 2271
test, 1893
test.regrtest, 1895
test.support, 1896
test.support.bytecode_helper, 1907
test.support.import_helper, 1910
test.support.os_helper, 1908
test.support.script_helper, 1906
test.support.socket_helper, 1905
test.support.threading_helper, 1907
test.support.warnings_helper, 1912
textwrap, 177
threading, 995
time, 771
timeit, 1947
tkinter, 1627
tkinter.colorchooser (*Tk*), 1640
tkinter.commondialog (*Tk*), 1644
tkinter.dnd (*Tk*), 1647
tkinter.filedialog (*Tk*), 1642
tkinter.font (*Tk*), 1640
tkinter.messagebox (*Tk*), 1644
tkinter.scrolledtext (*Tk*), 1647
tkinter.simpledialog (*Tk*), 1641
tkinter.ttk, 1648
token, 2205
tokenize, 2210
tomllib, 667
trace, 1952
traceback, 2065
tracemalloc, 1954
tty (*Unix*), 2272
turtle, 1678
turtledemo, 1714
types, 311
typing, 1717

u

unicodedata, 180
unittest, 1800
unittest.mock, 1831
urllib, 1478
urllib.error, 1504
urllib.parse, 1496
urllib.request, 1478
urllib.response, 1495
urllib.robotparser, 1505
usercustomize, 2110

uu, 2297
uuid, 1541

V

venv, 1967

W

warnings, 2025
wave, 1605
weakref, 304
webbrowser, 1465
winreg (*Windows*), 2250
winsound (*Windows*), 2258
wsgiref, 1468
wsgiref.handlers, 1473
wsgiref.headers, 1470
wsgiref.simple_server, 1471
wsgiref.types, 1476
wsgiref.util, 1468
wsgiref.validate, 1472

X

xdrlib, 2297
xml, 1407
xml.dom, 1426
xml.dom.minidom, 1436
xml.dom.pulldom, 1440
xml.etree.ElementInclude, 1418
xml.etree.ElementTree, 1408
xml.parsers.expat, 1454
xml.parsers.expat.errors, 1461
xml.parsers.expat.model, 1461
xml.sax, 1442
xml.sax.handler, 1444
xml.sax.saxutils, 1449
xml.sax.xmlreader, 1450
xmlrpc, 1575
xmlrpc.client, 1576
xmlrpc.server, 1583

Z

zipapp, 1976
zipfile, 611
zipimport, 2117
zlib, 594
zoneinfo, 256

색인

알파벳 이외

??
    in regular expressions, 143
..
    in pathnames, 755
..., **2301**
    doctest 에서, 1784
    ellipsis literal, 106, 1720, 1722, 1734, 2301
    인터프리터 프롬프트, 1782, 2001
    자리 표시자, 180, 320, 325
    줄임표 리터럴, 36
. (*dot*)
    in pathnames, 754, 755
    in printf-style formatting, 66, 82
    in regular expressions, 143
! (*exclamation mark*)
    in formatted string literal, 64
- (*minus*)
    binary operator, 39
    in printf-style formatting, 66, 82
    in regular expressions, 144
    unary operator, 39
! (*pdb command*), 1937
? (*question mark*)
    in argparse module, 893
    in AST grammar, 2164
    in regular expressions, 143
    in SQL statements, 568
! (느낌표)
    curses 모듈, 987
    glob 스타일 와일드카드, 506, 509
    struct 포맷 문자열, 192
    명령 인터프리터에서, 989
    문자열 포매팅에서, 127
- (마이너스)
    doctest 에서, 1786
    glob 스타일 와일드카드, 506, 509
    문자열 포매팅에서, 129
. (마침표)
    문자열 포매팅에서, 127
? (물음표)
    glob 스타일 와일드카드, 506, 509
    struct 포맷 문자열, 194, 195
    명령 인터프리터에서, 989
    치환 문자, 202
. (점)
    glob 스타일 와일드카드, 506
! formatted string literal, 64
! f-string, 64
(*hash*)
    in printf-style formatting, 66, 82
    in regular expressions, 150
(해시)
    doctest 에서, 1786
    문자열 포매팅에서, 129
    주석, 2109
$ (*dollar*)
    in regular expressions, 143
$ (달러)
    구성 파일의 보간, 654
    템플릿 문자열에서, 134
    환경 변수 확장, 488
% (*percent*)
    datetime format, 251
    operator, 39
    printf-style formatting, 66, 81
% (퍼센트)
    구성 파일의 보간, 653
    날짜/시간 형식, 776, 778
    환경 변수 확장 (윈도우), 488, 2252
& (*ampersand*)
    operator, 40
(?
    in regular expressions, 144
(?!
    in regular expressions, 146
(?#
    in regular expressions, 146
(?(
    in regular expressions, 147
() (*parentheses*)
    in printf-style formatting, 66, 82
    in regular expressions, 144
(?:
    in regular expressions, 145
(?<!

in regular expressions, 147
(?<=
in regular expressions, 146
(?=
in regular expressions, 146
(?P<
in regular expressions, 146
(?P=
in regular expressions, 146
*?
in regular expressions, 143
* (*asterisk*)
in argparse module, 894
in AST grammar, 2164
in printf-style formatting, 66, 82
in regular expressions, 143
operator, 39
* (애스터리스크)
glob 스타일 와일드 카드, 506, 509
**
glob 스타일 와일드 카드, 507
operator, 39
*+
in regular expressions, 143
+?
in regular expressions, 143
?+
in regular expressions, 143
+ (*plus*)
binary operator, 39
in argparse module, 894
in printf-style formatting, 66, 82
in regular expressions, 143
unary operator, 39
+ (플러스)
doctest 에서, 1786
문자열 포매팅에서, 129
++
in regular expressions, 143
, (쉼표)
문자열 포매팅에서, 130
-
idle 명령줄 옵션, 1674
python--m-py_compile 명령줄 옵션, 2218
/ (*slash*)
in pathnames, 755
operator, 39
//
operator, 39
2자리 숫자 연도, 771
2038년, 771
: (*colon*)
in formatted string literal, 64
in SQL statements, 568
path separator (*POSIX*), 755
: (콜론)
문자열 포매팅에서, 127
; (*semicolon*), 755
< (*less*)
operator, 38
< (작다)
struct 포맷 문자열, 192
문자열 포매팅에서, 129
<<
operator, 40
<=
operator, 38
<BLANKLINE>, 1784
<file>
python--m-py_compile 명령줄 옵션, 2218
!=
operator, 38
= (*equals*)
for help in debugging using string literals, 64
= (등호)
struct 포맷 문자열, 192
문자열 포매팅에서, 129
==
operator, 38
> (*greater*)
operator, 38
> (크다)
struct 포맷 문자열, 192
문자열 포매팅에서, 129
>=
operator, 38
>>
operator, 40
>>>, **2301**
인터프리터 프롬프트, 1782, 2001
@ (*at*)
struct 포맷 문자열, 192
[] (*square brackets*)
in regular expressions, 144
[] (대괄호)
glob 스타일 와일드 카드, 506, 509
문자열 포매팅에서, 127
\ (*backslash*)
in pathnames (*Windows*), 755
in regular expressions, 144, 147
\ (역슬래시)
이스케이프 시퀀스, 202
\\
in regular expressions, 148
\A
in regular expressions, 147
\a
in regular expressions, 148
\B
in regular expressions, 147
\b
in regular expressions, 147, 148
\D
in regular expressions, 148
\d

in regular expressions, 147
\f
in regular expressions, 148
\g
in regular expressions, 153
\N
in regular expressions, 148
이스케이프 시퀀스, 202
\n
in regular expressions, 148
\r
in regular expressions, 148
\S
in regular expressions, 148
\s
in regular expressions, 148
\t
in regular expressions, 148
\U
in regular expressions, 148
이스케이프 시퀀스, 202
\u
in regular expressions, 148
이스케이프 시퀀스, 202
\v
in regular expressions, 148
\W
in regular expressions, 148
\w
in regular expressions, 148
\x
in regular expressions, 148
이스케이프 시퀀스, 202
\Z
in regular expressions, 148
\z
in regular expressions, 148
^ (*caret*)
in regular expressions, 143, 144
operator, 40
^ (캐럿)
curses 모듈, 987
마커, 1784, 2066
문자열 포매팅에서, 129
_ (밑줄)
문자열 포매팅에서, 130
_ (언더스코어)
gettext, 1610
__abs__() (*operator* 모듈), 452
__add__() (*operator* 모듈), 452
__and__() (*enum.Flag* 메서드), 337
__and__() (*operator* 모듈), 452
__args__ (*genericalias*의 속성), 102
__bound__ (*typing.TypeVar*의 속성), 1744
__breakpointhook__ (*sys* 모듈), 1986
__bytes__() (*email.message.EmailMessage* 메서드), 1305
__bytes__() (*email.message.Message* 메서드), 1342
__call__() (*argparse.Action* 메서드), 900
__call__() (*email.headerregistry.HeaderRegistry* 메서드), 1331
__call__() (*enum.EnumType* 메서드), 330
__call__() (*operator* 모듈), 454
__call__() (*weakref.finalize* 메서드), 307
__callback__ (*weakref.ref*의 속성), 305
__cause__ (*BaseException*의 속성), 111
__cause__ (*traceback.TracebackException*의 속성), 2068
__cause__ (예외 어트리뷰트), 111
__ceil__() (*fractions.Fraction* 메서드), 396
__class__ (*unittest.mock.Mock*의 속성), 1841
__code__ (*function object attribute*), 106
__concat__() (*operator* 모듈), 453
__constraints__ (*typing.TypeVar*의 속성), 1744
__contains__() (*email.message.EmailMessage* 메서드), 1306
__contains__() (*email.message.Message* 메서드), 1344
__contains__() (*enum.EnumType* 메서드), 330
__contains__() (*enum.Flag* 메서드), 336
__contains__() (*mailbox.Mailbox* 메서드), 1373
__contains__() (*operator* 모듈), 453
__context__ (*BaseException*의 속성), 111
__context__ (*traceback.TracebackException*의 속성), 2068
__context__ (예외 어트리뷰트), 111
__contravariant__ (*typing.TypeVar*의 속성), 1744
__copy__() (복사 프로토콜), 318
__covariant__ (*typing.TypeVar*의 속성), 1744
__debug__ (내장 변수), 36
__deepcopy__() (복사 프로토콜), 318
__default__ (*typing.ParamSpec*의 속성), 1747
__default__ (*typing.TypeVarTuple*의 속성), 1746
__default__ (*typing.TypeVar*의 속성), 1744
__del__() (*io.IOBase* 메서드), 762
__delitem__() (*email.message.EmailMessage* 메서드), 1306
__delitem__() (*email.message.Message* 메서드), 1344
__delitem__() (*mailbox.Mailbox* 메서드), 1371
__delitem__() (*mailbox.MH* 메서드), 1378
__delitem__() (*operator* 모듈), 454
__dir__() (*enum.Enum* 메서드), 332
__dir__() (*enum.EnumType* 메서드), 330
__dir__() (*unittest.mock.Mock* 메서드), 1838
__displayhook__ (*sys* 모듈), 1986
__doc__ (*definition*의 속성), 107
__enter__() (*contextmanager* 메서드), 98
__enter__() (*winreg.PyHKEY* 메서드), 2258
__eq__() (*email.charset.Charset* 메서드), 1355
__eq__() (*email.header.Header* 메서드), 1353
__eq__() (*instance method*), 38
__eq__() (*memoryview* 메서드), 85
__eq__() (*operator* 모듈), 451
__excepthook__ (*sys* 모듈), 1986
__excepthook__ (*threading* 모듈), 997

__exit__() (*contextmanager* 메서드), 99
__exit__() (*winreg.PyHKEY* 메서드), 2258
__floor__() (*fractions.Fraction* 메서드), 395
__floordiv__() (*operator* 모듈), 452
__format__, 17
__format__() (*datetime.date* 메서드), 227
__format__() (*datetime.datetime* 메서드), 238
__format__() (*datetime.time* 메서드), 243
__format__() (*enum.Enum* 메서드), 334
__format__() (*fractions.Fraction* 메서드), 396
__format__() (*ipaddress.IPv4Address* 메서드), 1592
__format__() (*ipaddress.IPv6Address* 메서드), 1594
__forward_arg__ (*annotationlib.ForwardRef* 의 속성), 2102
__fspath__() (*os.PathLike* 메서드), 693
__future__, **2307**
  module, 2075
__ge__() (*instance method*), 38
__ge__() (*operator* 모듈), 451
__getitem__() (*email.headerregistry.HeaderRegistry* 메서드), 1331
__getitem__() (*email.message.EmailMessage* 메서드), 1306
__getitem__() (*email.message.Message* 메서드), 1344
__getitem__() (*enum.EnumType* 메서드), 331
__getitem__() (*mailbox.Mailbox* 메서드), 1372
__getitem__() (*operator* 모듈), 454
__getitem__() (*re.Match* 메서드), 157
__getnewargs__() (*object* 메서드), 530
__getnewargs_ex__() (*object* 메서드), 530
__getstate__() (*object* 메서드), 530
__getstate__() (복사 프로토콜), 534
__gt__() (*instance method*), 38
__gt__() (*operator* 모듈), 451
__iadd__() (*operator* 모듈), 457
__iand__() (*operator* 모듈), 457
__iconcat__() (*operator* 모듈), 457
__ifloordiv__() (*operator* 모듈), 457
__ilshift__() (*operator* 모듈), 457
__imatmul__() (*operator* 모듈), 457
__imod__() (*operator* 모듈), 457
__import__()
  built-in function, 33
__import__() (*importlib* 모듈), 2126
__imul__() (*operator* 모듈), 457
__index__() (*operator* 모듈), 452
__infer_variance__ (*typing.TypeVar* 의 속성), 1744
__init__() (*asyncio.Future* 메서드), 1201
__init__() (*asyncio.Task* 메서드), 1201
__init__() (*difflib.HtmlDiff* 메서드), 165
__init__() (*enum.Enum* 메서드), 333
__init__() (*logging.Handler* 메서드), 790
__init_subclass__() (*enum.Enum* 메서드), 333
__interactivehook__ (*sys* 모듈), 1996
__inv__() (*operator* 모듈), 452
__invert__() (*operator* 모듈), 452
__ior__() (*operator* 모듈), 457
__ipow__() (*operator* 모듈), 458
__irshift__() (*operator* 모듈), 458
__isub__() (*operator* 모듈), 458
__iter__() (*container* 메서드), 46
__iter__() (*enum.EnumType* 메서드), 331
__iter__() (*iterator* 메서드), 46
__iter__() (*mailbox.Mailbox* 메서드), 1372
__iter__() (*unittest.TestSuite* 메서드), 1821
__itruediv__() (*operator* 모듈), 458
__ixor__() (*operator* 모듈), 458
__le__() (*instance method*), 38
__le__() (*operator* 모듈), 451
__len__() (*email.message.EmailMessage* 메서드), 1306
__len__() (*email.message.Message* 메서드), 1344
__len__() (*enum.EnumType* 메서드), 331
__len__() (*mailbox.Mailbox* 메서드), 1373
__lshift__() (*operator* 모듈), 452
__lt__() (*instance method*), 38
__lt__() (*operator* 모듈), 451
__main__
  module, 2020
  모듈, 2123, 2124
__matmul__() (*operator* 모듈), 453
__members__ (*enum.EnumType*의 속성), 331
__missing__(), 95
__missing__() (*collections.defaultdict* 메서드), 278
__mod__() (*operator* 모듈), 453
__module__ (*definition*의 속성), 107
__module__ (*typing.NewType*의 속성), 1751
__module__ (*typing.TypeAliasType*의 속성), 1749
__mul__() (*operator* 모듈), 453
__mutable_keys__ (*typing.TypedDict*의 속성), 1756
__name__ (*definition*의 속성), 107
__name__ (*property*의 속성), 27
__name__ (*typing.NewType*의 속성), 1752
__name__ (*typing.ParamSpec*의 속성), 1747
__name__ (*typing.TypeAliasType*의 속성), 1749
__name__ (*typing.TypeVarTuple*의 속성), 1746
__name__ (*typing.TypeVar*의 속성), 1743
__ne__() (*email.charset.Charset* 메서드), 1356
__ne__() (*email.header.Header* 메서드), 1353
__ne__() (*instance method*), 38
__ne__() (*operator* 모듈), 451
__neg__() (*operator* 모듈), 453
__new__() (*enum.Enum* 메서드), 333
__new__() (*string.templatelib.Interpolation* 메서드), 141
__new__() (*string.templatelib.Template* 메서드), 139
__next__() (*csv.csvreader* 메서드), 647
__next__() (*iterator* 메서드), 46
__not__() (*operator* 모듈), 452
__notes__ (*BaseException*의 속성), 112
__notes__ (*traceback.TracebackException* 의 속성), 2068
__optional_keys__ (*typing.TypedDict* 의 속성), 1756

__or__() (*enum.Flag* 메서드), 337
__or__() (*operator* 모듈), 453
__origin__ (*genericalias*의 속성), 102
__parameters__ (*genericalias*의 속성), 103
__pointer_type__ (*ctypes._CData*의 속성), 871
__pos__() (*operator* 모듈), 453
__post_init__() (*dataclasses* 모듈), 2040
__pow__() (*operator* 모듈), 453
__qualname__ (*definition*의 속성), 107
__readonly_keys__ (*typing.TypedDict* 의 속성), 1756
__reduce__() (*object* 메서드), 531
__reduce_ex__() (*object* 메서드), 531
__replace__() (치환 프로토콜), 318
__repr__() (*enum.Enum* 메서드), 333
__repr__() (*multiprocessing.managers.BaseProxy* 메서드), 1041
__repr__() (*netrc.netrc* 메서드), 670
__required_keys__ (*typing.TypedDict* 의 속성), 1756
__reversed__() (*enum.EnumType* 메서드), 331
__round__() (*fractions.Fraction* 메서드), 396
__rshift__() (*operator* 모듈), 453
__setitem__() (*email.message.EmailMessage* 메서드), 1306
__setitem__() (*email.message.Message* 메서드), 1344
__setitem__() (*mailbox.Mailbox* 메서드), 1372
__setitem__() (*mailbox.Maildir* 메서드), 1376
__setitem__() (*operator* 모듈), 454
__setstate__() (*object* 메서드), 530
__setstate__() (복사 프로토콜), 534
__slots__, **2315**
__stderr__ (*sys* 모듈), 2006
__stdin__ (*sys* 모듈), 2006
__stdout__ (*sys* 모듈), 2006
__str__() (*datetime.date* 메서드), 227
__str__() (*datetime.datetime* 메서드), 237
__str__() (*datetime.time* 메서드), 243
__str__() (*email.charset.Charset* 메서드), 1355
__str__() (*email.header.Header* 메서드), 1353
__str__() (*email.headerregistry.Address* 메서드), 1331
__str__() (*email.headerregistry.Group* 메서드), 1332
__str__() (*email.message.EmailMessage* 메서드), 1305
__str__() (*email.message.Message* 메서드), 1342
__str__() (*enum.Enum* 메서드), 334
__str__() (*multiprocessing.managers.BaseProxy* 메서드), 1041
__sub__() (*operator* 모듈), 453
__subclasshook__() (*abc.ABCMeta* 메서드), 2059
__supertype__ (*typing.NewType*의 속성), 1752
__suppress_context__ (*BaseException* 의 속성), 111
__suppress_context__ (*traceback.TracebackException*의 속성), 2068
__suppress_context__ (예외 어트리뷰트), 111
__test__ (*module*의 속성), 1781
__total__ (*typing.TypedDict*의 속성), 1755
__traceback__ (*BaseException*의 속성), 112
__truediv__() (*importlib.abc.Traversable* 메서드), 2134
__truediv__() (*importlib.resources.abc.Traversable* 메서드), 2150
__truediv__() (*operator* 모듈), 453
__type_params__ (*definition*의 속성), 107
__type_params__ (*typing.TypeAliasType* 의 속성), 1749
__unpacked__ (*genericalias*의 속성), 103
__unraisablehook__ (*sys* 모듈), 1986
__value__ (*typing.TypeAliasType*의 속성), 1749
__version__ (*curses* 모듈), 971
__xor__() (*enum.Flag* 메서드), 337
__xor__() (*operator* 모듈), 453
_add_alias_() (*enum.Enum* 메서드), 334
_add_value_alias_() (*enum.Enum* 메서드), 335
align (*ctypes.Structure*의 속성), 875
anonymous (*ctypes.Structure*의 속성), 876
_asdict() (*collections.somenamedtuple* 메서드), 281
_b_base_ (*ctypes._CData*의 속성), 871
_b_needsfree_ (*ctypes._CData*의 속성), 871
_callmethod() (*multiprocessing.managers.BaseProxy* 메서드), 1040
_CData (*ctypes* 클래스), 871
_CFuncPtr (*ctypes* 클래스), 864
_clear_internal_caches() (*sys* 모듈), 1983
_clear_type_cache() (*sys* 모듈), 1983
_current_exceptions() (*sys* 모듈), 1983
_current_frames() (*sys* 모듈), 1983
_debugmallocstats() (*sys* 모듈), 1984
_emscripten_info (*sys* 모듈), 1985
_enablelegacywindowsfsencoding() (*sys* 모듈), 2004
_enter_task() (*asyncio* 모듈), 1201
_exit() (*os* 모듈), 740
_Feature (*__future__* 클래스), 2076
_field_defaults (*collections.somenamedtuple*의 속성), 282
_field_types (*ast.AST*의 속성), 2165
_fields (*ast.AST*의 속성), 2164
_fields (*collections.somenamedtuple*의 속성), 281
fields (*ctypes.Structure*의 속성), 875
_flush() (*wsgiref.handlers.BaseHandler* 메서드), 1474
_for_archive() (*zipfile.ZipInfo* 메서드), 612
_generate_next_value_() (*enum.Enum* 메서드), 332
_get_child_mock() (*unittest.mock.Mock* 메서드), 1838
_get_preferred_schemes() (*sysconfig* 모듈), 2017
_getframe() (*sys* 모듈), 1993
_getframemodulename() (*sys* 모듈), 1993
_getvalue() (*multiprocessing.managers.BaseProxy* 메서드), 1041
_handle (*ctypes.PyDLL*의 속성), 863

ignore (*enum.Enum*의 속성), 332
_incompatible_extension_module_restrictions() (*importlib.util* 모듈), 2142
_is_gil_enabled() (*sys* 모듈), 1996
_is_immortal() (*sys* 모듈), 1998
_is_interned() (*sys* 모듈), 1998
_jit (*sys* 모듈), 1997
_jit.is_active() (*sys* 모듈), 1997
_jit.is_available() (*sys* 모듈), 1997
_jit.is_enabled() (*sys* 모듈), 1997
layout (*ctypes.Structure*의 속성), 876
_leave_task() (*asyncio* 모듈), 1202
length (*ctypes.Array*의 속성), 878
_locale
    모듈, 1617
_log (*logging.LoggerAdapter*의 속성), 797
_make() (*collections.somenamedtuple*의 클래스 메서드), 281
_makeResult() (*unittest.TextTestRunner* 메서드), 1827
missing() (*enum.Enum* 메서드), 333
_name (*ctypes.PyDLL*의 속성), 863
name (*enum.Enum*의 속성), 332
_numeric_repr_() (*enum.Flag* 메서드), 337
_objects (*ctypes._CData*의 속성), 871
order (*enum.Enum*의 속성), 332
pack (*ctypes.Structure*의 속성), 875
_parse() (*gettext.NullTranslations* 메서드), 1611
_Pointer (*ctypes* 클래스), 878
_register_task() (*asyncio* 모듈), 1201
_replace() (*collections.somenamedtuple* 메서드), 281
_setroot() (*xml.etree.ElementTree.ElementTree* 메서드), 1421
_SimpleCData (*ctypes* 클래스), 872
_structure() (*email.iterators* 모듈), 1360
_thread
    module, 1105
_tkinter
    module, 1630
type (*ctypes._Pointer*의 속성), 878
type (*ctypes.Array*의 속성), 878
_unregister_task() (*asyncio* 모듈), 1201
value (*enum.Enum*의 속성), 332
_write() (*wsgiref.handlers.BaseHandler* 메서드), 1474
_xoptions (*sys* 모듈), 2008
{} (*curly brackets*)
    in formatted string literal, 64
    in regular expressions, 143
{} (중괄호)
    문자열 포매팅에서, 127
| (*vertical bar*)
    in regular expressions, 144
    operator, 40
~ (*tilde*)
    operator, 40
~ (물결표)
    홈 디렉터리 확장, 487
가상
    환경, 1967
감사, 1982
감사 이벤트, 1915
객체
    마샬링, 523
    지속적, 523
    직렬화하기, 523
    코드, 543
    트레이스백, 1986, 2065
    평탄화, 523
    피클링, 523
검색
    경로, 모듈, 510, 1999, 2108
결정론적 프로파일링, 1939
경고, 2025
경량 프로세스, 1105
경로
    구성 파일, 2109
    모듈 검색, 510, 1999, 2108
    연산, 486
경로 브라우저, 1666
공유 메모리, 1057
구문 분석
    URL, 1496
구성
    파일, 649
    파일, 경로, 2109
    파일, 디버거, 1933
구성 정보, 2013
구조체
    C, 191
그래픽 사용자 인터페이스, 1627
그리니치 표준시, 772
상대적
    URL, 1496
서버
    WWW, 1556
성능, 1947
세마포어, 이진, 1105
소켓
    모듈, 1465
수학
    모듈, 364
순환 중복 검사, 595
쉘프
    모듈, 543
스레드
    POSIX, 1105
스크립트 실행, 1668
스택 뷰어, 1669
스트림, 198
    stackable, 198
스페이스
    문자열 포매팅에서, 129
시그널
    모듈, 1107
실행 가능한 Zip 파일, 1976

암호화, 673
에러
    logging, 784
에러 처리기의 이름
    backslashreplace, 202
    ignore, 202
    namereplace, 202
    replace, 202
    strict, 202
    surrogateescape, 202
    surrogatepass, 202
    xmlcharrefreplace, 202
에포크, 771
에포크 이후 초, 771
연쇄
    예외, 111
예외
    연쇄, 111
외부 데이터 표현, 524
운영 체제
    모듈, 2268
월드 와이드 웹, 1465, 1496, 1505
윈도우 ini 파일, 649
유니버설 줄 넘김
    :mod:`subprocess` 모듈, 1080
    csv.reader 함수, 641
    importlib.abc.InspectLoader.get_source 메서드, 2130
    io.IncrementalNewlineDecoder 클래스, 770
    io.TextIOWrapper 클래스, 768
유니코드, 180, 198
    데이터베이스, 180
이름 지정 공유 메모리, 1057
이벤트 (위젯), 1638
이벤트 스케줄링, 1096
이진 세마포어, 1105
인코딩
    base64, 1394
    quoted-printable, 1400
인터넷, 1465
인터프리터 프롬프트, 2001
일광 절약 시간, 772
임시
    파일, 501
    파일명, 501
임포트
    문장, 2109
종단점, 1670
지속성, 523
지속적
    객체, 523
직렬화하기
    객체, 523
창 관리자 (위젯), 1636
체크섬
    순환 중복 검사, 595
추적 함수, 998, 1993, 2002
컴파일
    내장 함수, 313
코덱, 198
    decode, 198
    encode, 198
코드 객체, 543
콘텐츠 유형
    MIME, 1389

A

-a
    ast 명령줄 옵션, 2200
    pickletools 명령줄 옵션, 2246
A (*re* 모듈), 149
a2b_base64() (*binascii* 모듈), 1398
a2b_hex() (*binascii* 모듈), 1399
a2b_qp() (*binascii* 모듈), 1398
a2b_uu() (*binascii* 모듈), 1398
a85decode() (*base64* 모듈), 1396
a85encode() (*base64* 모듈), 1396
A_ALTCHARSET (*curses* 모듈), 972
A_ATTRIBUTES (*curses* 모듈), 973
A_BLINK (*curses* 모듈), 972
A_BOLD (*curses* 모듈), 972
A_CHARTEXT (*curses* 모듈), 973
A_COLOR (*curses* 모듈), 973
A_DIM (*curses* 모듈), 972
A_HORIZONTAL (*curses* 모듈), 972
A_INVIS (*curses* 모듈), 972
A_ITALIC (*curses* 모듈), 972
A_LEFT (*curses* 모듈), 972
A_LOW (*curses* 모듈), 972
A_NORMAL (*curses* 모듈), 972
A_PROTECT (*curses* 모듈), 972
A_REVERSE (*curses* 모듈), 972
A_RIGHT (*curses* 모듈), 972
A_STANDOUT (*curses* 모듈), 972
A_TOP (*curses* 모듈), 972
A_UNDERLINE (*curses* 모듈), 972
A_VERTICAL (*curses* 모듈), 972
abc
    module, 2058
ABC (*abc* 클래스), 2059
ABCMeta (*abc* 클래스), 2059
ABDAY_1 (*locale* 모듈), 1619
ABDAY_2 (*locale* 모듈), 1619
ABDAY_3 (*locale* 모듈), 1619
ABDAY_4 (*locale* 모듈), 1619
ABDAY_5 (*locale* 모듈), 1619
ABDAY_6 (*locale* 모듈), 1619
ABDAY_7 (*locale* 모듈), 1619
abiflags (*sys* 모듈), 1981
ABMON_1 (*locale* 모듈), 1620
ABMON_2 (*locale* 모듈), 1620
ABMON_3 (*locale* 모듈), 1620
ABMON_4 (*locale* 모듈), 1620
ABMON_5 (*locale* 모듈), 1620
ABMON_6 (*locale* 모듈), 1620

ABMON_7 (*locale* 모듈), 1620
ABMON_8 (*locale* 모듈), 1620
ABMON_9 (*locale* 모듈), 1620
ABMON_10 (*locale* 모듈), 1620
ABMON_11 (*locale* 모듈), 1620
ABMON_12 (*locale* 모듈), 1620
ABORT (*tkinter.messagebox* 모듈), 1646
abort() (*asyncio.Barrier* 메서드), 1146
abort() (*asyncio.DatagramTransport* 메서드), 1188
abort() (*asyncio.WriteTransport* 메서드), 1187
abort() (*ftplib.FTP* 메서드), 1519
abort() (*os* 모듈), 738
abort() (*threading.Barrier* 메서드), 1011
abort_clients() (*asyncio.Server* 메서드), 1176
ABORTRETRYIGNORE (*tkinter.messagebox* 모듈), 1646
above() (*curses.panel.Panel* 메서드), 987
ABOVE_NORMAL_PRIORITY_CLASS (*subprocess* 모듈), 1090
abs()
    built-in function, 8
abs() (*decimal.Context* 메서드), 379
abs() (*operator* 모듈), 452
absolute() (*pathlib.Path* 메서드), 473
absolute_import (*__future__* 모듈), 2076
AbsoluteLinkError, 626
AbsolutePathError, 625
abspath() (*os.path* 모듈), 486
abstract base class (추상 베이스 클래스), **2301**
AbstractAsyncContextManager (*contextlib* 클래스), 2044
AbstractBasicAuthHandler (*urllib.request* 클래스), 1483
abstractclassmethod() (*abc* 모듈), 2061
AbstractContextManager (*contextlib* 클래스), 2044
AbstractDigestAuthHandler (*urllib.request* 클래스), 1483
AbstractEventLoop (*asyncio* 클래스), 1178
AbstractEventLoopPolicy (*asyncio* 클래스), 1198
abstractmethod() (*abc* 모듈), 2060
abstractproperty() (*abc* 모듈), 2062
AbstractSet (*typing* 클래스), 1769
abstractstaticmethod() (*abc* 모듈), 2062
accept() (*multiprocessing.connection.Listener* 메서드), 1045
accept() (*socket.socket* 메서드), 1230
access() (*os* 모듈), 712
accumulate() (*itertools* 모듈), 425
ACK (*curses.ascii* 모듈), 984
aclose() (*contextlib.AsyncExitStack* 메서드), 2053
aclosing() (*contextlib* 모듈), 2046
acos() (*cmath* 모듈), 362
acos() (*math* 모듈), 357
acosh() (*cmath* 모듈), 362
acosh() (*math* 모듈), 358
acquire() (*_thread.lock* 메서드), 1107
acquire() (*asyncio.Condition* 메서드), 1143
acquire() (*asyncio.Lock* 메서드), 1141
acquire() (*asyncio.Semaphore* 메서드), 1144
acquire() (*logging.Handler* 메서드), 790
acquire() (*multiprocessing.Lock* 메서드), 1030
acquire() (*multiprocessing.RLock* 메서드), 1031
acquire() (*threading.Condition* 메서드), 1007
acquire() (*threading.Lock* 메서드), 1004
acquire() (*threading.RLock* 메서드), 1005
acquire() (*threading.Semaphore* 메서드), 1008
ACS_BBSS (*curses* 모듈), 979
ACS_BLOCK (*curses* 모듈), 979
ACS_BOARD (*curses* 모듈), 979
ACS_BSBS (*curses* 모듈), 979
ACS_BSSB (*curses* 모듈), 979
ACS_BSSS (*curses* 모듈), 979
ACS_BTEE (*curses* 모듈), 979
ACS_BULLET (*curses* 모듈), 979
ACS_CKBOARD (*curses* 모듈), 979
ACS_DARROW (*curses* 모듈), 979
ACS_DEGREE (*curses* 모듈), 979
ACS_DIAMOND (*curses* 모듈), 979
ACS_GEQUAL (*curses* 모듈), 979
ACS_HLINE (*curses* 모듈), 979
ACS_LANTERN (*curses* 모듈), 979
ACS_LARROW (*curses* 모듈), 979
ACS_LEQUAL (*curses* 모듈), 979
ACS_LLCORNER (*curses* 모듈), 979
ACS_LRCORNER (*curses* 모듈), 980
ACS_LTEE (*curses* 모듈), 980
ACS_NEQUAL (*curses* 모듈), 980
ACS_PI (*curses* 모듈), 980
ACS_PLMINUS (*curses* 모듈), 980
ACS_PLUS (*curses* 모듈), 980
ACS_RARROW (*curses* 모듈), 980
ACS_RTEE (*curses* 모듈), 980
ACS_S1 (*curses* 모듈), 980
ACS_S3 (*curses* 모듈), 980
ACS_S7 (*curses* 모듈), 980
ACS_S9 (*curses* 모듈), 980
ACS_SBBS (*curses* 모듈), 980
ACS_SBSB (*curses* 모듈), 980
ACS_SBSS (*curses* 모듈), 980
ACS_SSBB (*curses* 모듈), 980
ACS_SSBS (*curses* 모듈), 980
ACS_SSSB (*curses* 모듈), 980
ACS_SSSS (*curses* 모듈), 981
ACS_STERLING (*curses* 모듈), 981
ACS_TTEE (*curses* 모듈), 981
ACS_UARROW (*curses* 모듈), 981
ACS_ULCORNER (*curses* 모듈), 981
ACS_URCORNER (*curses* 모듈), 981
ACS_VLINE (*curses* 모듈), 981
Action (*argparse* 클래스), 900
action (*optparse.Option*의 속성), 940
ACTIONS (*optparse.Option*의 속성), 952
activate_stack_trampoline() (*sys* 모듈), 2004
active_children() (*multiprocessing* 모듈), 1026
active_count() (*threading* 모듈), 997
actual() (*tkinter.font.Font* 메서드), 1641
Add (*ast* 클래스), 2171

add() (*decimal.Context* 메서드), 379
add() (*frozenset* 메서드), 93
add() (*graphlib.TopologicalSorter* 메서드), 344
add() (*mailbox.Mailbox* 메서드), 1371
add() (*mailbox.Maildir* 메서드), 1376
add() (*operator* 모듈), 452
add() (*pstats.Stats* 메서드), 1943
add() (*tarfile.TarFile* 메서드), 630
add() (*tkinter.ttk.Notebook* 메서드), 1654
add_alias() (*email.charset* 모듈), 1356
add_alternative() (*email.message.EmailMessage* 메서드), 1311
add_argument() (*argparse.ArgumentParser* 메서드), 890
add_argument_group() (*argparse.ArgumentParser* 메서드), 907
add_attachment() (*email.message.EmailMessage* 메서드), 1311
add_cgi_vars() (*wsgiref.handlers.BaseHandler* 메서드), 1474
add_charset() (*email.charset* 모듈), 1356
add_codec() (*email.charset* 모듈), 1356
add_cookie_header() (*http.cookiejar.CookieJar* 메서드), 1569
add_dll_directory() (*os* 모듈), 739
add_done_callback() (*asyncio.Future* 메서드), 1182
add_done_callback() (*asyncio.Task* 메서드), 1130
add_done_callback() (*concurrent.futures.Future* 메서드), 1070
add_fallback() (*gettext.NullTranslations* 메서드), 1611
add_flag() (*mailbox.Maildir* 메서드), 1375
add_flag() (*mailbox.MaildirMessage* 메서드), 1381
add_flag() (*mailbox.mboxMessage* 메서드), 1383
add_flag() (*mailbox.MMDFMessage* 메서드), 1387
add_folder() (*mailbox.Maildir* 메서드), 1374
add_folder() (*mailbox.MH* 메서드), 1378
add_get_handler() (*email.contentmanager.ContentManager* 메서드), 1332
add_handler() (*urllib.request.OpenerDirector* 메서드), 1485
add_header() (*email.message.EmailMessage* 메서드), 1307
add_header() (*email.message.Message* 메서드), 1345
add_header() (*urllib.request.Request* 메서드), 1485
add_header() (*wsgiref.headers.Headers* 메서드), 1470
add_history() (*readline* 모듈), 186
add_label() (*mailbox.BabylMessage* 메서드), 1385
add_mutually_exclusive_group() (*argparse.ArgumentParser* 메서드), 908
add_note() (*BaseException* 메서드), 112
add_option() (*optparse.OptionParser* 메서드), 939
add_parent() (*urllib.request.BaseHandler* 메서드), 1486
add_password() (*urllib.request.HTTPPasswordMgr* 메서드), 1489
add_password() (*urllib.request.HTTPPasswordMgrWithPriorAuth* 메서드), 1489
add_reader() (*asyncio.loop* 메서드), 1167
add_related() (*email.message.EmailMessage* 메서드), 1311
add_section() (*configparser.ConfigParser* 메서드), 663
add_section() (*configparser.RawConfigParser* 메서드), 666
add_sequence() (*mailbox.MHMessage* 메서드), 1384
add_set_handler() (*email.contentmanager.ContentManager* 메서드), 1333
add_signal_handler() (*asyncio.loop* 메서드), 1171
add_subparsers() (*argparse.ArgumentParser* 메서드), 903
add_type() (*mimetypes* 모듈), 1391
add_type() (*mimetypes.MimeTypes* 메서드), 1392
add_unredirected_header() (*urllib.request.Request* 메서드), 1485
add_writer() (*asyncio.loop* 메서드), 1168
addAsyncCleanup() (*unittest.IsolatedAsyncioTestCase* 메서드), 1820
addaudithook() (*sys* 모듈), 1981
addch() (*curses.window* 메서드), 964
addClassCleanup() (*unittest.TestCase*의 클래스 메서드), 1819
addCleanup() (*unittest.TestCase* 메서드), 1818
addcomponent() (*turtle.Shape* 메서드), 1710
addDuration() (*unittest.TestResult* 메서드), 1827
addError() (*unittest.TestResult* 메서드), 1826
addExpectedFailure() (*unittest.TestResult* 메서드), 1826
addFailure() (*unittest.TestResult* 메서드), 1826
addfile() (*tarfile.TarFile* 메서드), 630
addFilter() (*logging.Handler* 메서드), 790
addFilter() (*logging.Logger* 메서드), 788
addHandler() (*logging.Logger* 메서드), 789
addinfourl (*urllib.response* 클래스), 1495
addLevelName() (*logging* 모듈), 798
addModuleCleanup() (*unittest* 모듈), 1830
addnstr() (*curses.window* 메서드), 965
AddPackagePath() (*modulefinder* 모듈), 2122
addr_spec (*email.headerregistry.Address*의 속성), 1331
Address (*email.headerregistry* 클래스), 1331
address (*email.headerregistry.SingleAddressHeader*의 속성), 1329
address (*multiprocessing.connection.Listener*의 속성), 1045
address (*multiprocessing.managers.BaseManager*의 속성), 1036
address_exclude() (*ipaddress.IPv4Network* 메서드), 1597
address_exclude() (*ipaddress.IPv6Network* 메서드), 1600

address_family (*socketserver.BaseServer*의 속성), 1550
address_string() (*http.server.BaseHTTPRequestHandler* 메서드), 1560
addresses (*email.headerregistry.AddressHeader*의 속성), 1328
addresses (*email.headerregistry.Group*의 속성), 1332
AddressHeader (*email.headerregistry* 클래스), 1328
addressof() (*ctypes* 모듈), 867
AddressValueError, 1603
addshape() (*turtle* 모듈), 1707
addsitedir() (*site* 모듈), 2111
addSkip() (*unittest.TestResult* 메서드), 1826
addstr() (*curses.window* 메서드), 965
addSubTest() (*unittest.TestResult* 메서드), 1826
addSuccess() (*unittest.TestResult* 메서드), 1826
addTest() (*unittest.TestSuite* 메서드), 1821
addTests() (*unittest.TestSuite* 메서드), 1821
addTypeEqualityFunc() (*unittest.TestCase* 메서드), 1816
addUnexpectedSuccess() (*unittest.TestResult* 메서드), 1826
adjust_int_max_str_digits() (*test.support* 모듈), 1905
adjusted() (*decimal.Decimal* 메서드), 369
adler32() (*zlib* 모듈), 594
AF_ALG (*socket* 모듈), 1219
AF_BLUETOOTH (*socket* 모듈), 1220
AF_CAN (*socket* 모듈), 1218
AF_DIVERT (*socket* 모듈), 1219
AF_HYPERV (*socket* 모듈), 1222
AF_INET (*socket* 모듈), 1216
AF_INET6 (*socket* 모듈), 1216
AF_LINK (*socket* 모듈), 1220
AF_PACKET (*socket* 모듈), 1219
AF_QIPCRTR (*socket* 모듈), 1221
AF_RDS (*socket* 모듈), 1219
AF_UNIX (*socket* 모듈), 1216
AF_UNSPEC (*socket* 모듈), 1216
AF_VSOCK (*socket* 모듈), 1220
aifc
    module, 2293
aiter()
    built-in function, 8
alarm() (*signal* 모듈), 1291
ALERT_DESCRIPTION_HANDSHAKE_FAILURE (*ssl* 모듈), 1253
ALERT_DESCRIPTION_INTERNAL_ERROR (*ssl* 모듈), 1253
AlertDescription (*ssl* 클래스), 1253
algorithm (*sys.hash_info*의 속성), 1995
algorithms_available (*hashlib* 모듈), 675
algorithms_guaranteed (*hashlib* 모듈), 675
Alias
    Generic, 99
alias (*ast* 클래스), 2179
alias (*pdb command*), 1937
alignment() (*ctypes* 모듈), 867
alive (*weakref.finalize*의 속성), 307
all()
    built-in function, 8
ALL_COMPLETED (*asyncio* 모듈), 1125
ALL_COMPLETED (*concurrent.futures* 모듈), 1071
all_errors (*ftplib* 모듈), 1524
all_features (*xml.sax.handler* 모듈), 1445
all_frames (*tracemalloc.Filter*의 속성), 1961
all_properties (*xml.sax.handler* 모듈), 1446
all_suffixes() (*importlib.machinery* 모듈), 2135
all_tasks() (*asyncio* 모듈), 1128
allocate_lock() (*_thread* 모듈), 1106
ALLOW_MISSING (*os.path* 모듈), 491
allow_reuse_address (*socketserver.BaseServer*의 속성), 1550
allowed_domains() (*http.cookiejar.DefaultCookiePolicy* 메서드), 1572
alt() (*curses.ascii* 모듈), 987
ALT_DIGITS (*locale* 모듈), 1621
--altinstall
    ensurepip 명령줄 옵션, 1966
altsep (*os* 모듈), 755
altzone (*time* 모듈), 783
ALWAYS_EQ (*test.support* 모듈), 1898
ALWAYS_TYPED_ACTIONS (*optparse.Option*의 속성), 952
AmbiguousOptionError, 953
AMPER (*token* 모듈), 2207
AMPEREQUAL (*token* 모듈), 2209
Anchor (*importlib.resources* 클래스), 2146
anchor (*pathlib.PurePath*의 속성), 465
and
    operator, 37, 38
And (*ast* 클래스), 2171
and_() (*operator* 모듈), 452
android_ver() (*platform* 모듈), 832
anext()
    built-in function, 8
AnnAssign (*ast* 클래스), 2176
--annotate
    pickletools 명령줄 옵션, 2246
annotate function, **2301**
Annotated (*typing* 모듈), 1736
annotation
    type annotation; type hint, 99
annotation (*inspect.Parameter*의 속성), 2090
ANNOTATION (*symtable.SymbolTableType*의 속성), 2201
annotation (어노테이션), **2301**
annotationlib
    module, 2100
annotations (*__future__* 모듈), 2076
annotations_to_string() (*annotationlib* 모듈), 2103
answer_challenge() (*multiprocessing.connection* 모듈), 1044
anticipate_failure() (*test.support* 모듈), 1901

Any (*typing* 모듈), 1729
ANY (*unittest.mock* 모듈), 1866
any()
    built-in function, 9
ANY_CONTIGUOUS (*inspect.BufferFlags*의 속성), 2099
AnyStr (*typing* 모듈), 1729
api_version (*sys* 모듈), 2007
apilevel (*sqlite3* 모듈), 556
apop() (*poplib.POP3* 메서드), 1525
append() (*array.array* 메서드), 302
append() (*collections.deque* 메서드), 275
append() (*email.header.Header* 메서드), 1352
append() (*imaplib.IMAP4* 메서드), 1529
append() (*sequence* 메서드), 49
append() (*xml.etree.ElementTree.Element* 메서드), 1420
append_history_file() (*readline* 모듈), 185
appendChild() (*xml.dom.Node* 메서드), 1430
appendleft() (*collections.deque* 메서드), 275
AppleFrameworkLoader (*importlib.machinery* 클래스), 2139
application_uri() (*wsgiref.util* 모듈), 1468
apply() (*multiprocessing.pool.Pool* 메서드), 1042
apply_async() (*multiprocessing.pool.Pool* 메서드), 1042
apply_defaults() (*inspect.BoundArguments* 메서드), 2091
APRIL (*calendar* 모듈), 266
architecture() (*platform* 모듈), 828
archive (*zipimport.zipimporter*의 속성), 2119
AREGTYPE (*tarfile* 모듈), 626
aRepr (*reprlib* 모듈), 325
arg (*ast* 클래스), 2192
argparse
    module, 881
args (*BaseException*의 속성), 112
args (*functools.partial*의 속성), 451
args (*inspect.BoundArguments*의 속성), 2091
args (*pdb command*), 1935
args (*subprocess.CompletedProcess*의 속성), 1078
args (*subprocess.Popen*의 속성), 1087
args (*typing.ParamSpec*의 속성), 1747
args_from_interpreter_flags() (*test.support* 모듈), 1900
argtypes (*ctypes._CFuncPtr*의 속성), 864
argument (인자), **2301**
ArgumentDefaultsHelpFormatter (*argparse* 클래스), 885
ArgumentError, 878, 912
ArgumentParser (*argparse* 클래스), 882
arguments (*ast* 클래스), 2192
arguments (*inspect.BoundArguments*의 속성), 2091
ArgumentTypeError, 912
argv (*sys* 모듈), 1982
arithmetic, 39
ArithmeticError, 113
array
    module, 67, 301
array (*array* 클래스), 301
Array (*ctypes* 클래스), 878
ARRAY() (*ctypes* 모듈), 878
Array() (*multiprocessing* 모듈), 1032
Array() (*multiprocessing.managers.SyncManager* 메서드), 1037
Array() (*multiprocessing.sharedctypes* 모듈), 1033
arraysize (*sqlite3.Cursor*의 속성), 569
as_bytes() (*email.message.EmailMessage* 메서드), 1305
as_bytes() (*email.message.Message* 메서드), 1342
as_completed() (*asyncio* 모듈), 1125
as_completed() (*concurrent.futures* 모듈), 1071
as_digested_dict (*compression.zstd.ZstdDict*의 속성), 589
as_file() (*importlib.resources* 모듈), 2146
as_integer_ratio(), 393
as_integer_ratio() (*decimal.Decimal* 메서드), 370
as_integer_ratio() (*float* 메서드), 43
as_integer_ratio() (*fractions.Fraction* 메서드), 395
as_integer_ratio() (*int* 메서드), 43
as_posix() (*pathlib.PurePath* 메서드), 467
as_string() (*email.message.EmailMessage* 메서드), 1305
as_string() (*email.message.Message* 메서드), 1341
as_tuple() (*decimal.Decimal* 메서드), 370
as_undigested_dict (*compression.zstd.ZstdDict*의 속성), 589
as_uri() (*pathlib.Path* 메서드), 472
ASCII (*re* 모듈), 149
ascii()
    built-in function, 9
ascii() (*curses.ascii* 모듈), 987
ascii_letters (*string* 모듈), 125
ascii_lowercase (*string* 모듈), 125
ascii_uppercase (*string* 모듈), 125
asctime() (*time* 모듈), 772
asdict() (*dataclasses* 모듈), 2037
asin() (*cmath* 모듈), 362
asin() (*math* 모듈), 357
asinh() (*cmath* 모듈), 362
asinh() (*math* 모듈), 358
askcolor() (*tkinter.colorchooser* 모듈), 1640
askdirectory() (*tkinter.filedialog* 모듈), 1643
askfloat() (*tkinter.simpledialog* 모듈), 1642
askinteger() (*tkinter.simpledialog* 모듈), 1642
askokcancel() (*tkinter.messagebox* 모듈), 1646
askopenfile() (*tkinter.filedialog* 모듈), 1642
askopenfilename() (*tkinter.filedialog* 모듈), 1643
askopenfilenames() (*tkinter.filedialog* 모듈), 1643
askopenfiles() (*tkinter.filedialog* 모듈), 1642
askquestion() (*tkinter.messagebox* 모듈), 1645
askretrycancel() (*tkinter.messagebox* 모듈), 1646
asksaveasfile() (*tkinter.filedialog* 모듈), 1643
asksaveasfilename() (*tkinter.filedialog* 모듈), 1643
askstring() (*tkinter.simpledialog* 모듈), 1642
askyesno() (*tkinter.messagebox* 모듈), 1646

askyesnocancel() (*tkinter.messagebox* 모듈), 1646
assert
    문 장, 113
Assert (*ast* 클래스), 2178
assert_any_await() (*unittest.mock.AsyncMock* 메서드), 1846
assert_any_call() (*unittest.mock.Mock* 메서드), 1836
assert_awaited() (*unittest.mock.AsyncMock* 메서드), 1844
assert_awaited_once() (*unittest.mock.AsyncMock* 메서드), 1845
assert_awaited_once_with() (*unittest.mock.AsyncMock* 메서드), 1845
assert_awaited_with() (*unittest.mock.AsyncMock* 메서드), 1845
assert_called() (*unittest.mock.Mock* 메서드), 1835
assert_called_once() (*unittest.mock.Mock* 메서드), 1835
assert_called_once_with() (*unittest.mock.Mock* 메서드), 1835
assert_called_with() (*unittest.mock.Mock* 메서드), 1835
assert_has_awaits() (*unittest.mock.AsyncMock* 메서드), 1846
assert_has_calls() (*unittest.mock.Mock* 메서드), 1836
assert_never() (*typing* 모듈), 1758
assert_not_awaited() (*unittest.mock.AsyncMock* 메서드), 1846
assert_not_called() (*unittest.mock.Mock* 메서드), 1836
assert_python_failure() (*test.support.script_helper* 모듈), 1906
assert_python_ok() (*test.support.script_helper* 모듈), 1906
assert_type() (*typing* 모듈), 1758
assertAlmostEqual() (*unittest.TestCase* 메서드), 1815
assertCountEqual() (*unittest.TestCase* 메서드), 1816
assertDictEqual() (*unittest.TestCase* 메서드), 1817
assertEndsWith() (*unittest.TestCase* 메서드), 1816
assertEqual() (*unittest.TestCase* 메서드), 1811
assertFalse() (*unittest.TestCase* 메서드), 1811
assertGreater() (*unittest.TestCase* 메서드), 1816
assertGreaterEqual() (*unittest.TestCase* 메서드), 1816
assertHasAttr() (*unittest.TestCase* 메서드), 1816
assertIn() (*unittest.TestCase* 메서드), 1812
assertInBytecode() (*test.support.bytecode_helper.BytecodeTestCase* 메서드), 1907
AssertionError, 113
assertIs() (*unittest.TestCase* 메서드), 1812
assertIsInstance() (*unittest.TestCase* 메서드), 1812
assertIsNone() (*unittest.TestCase* 메서드), 1812
assertIsNot() (*unittest.TestCase* 메서드), 1812
assertIsNotNone() (*unittest.TestCase* 메서드), 1812
assertIsSubclass() (*unittest.TestCase* 메서드), 1812
assertLess() (*unittest.TestCase* 메서드), 1816
assertLessEqual() (*unittest.TestCase* 메서드), 1816
assertListEqual() (*unittest.TestCase* 메서드), 1817
assertLogs() (*unittest.TestCase* 메서드), 1814
assertMultiLineEqual() (*unittest.TestCase* 메서드), 1817
assertNoLogs() (*unittest.TestCase* 메서드), 1815
assertNotAlmostEqual() (*unittest.TestCase* 메서드), 1815
assertNotEndsWith() (*unittest.TestCase* 메서드), 1816
assertNotEqual() (*unittest.TestCase* 메서드), 1811
assertNotHasAttr() (*unittest.TestCase* 메서드), 1816
assertNotIn() (*unittest.TestCase* 메서드), 1812
assertNotInBytecode() (*test.support.bytecode_helper.BytecodeTestCase* 메서드), 1907
assertNotIsInstance() (*unittest.TestCase* 메서드), 1812
assertNotIsSubclass() (*unittest.TestCase* 메서드), 1812
assertNotRegex() (*unittest.TestCase* 메서드), 1816
assertNotStartsWith() (*unittest.TestCase* 메서드), 1816
assertRaises() (*unittest.TestCase* 메서드), 1812
assertRaisesRegex() (*unittest.TestCase* 메서드), 1813
assertRegex() (*unittest.TestCase* 메서드), 1816
assertSequenceEqual() (*unittest.TestCase* 메서드), 1817
assertSetEqual() (*unittest.TestCase* 메서드), 1817
assertStartsWith() (*unittest.TestCase* 메서드), 1816
assertTrue() (*unittest.TestCase* 메서드), 1811
assertTupleEqual() (*unittest.TestCase* 메서드), 1817
assertWarns() (*unittest.TestCase* 메서드), 1813
assertWarnsRegex() (*unittest.TestCase* 메서드), 1814
Assign (*ast* 클래스), 2176
assignment
    slice, 49
    subscript, 49
assume_default_colors() (*curses* 모듈), 958
ast
    module, 2161
AST (*ast* 클래스), 2164
ast 명령줄 옵션
    -a, 2200
    --feature-version, 2200
    -h, 2200
    --help, 2200
    -i, 2200

--include-attributes, 2200
--indent, 2200
-m, 2200
--mode, 2200
--no-type-comments, 2200
-O, 2200
--optimize, 2200
--show-empty, 2200
astimezone() (*datetime.datetime* 메서드), 234
astuple() (*dataclasses* 모듈), 2038
AsyncContextDecorator (*contextlib* 클래스), 2050
AsyncContextManager (*typing* 클래스), 1772
asynccontextmanager() (*contextlib* 모듈), 2045
AsyncExitStack (*contextlib* 클래스), 2052
AsyncFor (*ast* 클래스), 2195
AsyncFunctionDef (*ast* 클래스), 2194
AsyncGenerator (*collections.abc* 클래스), 292
AsyncGenerator (*typing* 클래스), 1770
AsyncGeneratorType (*types* 모듈), 313
asynchat
module, 2293
asynchronous context manager (비동기 컨텍스트 관리자), **2302**
asynchronous generator (비동기 제너레이터), **2302**
asynchronous generator iterator (비동기 제너레이터 이터레이터), **2302**
asynchronous iterable (비동기 이터러블), **2302**
asynchronous iterator (비동기 이터레이터), **2302**
asyncio
module, 1109
asyncio.subprocess.DEVNULL (내장 변수), 1148
asyncio.subprocess.PIPE (내장 변수), 1148
asyncio.subprocess.Process (내장 클래스), 1148
asyncio.subprocess.STDOUT (내장 변수), 1148
AsyncIterable (*collections.abc* 클래스), 291
AsyncIterable (*typing* 클래스), 1770
AsyncIterator (*collections.abc* 클래스), 292
AsyncIterator (*typing* 클래스), 1770
AsyncMock (*unittest.mock* 클래스), 1844
asyncore
module, 2293
AsyncResult (*multiprocessing.pool* 클래스), 1043
asyncSetUp() (*unittest.IsolatedAsyncioTestCase* 메서드), 1819
asyncTearDown() (*unittest.IsolatedAsyncioTestCase* 메서드), 1820
AsyncWith (*ast* 클래스), 2195
AT (*token* 모듈), 2209
at_eof() (*asyncio.StreamReader* 메서드), 1136
atan() (*cmath* 모듈), 362
atan() (*math* 모듈), 357
atan2() (*math* 모듈), 357
atanh() (*cmath* 모듈), 362
atanh() (*math* 모듈), 358
ATEQUAL (*token* 모듈), 2209
atexit
module, 2063
atexit (*weakref.finalize*의 속성), 307
atof() (*locale* 모듈), 1623
atoi() (*locale* 모듈), 1623
attach() (*email.message.Message* 메서드), 1343
attach_mock() (*unittest.mock.Mock* 메서드), 1837
attached thread state, **2302**
attempted (*doctest.TestResults*의 속성), 1794
AttlistDeclHandler() (*xml.parsers.expat.xmlparser* 메서드), 1458
attrgetter() (*operator* 모듈), 454
attrib (*xml.etree.ElementTree.Element*의 속성), 1419
Attribute (*ast* 클래스), 2173
attribute (어트리뷰트), **2303**
AttributeError, 113
attributes (*xml.dom.Node*의 속성), 1429
AttributesImpl (*xml.sax.xmlreader* 클래스), 1451
AttributesNSImpl (*xml.sax.xmlreader* 클래스), 1451
attroff() (*curses.window* 메서드), 965
attron() (*curses.window* 메서드), 965
attrset() (*curses.window* 메서드), 965
audioop
module, 2294
audit() (*sys* 모듈), 1982
AugAssign (*ast* 클래스), 2177
AUGUST (*calendar* 모듈), 266
auth() (*ftplib.FTP_TLS* 메서드), 1523
auth() (*smtplib.SMTP* 메서드), 1538
authenticate() (*imaplib.IMAP4* 메서드), 1529
AuthenticationError, 1023
authenticators() (*netrc.netrc* 메서드), 670
authkey (*multiprocessing.Process*의 속성), 1021
auto (*enum* 클래스), 341
autocommit (*sqlite3.Connection*의 속성), 566
autorange() (*timeit.Timer* 메서드), 1948
available_timezones() (*zoneinfo* 모듈), 260
avoids_symlink_attacks (*shutil.rmtree*의 속성), 515
Await (*ast* 클래스), 2194
await_args (*unittest.mock.AsyncMock*의 속성), 1847
await_args_list (*unittest.mock.AsyncMock*의 속성), 1847
await_count (*unittest.mock.AsyncMock*의 속성), 1846
Awaitable (*collections.abc* 클래스), 291
Awaitable (*typing* 클래스), 1771
awaitable (어웨이터블), **2303**

B

-b
compileall 명령줄 옵션, 2219
http.server 명령줄 옵션, 1562
unittest 명령줄 옵션, 1803
b2a_base64() (*binascii* 모듈), 1398
b2a_hex() (*binascii* 모듈), 1399
b2a_qp() (*binascii* 모듈), 1399
b2a_uu() (*binascii* 모듈), 1398

b16decode() (*base64* 모듈), 1396
b16encode() (*base64* 모듈), 1395
b32decode() (*base64* 모듈), 1395
b32encode() (*base64* 모듈), 1395
b32hexdecode() (*base64* 모듈), 1395
b32hexencode() (*base64* 모듈), 1395
b64decode() (*base64* 모듈), 1395
b64encode() (*base64* 모듈), 1394
b85decode() (*base64* 모듈), 1396
b85encode() (*base64* 모듈), 1396
Babyl (*mailbox* 클래스), 1379
BabylMessage (*mailbox* 클래스), 1385
back() (*turtle* 모듈), 1685
backend (*readline* 모듈), 185
backslashreplace
    에러 처리기의 이름, 202
backslashreplace_errors() (*codecs* 모듈), 203
backup() (*sqlite3.Connection* 메서드), 564
backward() (*turtle* 모듈), 1685
BadGzipFile, 598
BadOptionError, 953
BadStatusLine, 1512
BadZipFile, 612
BadZipfile, 612
Barrier (*asyncio* 클래스), 1145
Barrier (*multiprocessing* 클래스), 1030
Barrier (*threading* 클래스), 1011
Barrier() (*multiprocessing.managers.SyncManager* 메서드), 1036
base64
    module, 1394
    모듈, 1398
    인코딩, 1394
base_exec_prefix (*sys* 모듈), 1982
base_prefix (*sys* 모듈), 1982
BaseCGIHandler (*wsgiref.handlers* 클래스), 1473
BaseCookie (*http.cookies* 클래스), 1564
BaseException, 112
BaseExceptionGroup, 121
BaseHandler (*urllib.request* 클래스), 1482
BaseHandler (*wsgiref.handlers* 클래스), 1474
BaseHeader (*email.headerregistry* 클래스), 1327
BaseHTTPRequestHandler (*http.server* 클래스), 1557
BaseManager (*multiprocessing.managers* 클래스), 1035
basename() (*os.path* 모듈), 487
BaseProtocol (*asyncio* 클래스), 1189
BaseProxy (*multiprocessing.managers* 클래스), 1040
BaseRequestHandler (*socketserver* 클래스), 1551
BaseRotatingHandler (*logging.handlers* 클래스), 816
BaseSelector (*selectors* 클래스), 1285
BaseServer (*socketserver* 클래스), 1549
BaseTransport (*asyncio* 클래스), 1185
basicConfig() (*logging* 모듈), 799
BasicContext (*decimal* 모듈), 377
BasicInterpolation (*configparser* 클래스), 653
batched() (*itertools* 모듈), 425
baudrate() (*curses* 모듈), 958
bbox() (*tkinter.ttk.Treeview* 메서드), 1658
BDADDR_ANY (*socket* 모듈), 1220
BDADDR_BREDR (*socket* 모듈), 1220
BDADDR_LE_PUBLIC (*socket* 모듈), 1220
BDADDR_LE_RANDOM (*socket* 모듈), 1220
BDADDR_LOCAL (*socket* 모듈), 1220
bdb
    module, 1919
    모듈, 1928
Bdb (*bdb* 클래스), 1920
BdbQuit, 1919
BDFL, **2303**
beep() (*curses* 모듈), 958
Beep() (*winsound* 모듈), 2258
begin_fill() (*turtle* 모듈), 1695
begin_poly() (*turtle* 모듈), 1700
BEL (*curses.ascii* 모듈), 984
below() (*curses.panel.Panel* 메서드), 987
BELOW_NORMAL_PRIORITY_CLASS (*subprocess* 모듈), 1090
--best
    gzip 명령줄 옵션, 601
betavariate() (*random* 모듈), 400
bgcolor() (*turtle* 모듈), 1702
bgpic() (*turtle* 모듈), 1702
bidirectional() (*unicodedata* 모듈), 182
bigaddrspacetest() (*test.support* 모듈), 1902
BigEndianStructure (*ctypes* 클래스), 875
BigEndianUnion (*ctypes* 클래스), 874
bigmemtest() (*test.support* 모듈), 1902
bin()
    built-in function, 9
binary
    literals, 39
Binary (*xmlrpc.client* 클래스), 1579
binary file (바이너리 파일), **2303**
BINARY_OP (*opcode*), 2230
BINARY_SLICE (*opcode*), 2230
BinaryIO (*typing* 클래스), 1757
binascii
    module, 1398
--bind
    http.server 명령줄 옵션, 1562
bind (위젯), 1638
bind() (*inspect.Signature* 메서드), 2088
bind() (*socket.socket* 메서드), 1230
bind_partial() (*inspect.Signature* 메서드), 2088
bind_port() (*test.support.socket_helper* 모듈), 1905
bind_textdomain_codeset() (*locale* 모듈), 1625
bind_unix_socket() (*test.support.socket_helper* 모듈), 1906
bindtextdomain() (*gettext* 모듈), 1609
bindtextdomain() (*locale* 모듈), 1625
binomialvariate() (*random* 모듈), 400
BinOp (*ast* 클래스), 2171
bisect

module, 297
bisect() (*bisect* 모듈), 298
bisect_left() (*bisect* 모듈), 298
bisect_right() (*bisect* 모듈), 298
bit_count() (*int* 메서드), 41
bit_length() (*int* 메서드), 40
bit_offset (*ctypes.CField*의 속성), 877
bit_size (*ctypes.CField*의 속성), 877
BitAnd (*ast* 클래스), 2171
BitOr (*ast* 클래스), 2171
bits_per_digit (*sys.int_info*의 속성), 1996
bitwise
operations, 40
BitXor (*ast* 클래스), 2171
bk() (*turtle* 모듈), 1685
bkgd() (*curses.window* 메서드), 965
bkgdset() (*curses.window* 메서드), 965
blake2b() (*hashlib* 모듈), 678
blake2b, blake2s, 678
blake2b.MAX_DIGEST_SIZE (*hashlib* 모듈), 679
blake2b.MAX_KEY_SIZE (*hashlib* 모듈), 679
blake2b.PERSON_SIZE (*hashlib* 모듈), 679
blake2b.SALT_SIZE (*hashlib* 모듈), 679
blake2s() (*hashlib* 모듈), 678
blake2s.MAX_DIGEST_SIZE (*hashlib* 모듈), 679
blake2s.MAX_KEY_SIZE (*hashlib* 모듈), 679
blake2s.PERSON_SIZE (*hashlib* 모듈), 679
blake2s.SALT_SIZE (*hashlib* 모듈), 679
BLKTYPE (*tarfile* 모듈), 626
Blob (*sqlite3* 클래스), 571
blobopen() (*sqlite3.Connection* 메서드), 557
block_on_close (*socketserver.ThreadingMixIn*의 속성), 1548
block_size (*hmac.HMAC*의 속성), 686
blocked_domains() (*http.cookiejar.DefaultCookiePolicy* 메서드), 1572
BlockingIOError, 118, 759
blocksize (*http.client.HTTPConnection*의 속성), 1514
body() (*tkinter.simpledialog.Dialog* 메서드), 1642
body_encode() (*email.charset.Charset* 메서드), 1355
body_encoding (*email.charset.Charset*의 속성), 1355
body_line_iterator() (*email.iterators* 모듈), 1359
BOLD (*tkinter.font* 모듈), 1640
BOM (*codecs* 모듈), 201
BOM_BE (*codecs* 모듈), 201
BOM_LE (*codecs* 모듈), 201
BOM_UTF8 (*codecs* 모듈), 201
BOM_UTF16 (*codecs* 모듈), 201
BOM_UTF16_BE (*codecs* 모듈), 201
BOM_UTF16_LE (*codecs* 모듈), 201
BOM_UTF32 (*codecs* 모듈), 201
BOM_UTF32_BE (*codecs* 모듈), 201
BOM_UTF32_LE (*codecs* 모듈), 201
bool (내장 클래스), 9
Boolean
object, 38
operations, 37
type, 9
values, 46
BOOLEAN_STATES (*configparser.ConfigParser*의 속성), 658
BooleanOptionalAction (*argparse* 클래스), 900
BoolOp (*ast* 클래스), 2171
bootstrap() (*ensurepip* 모듈), 1966
border() (*curses.window* 메서드), 965
borrowed reference (빌린 참조), **2303**
bottom() (*curses.panel.Panel* 메서드), 988
bottom_panel() (*curses.panel* 모듈), 987
BoundArguments (*inspect* 클래스), 2091
BoundaryError, 1325
BoundedSemaphore (*asyncio* 클래스), 1144
BoundedSemaphore (*multiprocessing* 클래스), 1030
BoundedSemaphore (*threading* 클래스), 1009
BoundedSemaphore() (*multiprocessing.managers.SyncManager* 메서드), 1036
bounds() (*compression.zstd.CompressionParameter* 메서드), 589
bounds() (*compression.zstd.DecompressionParameter* 메서드), 592
box() (*curses.window* 메서드), 966
bpbynumber (*bdb.Breakpoint*의 속성), 1920
bpformat() (*bdb.Breakpoint* 메서드), 1919
bplist (*bdb.Breakpoint*의 속성), 1920
bpprint() (*bdb.Breakpoint* 메서드), 1920
BRANCH_LEFT (*monitoring event*), 2009
BRANCH_RIGHT (*monitoring event*), 2009
Break (*ast* 클래스), 2181
break (*pdb command*), 1933
break_anywhere() (*bdb.Bdb* 메서드), 1922
break_here() (*bdb.Bdb* 메서드), 1922
break_long_words (*textwrap.TextWrapper*의 속성), 180
break_on_hyphens (*textwrap.TextWrapper*의 속성), 180
Breakpoint (*bdb* 클래스), 1919
breakpoint()
built-in function, 9
breakpointhook() (*sys* 모듈), 1984
broadcast_address (*ipaddress.IPv4Network*의 속성), 1596
broadcast_address (*ipaddress.IPv6Network*의 속성), 1599
broken (*asyncio.Barrier*의 속성), 1146
broken (*threading.Barrier*의 속성), 1011
BrokenBarrierError, 1012, 1146
BrokenExecutor, 1071
BrokenInterpreterPool, 1072
BrokenPipeError, 119
BrokenProcessPool, 1072
BrokenThreadPool, 1072
BROWSER, 1465, 1466
BS (*curses.ascii* 모듈), 984
BsdDbShelf (*shelve* 클래스), 542
btlazy2 (*compression.zstd.Strategy*의 속성), 592
btopt (*compression.zstd.Strategy*의 속성), 593

BTPROTO_HCI (*socket* 모듈), 1220
BTPROTO_L2CAP (*socket* 모듈), 1220
BTPROTO_RFCOMM (*socket* 모듈), 1220
BTPROTO_SCO (*socket* 모듈), 1220
btultra (*compression.zstd.Strategy*의 속성), 593
btultra2 (*compression.zstd.Strategy*의 속성), 593
buf (*multiprocessing.shared_memory.SharedMemory*의 속성), 1059
--buffer
    unittest 명령줄 옵션, 1803
Buffer (*collections.abc* 클래스), 292
buffer (*io.TextIOBase*의 속성), 767
buffer (*unittest.TestResult*의 속성), 1825
buffer protocol
    binary sequence types, 67
    str (*built-in class*), 54
buffer size, I/O, 25
buffer_info() (*array.array* 메서드), 302
buffer_size (*xml.parsers.expat.xmlparser*의 속성), 1457
buffer_text (*xml.parsers.expat.xmlparser*의 속성), 1457
buffer_updated() (*asyncio.BufferedProtocol* 메서드), 1191
buffer_used (*xml.parsers.expat.xmlparser*의 속성), 1457
BufferedIOBase (*io* 클래스), 762
BufferedProtocol (*asyncio* 클래스), 1189
BufferedRandom (*io* 클래스), 766
BufferedReader (*io* 클래스), 765
BufferedRWPair (*io* 클래스), 766
BufferedWriter (*io* 클래스), 766
BufferError, 113
BufferFlags (*inspect* 클래스), 2099
BufferingFormatter (*logging* 클래스), 793
BufferingHandler (*logging.handlers* 클래스), 824
BufferTooShort, 1023
BUILD_INTERPOLATION (*opcode*), 2235
BUILD_LIST (*opcode*), 2235
BUILD_MAP (*opcode*), 2235
build_opener() (*urllib.request* 모듈), 1480
BUILD_SET (*opcode*), 2235
BUILD_SLICE (*opcode*), 2241
BUILD_STRING (*opcode*), 2235
BUILD_TEMPLATE (*opcode*), 2234
BUILD_TUPLE (*opcode*), 2235
built-in
    types, 37
built-in function
    __import__(), 33
    abs(), 8
    aiter(), 8
    all(), 8
    anext(), 8
    any(), 9
    ascii(), 9
    bin(), 9
    breakpoint(), 9
    callable(), 10
    chr(), 10
    classmethod(), 10
    compile, 106
    compile(), 11
    complex, 39
    delattr(), 13
    dir(), 13
    divmod(), 14
    enumerate(), 14
    eval, 106
    eval(), 14
    exec, 15, 106
    exec(), 15
    filter(), 16
    float, 39
    format(), 17
    getattr(), 17
    globals(), 18
    hasattr(), 18
    hash, 49
    hash(), 18
    help(), 18
    hex(), 18
    id(), 19
    input(), 19
    int, 39
    isinstance(), 20
    issubclass(), 20
    iter(), 20
    len, 47, 93
    len(), 21
    locals(), 21
    map(), 21
    max, 47
    max(), 22
    min, 47
    min(), 22
    multiprocessing.Manager(), 1035
    next(), 22
    oct(), 22
    open(), 23
    ord(), 25
    pow(), 25
    print(), 26
    property.deleter(), 27
    property.getter(), 27
    property.setter(), 27
    repr(), 28
    reversed(), 28
    round(), 28
    setattr(), 28
    slice, 2241
    sorted(), 29
    staticmethod(), 29
    sum(), 30
    type, 106
    vars(), 32

zip(), 32
builtin_module_names (*sys* 모듈), 1983
BuiltinFunctionType (*types* 모듈), 314
BuiltinImporter (*importlib.machinery* 클래스), 2135
BuiltinMethodType (*types* 모듈), 314
builtins
module, 33, 2019
burst() (*imaplib.IMAP4.Idler* 메서드), 1530
busy_retry() (*test.support* 모듈), 1898
BUTTON_ALT (*curses* 모듈), 981
BUTTON_CTRL (*curses* 모듈), 981
BUTTON_SHIFT (*curses* 모듈), 981
buttonbox() (*tkinter.simpledialog.Dialog* 메서드), 1642
BUTTONn_CLICKED (*curses* 모듈), 981
BUTTONn_DOUBLE_CLICKED (*curses* 모듈), 981
BUTTONn_PRESSED (*curses* 모듈), 981
BUTTONn_RELEASED (*curses* 모듈), 981
BUTTONn_TRIPLE_CLICKED (*curses* 모듈), 981
bye() (*turtle* 모듈), 1708
byref() (*ctypes* 모듈), 867
byte_offset (*ctypes.CField*의 속성), 877
byte_size (*ctypes.CField*의 속성), 877
bytearray
formatting, 81
interpolation, 81
methods, 70
object, 49, 67, 69
bytearray (내장 클래스), 69
Bytecode (*dis* 클래스), 2224
bytecode (바이트 코드), **2303**
BYTECODE_SUFFIXES (*importlib.machinery* 모듈), 2135
Bytecode.codeobj (*dis* 모듈), 2224
Bytecode.first_line (*dis* 모듈), 2224
BytecodeTestCase (*test.support.bytecode_helper* 클래스), 1907
byteorder (*sys* 모듈), 1983
bytes
formatting, 81
interpolation, 81
methods, 70
object, 67, 68
str (*built-in class*), 54
bytes (*uuid.UUID*의 속성), 1542
bytes (내장 클래스), 68
bytes-like object (바이트열류 객체), **2303**
bytes_le (*uuid.UUID*의 속성), 1542
bytes_warning (*sys.flags*의 속성), 1988
BytesFeedParser (*email.parser* 클래스), 1312
BytesGenerator (*email.generator* 클래스), 1316
BytesHeaderParser (*email.parser* 클래스), 1314
BytesIO (*io* 클래스), 765
BytesParser (*email.parser* 클래스), 1313
ByteString (*collections.abc* 클래스), 290
ByteString (*typing* 클래스), 1769
byteswap() (*array.array* 메서드), 302
BytesWarning, 121
bz2
module, 601
BZ2Compressor (*bz2* 클래스), 603
BZ2Decompressor (*bz2* 클래스), 604
BZ2File (*bz2* 클래스), 602

C

C
language, 38
구조체, 191
-C
dis 명령줄 옵션, 2223
trace 명령줄 옵션, 1953
uuid 명령줄 옵션, 1546
-c
calendar 명령줄 옵션, 269
idle 명령줄 옵션, 1673
pdb 명령줄 옵션, 1929
random 명령줄 옵션, 406
tarfile 명령줄 옵션, 637
trace 명령줄 옵션, 1952
unittest 명령줄 옵션, 1803
zipapp 명령줄 옵션, 1977
zipfile 명령줄 옵션, 622
C14NWriterTarget (*xml.etree.ElementTree* 클래스), 1424
c_bool (*ctypes* 클래스), 874
c_byte (*ctypes* 클래스), 872
c_char (*ctypes* 클래스), 872
c_char_p (*ctypes* 클래스), 872
C_CONTIGUOUS (*inspect.BufferFlags*의 속성), 2099
c_contiguous (*memoryview*의 속성), 91
c_double (*ctypes* 클래스), 872
c_double_complex (*ctypes* 클래스), 872
c_float (*ctypes* 클래스), 872
c_float_complex (*ctypes* 클래스), 872
c_int (*ctypes* 클래스), 873
c_int8 (*ctypes* 클래스), 873
c_int16 (*ctypes* 클래스), 873
c_int32 (*ctypes* 클래스), 873
c_int64 (*ctypes* 클래스), 873
c_long (*ctypes* 클래스), 873
c_longdouble (*ctypes* 클래스), 872
c_longdouble_complex (*ctypes* 클래스), 873
c_longlong (*ctypes* 클래스), 873
C_RAISE (*monitoring event*), 2009
C_RETURN (*monitoring event*), 2009
c_short (*ctypes* 클래스), 873
c_size_t (*ctypes* 클래스), 873
c_ssize_t (*ctypes* 클래스), 873
c_time_t (*ctypes* 클래스), 873
c_ubyte (*ctypes* 클래스), 873
c_uint (*ctypes* 클래스), 873
c_uint8 (*ctypes* 클래스), 873
c_uint16 (*ctypes* 클래스), 873
c_uint32 (*ctypes* 클래스), 874
c_uint64 (*ctypes* 클래스), 874

c_ulong (*ctypes* 클래스), 874
c_ulonglong (*ctypes* 클래스), 874
c_ushort (*ctypes* 클래스), 874
c_void_p (*ctypes* 클래스), 874
c_wchar (*ctypes* 클래스), 874
c_wchar_p (*ctypes* 클래스), 874
CACHE (*opcode*), 2229
cache() (*functools* 모듈), 440
cache_clear() (*functools.lru_cache* 메서드), 442
cache_from_source() (*importlib.util* 모듈), 2140
cache_info() (*functools.lru_cache* 메서드), 442
cached (*importlib.machinery.ModuleSpec* 의 속성), 2139
cached_property() (*functools* 모듈), 441
CacheFTPHandler (*urllib.request* 클래스), 1484
calcobjsize() (*test.support* 모듈), 1901
calcsize() (*struct* 모듈), 192
calcvobjsize() (*test.support* 모듈), 1901
calendar
    module, 261
Calendar (*calendar* 클래스), 261
calendar 명령줄 옵션
    -c, 269
    --css, 269
    -e, 268
    --encoding, 268
    -f, 268
    --first-weekday, 268
    -h, 268
    --help, 268
    -L, 268
    -l, 268
    --lines, 268
    --locale, 268
    -m, 269
    month, 268
    --months, 269
    -s, 269
    --spacing, 269
    -t, 268
    --type, 268
    -w, 268
    --width, 268
    year, 268
calendar() (*calendar* 모듈), 265
Call (*ast* 클래스), 2172
CALL (*monitoring event*), 2009
CALL (*opcode*), 2239
call() (*concurrent.interpreters.Interpreter* 메서드), 1075
call() (*operator* 모듈), 454
call() (*subprocess* 모듈), 1091
call() (*unittest.mock* 모듈), 1864
call_annotate_function() (*annotationlib* 모듈), 2103
call_args (*unittest.mock.Mock*의 속성), 1840
call_args_list (*unittest.mock.Mock*의 속성), 1840
call_at() (*asyncio.loop* 메서드), 1160
call_count (*unittest.mock.Mock*의 속성), 1838
call_evaluate_function() (*annotationlib* 모듈), 2103
call_exception_handler() (*asyncio.loop* 메서드), 1173
CALL_FUNCTION_EX (*opcode*), 2240
call_in_thread() (*concurrent.interpreters.Interpreter* 메서드), 1075
CALL_INTRINSIC_1 (*opcode*), 2242
CALL_INTRINSIC_2 (*opcode*), 2243
CALL_KW (*opcode*), 2240
call_later() (*asyncio.loop* 메서드), 1159
call_list() (*unittest.mock.call* 메서드), 1864
call_soon() (*asyncio.loop* 메서드), 1159
call_soon_threadsafe() (*asyncio.loop* 메서드), 1159
call_tracing() (*sys* 모듈), 1983
Callable (*collections.abc* 클래스), 290
Callable (*typing* 모듈), 1771
callable (콜러블), **2303**
callable()
    built-in function, 10
CallableProxyType (*weakref* 모듈), 308
callback (*optparse.Option*의 속성), 941
callback (콜백), **2304**
callback() (*contextlib.ExitStack* 메서드), 2052
callback_args (*optparse.Option*의 속성), 941
callback_kwargs (*optparse.Option*의 속성), 941
callbacks (*gc* 모듈), 2080
called (*unittest.mock.Mock*의 속성), 1838
CalledProcessError, 1079
CAN (*curses.ascii* 모듈), 985
CAN_BCM (*socket* 모듈), 1218
can_change_color() (*curses* 모듈), 958
can_fetch() (*urllib.robotparser.RobotFileParser* 메서드), 1505
CAN_ISOTP (*socket* 모듈), 1218
CAN_J1939 (*socket* 모듈), 1219
CAN_RAW_FD_FRAMES (*socket* 모듈), 1218
CAN_RAW_JOIN_FILTERS (*socket* 모듈), 1218
can_symlink() (*test.support.os_helper* 모듈), 1909
can_write_eof() (*asyncio.StreamWriter* 메서드), 1136
can_write_eof() (*asyncio.WriteTransport* 메서드), 1187
can_xattr() (*test.support.os_helper* 모듈), 1909
CANCEL (*tkinter.messagebox* 모듈), 1646
cancel() (*asyncio.Future* 메서드), 1183
cancel() (*asyncio.Handle* 메서드), 1175
cancel() (*asyncio.Task* 메서드), 1131
cancel() (*concurrent.futures.Future* 메서드), 1069
cancel() (*sched.scheduler* 메서드), 1097
cancel() (*threading.Timer* 메서드), 1010
cancel() (*tkinter.dnd.DndHandler* 메서드), 1647
cancel_command() (*tkinter.filedialog.FileDialog* 메서드), 1643
cancel_dump_traceback_later() (*faulthandler* 모듈), 1927

cancel_join_thread() (*multiprocessing.Queue* 메서드), 1025
cancelled() (*asyncio.Future* 메서드), 1182
cancelled() (*asyncio.Handle* 메서드), 1175
cancelled() (*asyncio.Task* 메서드), 1131
cancelled() (*concurrent.futures.Future* 메서드), 1070
CancelledError, 1071, 1153
cancelling() (*asyncio.Task* 메서드), 1132
CannotSendHeader, 1511
CannotSendRequest, 1511
canonic() (*bdb.Bdb* 메서드), 1920
canonical() (*decimal.Context* 메서드), 379
canonical() (*decimal.Decimal* 메서드), 370
canonicalize() (*xml.etree.ElementTree* 모듈), 1414
capa() (*poplib.POP3* 메서드), 1525
capitalize() (*bytearray* 메서드), 77
capitalize() (*bytes* 메서드), 77
capitalize() (*str* 메서드), 55
capitals (*decimal.Context*의 속성), 378
CapsuleType (*types* 클래스), 316
capture_call_graph() (*asyncio* 모듈), 1155
captured_stderr() (*test.support* 모듈), 1900
captured_stdin() (*test.support* 모듈), 1900
captured_stdout() (*test.support* 모듈), 1900
captureWarnings() (*logging* 모듈), 801
capwords() (*string* 모듈), 136
casefold() (*str* 메서드), 55
cast() (*ctypes* 모듈), 868
cast() (*memoryview* 메서드), 87
cast() (*typing* 모듈), 1758
--catch
    unittest 명령줄 옵션, 1803
catch_threading_exception() (*test.support.threading_helper* 모듈), 1908
catch_unraisable_exception() (*test.support* 모듈), 1903
catch_warnings (*warnings* 클래스), 2032
category() (*unicodedata* 모듈), 181
cbreak() (*curses* 모듈), 958
cbrt() (*math* 모듈), 355
ccc() (*ftplib.FTP_TLS* 메서드), 1523
C-contiguous, 2304
cdf() (*statistics.NormalDist* 메서드), 418
CDLL (*ctypes* 클래스), 861
ceil() (*in module math*), 39
ceil() (*math* 모듈), 353
CellType (*types* 모듈), 314
center() (*bytearray* 메서드), 74
center() (*bytes* 메서드), 74
center() (*str* 메서드), 55
CERT_NONE (*ssl* 모듈), 1247
CERT_OPTIONAL (*ssl* 모듈), 1247
CERT_REQUIRED (*ssl* 모듈), 1247
cert_store_stats() (*ssl.SSLContext* 메서드), 1259
cert_time_to_seconds() (*ssl* 모듈), 1246
CertificateError, 1245
certificates, 1267
CField (*ctypes* 클래스), 876
cfmakecbreak() (*tty* 모듈), 2273
cfmakeraw() (*tty* 모듈), 2272
CFUNCTYPE() (*ctypes* 모듈), 865
cget() (*tkinter.font.Font* 메서드), 1641
cgi
    module, 2294
--cgi
    http.server 명령줄 옵션, 1562
cgi_directories (*http.server.CGIHTTPRequestHandler*의 속성), 1561
CGIHandler (*wsgiref.handlers* 클래스), 1473
CGIHTTPRequestHandler (*http.server* 클래스), 1561
cgitb
    module, 2294
CGIXMLRPCRequestHandler (*xmlrpc.server* 클래스), 1584
chain() (*itertools* 모듈), 426
chain_log (*compression.zstd.CompressionParameter*의 속성), 590
chaining
    comparisons, 38
ChainMap (*collections* 클래스), 269
ChainMap (*typing* 클래스), 1768
change_cwd() (*test.support.os_helper* 모듈), 1909
CHANNEL_BINDING_TYPES (*ssl* 모듈), 1252
CHAR_MAX (*locale* 모듈), 1623
CharacterDataHandler() (*xml.parsers.expat.xmlparser* 메서드), 1458
characters() (*xml.sax.handler.ContentHandler* 메서드), 1447
characters_written (*BlockingIOError*의 속성), 119
charmap_build() (*codecs* 모듈), 199
Charset (*email.charset* 클래스), 1354
charset() (*gettext.NullTranslations* 메서드), 1612
chdir() (*contextlib* 모듈), 2049
chdir() (*os* 모듈), 713
check (*lzma.LZMADecompressor*의 속성), 609
check() (*imaplib.IMAP4* 메서드), 1529
check() (*tabnanny* 모듈), 2214
check__all__() (*test.support* 모듈), 1904
check_call() (*subprocess* 모듈), 1091
check_disallow_instantiation() (*test.support* 모듈), 1905
CHECK_EG_MATCH (*opcode*), 2232
CHECK_EXC_MATCH (*opcode*), 2232
check_free_after_iterating() (*test.support* 모듈), 1904
check_hostname (*ssl.SSLContext*의 속성), 1264
check_impl_detail() (*test.support* 모듈), 1899
check_no_resource_warning() (*test.support.warnings_helper* 모듈), 1912
check_output() (*doctest.OutputChecker* 메서드), 1796
check_output() (*subprocess* 모듈), 1091
check_returncode() (*subprocess.CompletedProcess* 메서드), 1078
check_syntax_error() (*test.support* 모듈), 1903

check_syntax_warning() (*test.support.warnings_helper* 모듈), 1912
check_unused_args() (*string.Formatter* 메서드), 127
check_warnings() (*test.support.warnings_helper* 모듈), 1912
checkcache() (*linecache* 모듈), 510
CHECKED_HASH (*py_compile.PycInvalidationMode*의 속성), 2218
checkfuncname() (*bdb* 모듈), 1924
checksizeof() (*test.support* 모듈), 1901
checksum_flag (*compression.zstd.CompressionParameter* 의 속성), 591
chflags() (*os* 모듈), 713
chgat() (*curses.window* 메서드), 966
childNodes (*xml.dom.Node*의 속성), 1429
ChildProcessError, 119
children (*pyclbr.Class*의 속성), 2216
children (*pyclbr.Function*의 속성), 2216
children (*tkinter.Tk*의 속성), 1629
chksum (*tarfile.TarInfo*의 속성), 632
chmod() (*os* 모듈), 714
chmod() (*pathlib.Path* 메서드), 483
--choice
    random 명령줄 옵션, 406
choice() (*random* 모듈), 399
choice() (*secrets* 모듈), 687
choices (*optparse.Option*의 속성), 941
choices() (*random* 모듈), 399
Chooser (*tkinter.colorchooser* 클래스), 1640
chown() (*os* 모듈), 715
chown() (*shutil* 모듈), 516
chr()
    built-in function, 10
chroot() (*os* 모듈), 715
CHRTYPE (*tarfile* 모듈), 626
chunk
    module, 2294
cipher() (*ssl.SSLSocket* 메서드), 1256
circle() (*turtle* 모듈), 1688
CIRCUMFLEX (*token* 모듈), 2208
CIRCUMFLEXEQUAL (*token* 모듈), 2209
clamp (*decimal.Context*의 속성), 378
Clamped (*decimal* 클래스), 384
Class (*pyclbr* 클래스), 2216
Class (*symtable* 클래스), 2202
CLASS (*symtable.SymbolTableType*의 속성), 2201
class (클래스), **2304**
class variable (클래스 변수), **2304**
ClassDef (*ast* 클래스), 2194
classmethod()
    built-in function, 10
ClassMethodDescriptorType (*types* 모듈), 314
ClassVar (*typing* 모듈), 1735
CLD_CONTINUED (*os* 모듈), 750
CLD_DUMPED (*os* 모듈), 750
CLD_EXITED (*os* 모듈), 750
CLD_KILLED (*os* 모듈), 750
CLD_STOPPED (*os* 모듈), 750
CLD_TRAPPED (*os* 모듈), 750
clean() (*mailbox.Maildir* 메서드), 1375
cleandoc() (*inspect* 모듈), 2087
CleanImport (*test.support.import_helper* 클래스), 1911
cleanup() (*tempfile.TemporaryDirectory* 메서드), 503
CLEANUP_THROW (*opcode*), 2231
--clear
    venv 명령줄 옵션, 1968
clear (*pdb command*), 1933
Clear Breakpoint, 1670
clear() (*array.array* 메서드), 303
clear() (*asyncio.Event* 메서드), 1142
clear() (*collections.deque* 메서드), 275
clear() (*curses.window* 메서드), 966
clear() (*dbm.gnu.gdbm* 메서드), 548
clear() (*dbm.ndbm.ndbm* 메서드), 549
clear() (*dict* 메서드), 95
clear() (*email.message.EmailMessage* 메서드), 1311
clear() (*frozenset* 메서드), 93
clear() (*http.cookiejar.CookieJar* 메서드), 1569
clear() (*mailbox.Mailbox* 메서드), 1373
clear() (*sequence* 메서드), 49
clear() (*threading.Event* 메서드), 1010
clear() (*turtle* 모듈), 1696
clear() (*xml.etree.ElementTree.Element* 메서드), 1419
clear_all_breaks() (*bdb.Bdb* 메서드), 1923
clear_all_file_breaks() (*bdb.Bdb* 메서드), 1923
clear_bpbynumber() (*bdb.Bdb* 메서드), 1923
clear_break() (*bdb.Bdb* 메서드), 1923
clear_cache() (*filecmp* 모듈), 499
clear_cache() (*zoneinfo.ZoneInfo*의 클래스 메서드), 258
clear_content() (*email.message.EmailMessage* 메서드), 1311
clear_flags() (*decimal.Context* 메서드), 378
clear_frames() (*traceback* 모듈), 2067
clear_history() (*readline* 모듈), 186
clear_memo() (*pickle.Pickler* 메서드), 527
clear_overloads() (*typing* 모듈), 1761
clear_session_cookies() (*http.cookiejar.CookieJar* 메서드), 1569
clear_tool_id() (*sys.monitoring* 모듈), 2009
clear_traces() (*tracemalloc* 모듈), 1959
clear_traps() (*decimal.Context* 메서드), 378
clearcache() (*linecache* 모듈), 510
clearok() (*curses.window* 메서드), 966
clearscreen() (*turtle* 모듈), 1703
clearstamp() (*turtle* 모듈), 1689
clearstamps() (*turtle* 모듈), 1689
Client() (*multiprocessing.connection* 모듈), 1044
client_address (*http.server.BaseHTTPRequestHandler*의 속성), 1557
client_address (*socketserver.BaseRequestHandler*의 속성), 1552
CLOCK_BOOTTIME (*time* 모듈), 781

clock_getres() (*time* 모듈), 773
clock_gettime() (*time* 모듈), 773
clock_gettime_ns() (*time* 모듈), 773
CLOCK_HIGHRES (*time* 모듈), 781
CLOCK_MONOTONIC (*time* 모듈), 781
CLOCK_MONOTONIC_RAW (*time* 모듈), 782
CLOCK_MONOTONIC_RAW_APPROX (*time* 모듈), 782
CLOCK_PROCESS_CPUTIME_ID (*time* 모듈), 782
CLOCK_PROF (*time* 모듈), 782
CLOCK_REALTIME (*time* 모듈), 783
clock_seq (*uuid.UUID*의 속성), 1543
clock_seq_hi_variant (*uuid.UUID*의 속성), 1543
clock_seq_low (*uuid.UUID*의 속성), 1543
clock_settime() (*time* 모듈), 773
clock_settime_ns() (*time* 모듈), 773
CLOCK_TAI (*time* 모듈), 782
CLOCK_THREAD_CPUTIME_ID (*time* 모듈), 782
CLOCK_UPTIME (*time* 모듈), 782
CLOCK_UPTIME_RAW (*time* 모듈), 782
CLOCK_UPTIME_RAW_APPROX (*time* 모듈), 782
clone() (*email.generator.BytesGenerator* 메서드), 1316
clone() (*email.generator.Generator* 메서드), 1317
clone() (*email.policy.Policy* 메서드), 1320
clone() (*turtle* 모듈), 1701
CLONE_FILES (*os* 모듈), 699
CLONE_FS (*os* 모듈), 699
CLONE_NEWCGROUP (*os* 모듈), 699
CLONE_NEWIPC (*os* 모듈), 699
CLONE_NEWNET (*os* 모듈), 699
CLONE_NEWNS (*os* 모듈), 699
CLONE_NEWPID (*os* 모듈), 699
CLONE_NEWTIME (*os* 모듈), 699
CLONE_NEWUSER (*os* 모듈), 699
CLONE_NEWUTS (*os* 모듈), 699
CLONE_SIGHAND (*os* 모듈), 699
CLONE_SYSVSEM (*os* 모듈), 699
CLONE_THREAD (*os* 모듈), 699
CLONE_VM (*os* 모듈), 699
cloneNode() (*xml.dom.Node* 메서드), 1430
close() (*asyncio.BaseTransport* 메서드), 1186
close() (*asyncio.loop* 메서드), 1158
close() (*asyncio.Runner* 메서드), 1112
close() (*asyncio.Server* 메서드), 1176
close() (*asyncio.StreamWriter* 메서드), 1136
close() (*asyncio.SubprocessTransport* 메서드), 1189
close() (*concurrent.interpreters.Interpreter* 메서드), 1075
close() (*contextlib.ExitStack* 메서드), 2052
close() (*dbm.dumb.dumbdbm* 메서드), 551
close() (*dbm.gnu.gdbm* 메서드), 548
close() (*dbm.ndbm.ndbm* 메서드), 549
close() (*email.parser.BytesFeedParser* 메서드), 1313
close() (*fileinput* 모듈), 956
close() (*ftplib.FTP* 메서드), 1522
close() (*html.parser.HTMLParser* 메서드), 1403
close() (*http.client.HTTPConnection* 메서드), 1514
close() (*imaplib.IMAP4* 메서드), 1529
close() (*io.IOBase* 메서드), 761
close() (*logging.FileHandler* 메서드), 815
close() (*logging.Handler* 메서드), 791
close() (*logging.handlers.MemoryHandler* 메서드), 825
close() (*logging.handlers.NTEventLogHandler* 메서드), 823
close() (*logging.handlers.SocketHandler* 메서드), 819
close() (*logging.handlers.SysLogHandler* 메서드), 821
close() (*mailbox.Mailbox* 메서드), 1374
close() (*mailbox.Maildir* 메서드), 1376
close() (*mailbox.MH* 메서드), 1379
close() (*mmap.mmap* 메서드), 1299
close() (*multiprocessing.connection.Connection* 메서드), 1028
close() (*multiprocessing.connection.Listener* 메서드), 1045
close() (*multiprocessing.pool.Pool* 메서드), 1043
close() (*multiprocessing.Process* 메서드), 1022
close() (*multiprocessing.Queue* 메서드), 1025
close() (*multiprocessing.shared_memory.SharedMemory* 메서드), 1058
close() (*multiprocessing.SimpleQueue* 메서드), 1025
close() (*os* 모듈), 699
close() (*os.scandir* 메서드), 722
close() (*select.devpoll* 메서드), 1279
close() (*select.epoll* 메서드), 1280
close() (*select.kqueue* 메서드), 1282
close() (*selectors.BaseSelector* 메서드), 1286
close() (*shelve.Shelf* 메서드), 541
close() (*socket* 모듈), 1225
close() (*socket.socket* 메서드), 1230
close() (*sqlite3.Blob* 메서드), 571
close() (*sqlite3.Connection* 메서드), 558
close() (*sqlite3.Cursor* 메서드), 569
close() (*tarfile.TarFile* 메서드), 630
close() (*urllib.request.BaseHandler* 메서드), 1486
close() (*wave.Wave_read* 메서드), 1606
close() (*wave.Wave_write* 메서드), 1607
Close() (*winreg.PyHKEY* 메서드), 2258
close() (*xml.etree.ElementTree.TreeBuilder* 메서드), 1423
close() (*xml.etree.ElementTree.XMLParser* 메서드), 1424
close() (*xml.etree.ElementTree.XMLPullParser* 메서드), 1425
close() (*xml.sax.xmlreader.IncrementalParser* 메서드), 1452
close() (*zipfile.ZipFile* 메서드), 614
close_clients() (*asyncio.Server* 메서드), 1176
close_connection (*http.server.BaseHTTPRequestHandler*의 속성), 1557
CloseBoundaryNotFoundDefect, 1326
closed (*http.client.HTTPResponse*의 속성), 1515
closed (*io.IOBase*의 속성), 761
closed (*mmap.mmap*의 속성), 1299

closed (*select.devpoll*의 속성), 1279
closed (*select.epoll*의 속성), 1280
closed (*select.kqueue*의 속성), 1282
CloseKey() (*winreg* 모듈), 2250
closelog() (*syslog* 모듈), 2283
closerange() (*os* 모듈), 699
closing() (*contextlib* 모듈), 2046
closure variable (클로저 변수), **2304**
clrtobot() (*curses.window* 메서드), 966
clrtoeol() (*curses.window* 메서드), 966
cmath
    module, 360
cmd
    module, 988
    모듈, 1928
Cmd (*cmd* 클래스), 988
cmd (*subprocess.CalledProcessError*의 속성), 1079
cmd (*subprocess.TimeoutExpired*의 속성), 1079
cmdloop() (*cmd.Cmd* 메서드), 989
cmdqueue (*cmd.Cmd*의 속성), 990
cmp() (*filecmp* 모듈), 499
cmp_op (*dis* 모듈), 2244
cmp_to_key() (*functools* 모듈), 442
cmpfiles() (*filecmp* 모듈), 499
CMSG_LEN() (*socket* 모듈), 1228
CMSG_SPACE() (*socket* 모듈), 1228
CO_ASYNC_GENERATOR (*inspect* 모듈), 2099
CO_COROUTINE (*inspect* 모듈), 2099
CO_GENERATOR (*inspect* 모듈), 2098
CO_HAS_DOCSTRING (*inspect* 모듈), 2099
CO_ITERABLE_COROUTINE (*inspect* 모듈), 2099
CO_METHOD (*inspect* 모듈), 2099
CO_NESTED (*inspect* 모듈), 2098
CO_NEWLOCALS (*inspect* 모듈), 2098
CO_OPTIMIZED (*inspect* 모듈), 2098
CO_VARARGS (*inspect* 모듈), 2098
CO_VARKEYWORDS (*inspect* 모듈), 2098
code
    module, 2113
code (*SystemExit*의 속성), 117
code (*urllib.error.HTTPError*의 속성), 1505
code (*urllib.response.addinfourl*의 속성), 1495
code (*xml.etree.ElementTree.ParseError*의 속성), 1426
code (*xml.parsers.expat.ExpatError*의 속성), 1460
code object, 106
code_context (*inspect.FrameInfo*의 속성), 2094
code_context (*inspect.Traceback*의 속성), 2095
code_info() (*dis* 모듈), 2225
Codec (*codecs* 클래스), 204
CodecInfo (*codecs* 클래스), 199
CodecRegistryError, 215
codecs
    module, 198
coded_value (*http.cookies.Morsel*의 속성), 1565
codeop
    module, 2115
codepoint2name (*html.entities* 모듈), 1406
codes (*xml.parsers.expat.errors* 모듈), 1461
CODESET (*locale* 모듈), 1619
CodeType (*types* 클래스), 313
col_offset (*ast.AST*의 속성), 2165
collapse_addresses() (*ipaddress* 모듈), 1603
collapse_rfc2231_value() (*email.utils* 모듈), 1359
collect() (*gc* 모듈), 2077
collectedDurations (*unittest.TestResult*의 속성), 1825
Collection (*collections.abc* 클래스), 290
Collection (*typing* 클래스), 1769
collections
    module, 269
collections.abc
    module, 287
colno (*json.JSONDecodeError*의 속성), 1368
colno (*re.PatternError*의 속성), 154
colno (*tomllib.TOMLDecodeError*의 속성), 668
colno (*traceback.FrameSummary*의 속성), 2071
colon (*mailbox.Maildir*의 속성), 1374
COLON (*token* 모듈), 2207
COLONEQUAL (*token* 모듈), 2209
color() (*turtle* 모듈), 1694
COLOR_BLACK (*curses* 모듈), 982
COLOR_BLUE (*curses* 모듈), 982
color_content() (*curses* 모듈), 958
COLOR_CYAN (*curses* 모듈), 982
COLOR_GREEN (*curses* 모듈), 982
COLOR_MAGENTA (*curses* 모듈), 982
color_pair() (*curses* 모듈), 958
COLOR_PAIRS (*curses* 모듈), 971
COLOR_RED (*curses* 모듈), 982
COLOR_WHITE (*curses* 모듈), 982
COLOR_YELLOW (*curses* 모듈), 982
colormode() (*turtle* 모듈), 1707
COLORS (*curses* 모듈), 971
colorsys
    module, 1608
COLS (*curses* 모듈), 971
column() (*tkinter.ttk.Treeview* 메서드), 1658
columnize() (*cmd.Cmd* 메서드), 989
COLUMNS, 964
columns (*os.terminal_size*의 속성), 711
comb() (*math* 모듈), 352
combinations() (*itertools* 모듈), 426
combinations_with_replacement() (*itertools* 모듈), 427
combine() (*datetime.datetime*의 클래스 메서드), 231
combining() (*unicodedata* 모듈), 182
Combobox (*tkinter.ttk* 클래스), 1652
COMError, 878
COMMA (*token* 모듈), 2207
--command
    pdb 명령줄 옵션, 1929
command (*http.server.BaseHTTPRequestHandler*의 속성), 1557
CommandCompiler (*codeop* 클래스), 2116
commands (*pdb command*), 1934

comment (*http.cookiejar.Cookie*의 속성), 1574
comment (*http.cookies.Morsel*의 속성), 1565
COMMENT (*token* 모듈), 2205
comment (*zipfile.ZipFile*의 속성), 617
comment (*zipfile.ZipInfo*의 속성), 621
Comment() (*xml.etree.ElementTree* 모듈), 1415
comment() (*xml.etree.ElementTree.TreeBuilder* 메서드), 1423
comment() (*xml.sax.handler.LexicalHandler* 메서드), 1449
comment_url (*http.cookiejar.Cookie*의 속성), 1574
commenters (*shlex.shlex*의 속성), 2266
CommentHandler() (*xml.parsers.expat.xmlparser* 메서드), 1459
commit() (*sqlite3.Connection* 메서드), 558
common (*filecmp.dircmp*의 속성), 500
Common Vulnerabilities and Exposures
    CVE 2020-10735, 108
    CVE 2023-52425, 1407
Common Weakness Enumeration
    CWE 257, 688
common_dirs (*filecmp.dircmp*의 속성), 500
common_files (*filecmp.dircmp*의 속성), 500
common_funny (*filecmp.dircmp*의 속성), 500
common_types (*mimetypes* 모듈), 1391
commonpath() (*os.path* 모듈), 487
commonprefix() (*os.path* 모듈), 487
communicate() (*asyncio.subprocess.Process* 메서드), 1148
communicate() (*subprocess.Popen* 메서드), 1086
--compact
    json 명령줄 옵션, 1370
Compare (*ast* 클래스), 2172
compare() (*ast* 모듈), 2199
compare() (*decimal.Context* 메서드), 379
compare() (*decimal.Decimal* 메서드), 370
compare() (*difflib.Differ* 메서드), 172
compare_digest() (*hmac* 모듈), 686
compare_digest() (*secrets* 모듈), 688
compare_networks() (*ipaddress.IPv4Network* 메서드), 1598
compare_networks() (*ipaddress.IPv6Network* 메서드), 1600
COMPARE_OP (*opcode*), 2237
compare_signal() (*decimal.Context* 메서드), 379
compare_signal() (*decimal.Decimal* 메서드), 370
compare_to() (*tracemalloc.Snapshot* 메서드), 1961
compare_total() (*decimal.Context* 메서드), 379
compare_total() (*decimal.Decimal* 메서드), 370
compare_total_mag() (*decimal.Context* 메서드), 379
compare_total_mag() (*decimal.Decimal* 메서드), 370
comparing
    objects, 38
comparison
    operator, 38
COMPARISON_FLAGS (*doctest* 모듈), 1785
comparisons
    chaining, 38
compat32 (*email.policy* 모듈), 1325
Compat32 (*email.policy* 클래스), 1324
compile
    built-in function, 106
Compile (*codeop* 클래스), 2116
compile()
    built-in function, 11
compile() (*py_compile* 모듈), 2217
compile() (*re* 모듈), 151
compile_command() (*code* 모듈), 2114
compile_command() (*codeop* 모듈), 2115
compile_dir() (*compileall* 모듈), 2220
compile_file() (*compileall* 모듈), 2221
compile_path() (*compileall* 모듈), 2222
compileall
    module, 2218
compileall 명령줄 옵션
    -b, 2219
    -d, 2219
    directory, 2219
    -e, 2220
    -f, 2219
    file, 2219
    --hardlink-dupes, 2220
    -i, 2219
    --invalidation-mode, 2219
    -j, 2219
    -l, 2219
    -o, 2220
    -p, 2219
    -q, 2219
    -r, 2219
    -s, 2219
    -x, 2219
compiler_flag (*__future__._Feature*의 속성), 2077
complete() (*rlcompleter.Completer* 메서드), 189
complete_statement() (*sqlite3* 모듈), 554
completedefault() (*cmd.Cmd* 메서드), 989
CompletedProcess (*subprocess* 클래스), 1078
Completer (*rlcompleter* 클래스), 189
complex
    built-in function, 39
Complex (*numbers* 클래스), 347
complex (내장 클래스), 12
complex number
    literals, 39
    object, 38
complex number (복소수), **2304**
comprehension (*ast* 클래스), 2175
--compress
    zipapp 명령줄 옵션, 1977
compress() (*bz2* 모듈), 604
compress() (*bz2.BZ2Compressor* 메서드), 603
compress() (*compression.zstd* 모듈), 585
compress() (*compression.zstd.ZstdCompressor* 메서드), 586

compress() (*gzip* 모듈), 600
compress() (*itertools* 모듈), 427
compress() (*lzma* 모듈), 609
compress() (*lzma.LZMACompressor* 메서드), 608
compress() (*zlib* 모듈), 594
compress() (*zlib.Compress* 메서드), 596
compress_size (*zipfile.ZipInfo*의 속성), 622
compress_type (*zipfile.ZipInfo*의 속성), 621
compressed (*ipaddress.IPv4Address*의 속성), 1591
compressed (*ipaddress.IPv4Network*의 속성), 1596
compressed (*ipaddress.IPv6Address*의 속성), 1593
compressed (*ipaddress.IPv6Network*의 속성), 1599
compression() (*ssl.SSLSocket* 메서드), 1257
compression_level (*compression.zstd.CompressionParameter* 의 속성), 589
COMPRESSION_LEVEL_DEFAULT (*compression.zstd* 모듈), 593
CompressionError, 625
CompressionParameter (*compression.zstd* 클래스), 589
compression.zstd
module, 583
compressobj() (*zlib* 모듈), 595
COMSPEC, 747, 1082
concat() (*operator* 모듈), 453
Concatenate (*typing* 모듈), 1733
concatenation
operation, 47
concurrent.futures
module, 1063
concurrent.interpreters
module, 1072
cond (*bdb.Breakpoint*의 속성), 1920
Condition (*asyncio* 클래스), 1142
Condition (*multiprocessing* 클래스), 1030
condition (*pdb command*), 1934
Condition (*threading* 클래스), 1007
Condition() (*multiprocessing.managers.SyncManager* 메서드), 1036
config() (*tkinter.font.Font* 메서드), 1641
configparser
module, 649
ConfigParser (*configparser* 클래스), 661
configure() (*tkinter.ttk.Style* 메서드), 1662
configure_mock() (*unittest.mock.Mock* 메서드), 1837
CONFORM (*enum.FlagBoundary*의 속성), 340
confstr() (*os* 모듈), 753
confstr_names (*os* 모듈), 754
conjugate() (*complex number method*), 39
conjugate() (*decimal.Decimal* 메서드), 371
conjugate() (*numbers.Complex* 메서드), 347
connect() (*ftplib.FTP* 메서드), 1519
connect() (*http.client.HTTPConnection* 메서드), 1514
connect() (*multiprocessing.managers.BaseManager* 메서드), 1035
connect() (*smtplib.SMTP* 메서드), 1537
connect() (*socket.socket* 메서드), 1231
connect() (*sqlite3* 모듈), 553
connect_accepted_socket() (*asyncio.loop* 메서드), 1166
connect_ex() (*socket.socket* 메서드), 1231
connect_read_pipe() (*asyncio.loop* 메서드), 1170
connect_write_pipe() (*asyncio.loop* 메서드), 1170
Connection (*multiprocessing.connection* 클래스), 1028
Connection (*sqlite3* 클래스), 557
connection (*sqlite3.Cursor*의 속성), 569
connection_lost() (*asyncio.BaseProtocol* 메서드), 1189
connection_made() (*asyncio.BaseProtocol* 메서드), 1189
ConnectionAbortedError, 119
ConnectionError, 119
ConnectionRefusedError, 119
ConnectionResetError, 119
ConnectRegistry() (*winreg* 모듈), 2250
const (*optparse.Option*의 속성), 941
Constant (*ast* 클래스), 2167
constructor() (*copyreg* 모듈), 540
consumed (*asyncio.LimitOverrunError*의 속성), 1154
container
iteration over, 46
Container (*collections.abc* 클래스), 290
Container (*typing* 클래스), 1769
contains() (*operator* 모듈), 453
CONTAINS_OP (*opcode*), 2237
content (*urllib.error.ContentTooShortError*의 속성), 1505
content_disposition (*email.headerregistry.ContentDispositionHeader* 의 속성), 1329
content_manager (*email.policy.EmailPolicy*의 속성), 1322
content_size_flag (*compression.zstd.CompressionParameter* 의 속성), 591
content_type (*email.headerregistry.ContentTypeHeader* 의 속성), 1329
ContentDispositionHeader (*email.headerregistry* 클래스), 1329
ContentHandler (*xml.sax.handler* 클래스), 1444
ContentManager (*email.contentmanager* 클래스), 1332
contents (*ctypes._Pointer*의 속성), 878
contents() (*importlib.abc.ResourceReader* 메서드), 2133
contents() (*importlib.resources* 모듈), 2148
contents() (*importlib.resources.abc.ResourceReader* 메서드), 2149
ContentTooShortError, 1505
ContentTransferEncoding (*email.headerregistry* 클래스), 1329
ContentTypeHeader (*email.headerregistry* 클래스), 1329
Context (*contextvars* 클래스), 1103

Context (*decimal* 클래스), 377
context (*ssl.SSLSocket*의 속성), 1258
context (컨텍스트), **2304**
context management protocol, 98
context management protocol (컨텍스트 관리 프로토콜), **2304**
context manager, 98
context manager (컨텍스트 관리자), **2304**
context variable (컨텍스트 변수), **2304**
context_aware_warnings (*sys.flags*의 속성), 1988
context_diff() (*difflib* 모듈), 166
ContextDecorator (*contextlib* 클래스), 2049
contextlib
    module, 2044
ContextManager (*typing* 클래스), 1772
contextmanager() (*contextlib* 모듈), 2044
ContextVar (*contextvars* 클래스), 1101
contextvars
    module, 1101
CONTIG (*inspect.BufferFlags*의 속성), 2100
CONTIG_RO (*inspect.BufferFlags*의 속성), 2100
contiguous (*memoryview*의 속성), 91
contiguous (연속), **2304**
Continue (*ast* 클래스), 2181
CONTINUE (*compression.zstd.ZstdCompressor*의 속성), 586
continue (*pdb command*), 1935
CONTINUOUS (*enum.EnumCheck*의 속성), 339
control() (*select.kqueue* 메서드), 1282
controlnames (*curses.ascii* 모듈), 987
CONTTYPE (*tarfile* 모듈), 626
conversion (*string.templatelib.Interpolation*의 속성), 140
conversions
    numeric, 39
convert() (*string.templatelib* 모듈), 141
convert_arg_line_to_args() (*argparse.ArgumentParser* 메서드), 910
convert_field() (*string.Formatter* 메서드), 127
CONVERT_VALUE (*opcode*), 2241
Cookie (*http.cookiejar* 클래스), 1568
CookieError, 1563
CookieJar (*http.cookiejar* 클래스), 1567
cookiejar (*urllib.request.HTTPCookieProcessor*의 속성), 1488
CookiePolicy (*http.cookiejar* 클래스), 1568
--copies
    venv 명령줄 옵션, 1968
Copy, 1670
copy
    module, 317
COPY (*opcode*), 2229
copy() (*collections.deque* 메서드), 275
copy() (*contextvars.Context* 메서드), 1104
copy() (*copy* 모듈), 318
copy() (*decimal.Context* 메서드), 378
copy() (*dict* 메서드), 95
copy() (*frozenset* 메서드), 92
copy() (*hashlib.hash* 메서드), 676
copy() (*hmac.HMAC* 메서드), 686
copy() (*http.cookies.Morsel* 메서드), 1566
copy() (*imaplib.IMAP4* 메서드), 1529
copy() (*multiprocessing.sharedctypes* 모듈), 1033
copy() (*pathlib.Path* 메서드), 481
copy() (*sequence* 메서드), 49
copy() (*shutil* 모듈), 513
copy() (*tkinter.font.Font* 메서드), 1641
copy() (*types.MappingProxyType* 메서드), 316
copy() (*zlib.Compress* 메서드), 596
copy() (*zlib.Decompress* 메서드), 597
copy2() (*shutil* 모듈), 513
copy_abs() (*decimal.Context* 메서드), 379
copy_abs() (*decimal.Decimal* 메서드), 371
copy_context() (*contextvars* 모듈), 1103
copy_decimal() (*decimal.Context* 메서드), 378
copy_file_range() (*os* 모듈), 700
COPY_FREE_VARS (*opcode*), 2239
copy_into() (*pathlib.Path* 메서드), 481
copy_location() (*ast* 모듈), 2197
copy_negate() (*decimal.Context* 메서드), 379
copy_negate() (*decimal.Decimal* 메서드), 371
copy_sign() (*decimal.Context* 메서드), 380
copy_sign() (*decimal.Decimal* 메서드), 371
CopyComPointer() (*ctypes* 모듈), 867
copyfile() (*shutil* 모듈), 511
copyfileobj() (*shutil* 모듈), 511
copymode() (*shutil* 모듈), 512
copyreg
    module, 540
copyright (*sys* 모듈), 1983
copyright (내장 변수), 36
copysign() (*math* 모듈), 354
copystat() (*shutil* 모듈), 512
copytree() (*shutil* 모듈), 513
Coroutine (*collections.abc* 클래스), 291
Coroutine (*typing* 클래스), 1770
coroutine (코루틴), **2305**
coroutine function (코루틴 함수), **2305**
coroutine() (*types* 모듈), 317
CoroutineType (*types* 모듈), 313
correlation() (*statistics* 모듈), 416
cos() (*cmath* 모듈), 362
cos() (*math* 모듈), 358
cosh() (*cmath* 모듈), 362
cosh() (*math* 모듈), 358
--count
    trace 명령줄 옵션, 1952
    uuid 명령줄 옵션, 1546
count (*tracemalloc.StatisticDiff*의 속성), 1963
count (*tracemalloc.Statistic*의 속성), 1962
count() (*array.array* 메서드), 302
count() (*bytearray* 메서드), 71
count() (*bytes* 메서드), 71
count() (*collections.deque* 메서드), 275
count() (*itertools* 모듈), 428
count() (*memoryview* 메서드), 89

count() (*multiprocessing.shared_memory.ShareableList* 메서드), 1062
count() (*sequence* 메서드), 48
count() (*str* 메서드), 55
count_diff (*tracemalloc.StatisticDiff*의 속성), 1963
Counter (*collections* 클래스), 272
Counter (*typing* 클래스), 1768
countOf() (*operator* 모듈), 453
countTestCases() (*unittest.TestCase* 메서드), 1818
countTestCases() (*unittest.TestSuite* 메서드), 1821
covariance() (*statistics* 모듈), 415
CoverageResults (*trace* 클래스), 1953
--coverdir
    trace 명령줄 옵션, 1953
cProfile
    module, 1941
cProfile 명령줄 옵션
    -m, 1940
    -o, 1940
    -s, 1940
CPU 시간, 775, 780
cpu_count() (*multiprocessing* 모듈), 1026
cpu_count() (*os* 모듈), 754
CPython, **2305**
cpython_only() (*test.support* 모듈), 1902
CR (*curses.ascii* 모듈), 984
crawl_delay() (*urllib.robotparser.RobotFileParser* 메서드), 1506
CRC (*zipfile.ZipInfo*의 속성), 622
crc32() (*binascii* 모듈), 1399
crc32() (*zlib* 모듈), 595
crc_hqx() (*binascii* 모듈), 1399
--create
    tarfile 명령줄 옵션, 637
    zipfile 명령줄 옵션, 622
create() (*concurrent.interpreters* 모듈), 1075
create() (*imaplib.IMAP4* 메서드), 1529
create() (*venv* 모듈), 1972
create() (*venv.EnvBuilder* 메서드), 1971
create_aggregate() (*sqlite3.Connection* 메서드), 559
create_archive() (*zipapp* 모듈), 1977
create_autospec() (*unittest.mock* 모듈), 1865
CREATE_BREAKAWAY_FROM_JOB (*subprocess* 모듈), 1090
create_collation() (*sqlite3.Connection* 메서드), 561
create_configuration() (*venv.EnvBuilder* 메서드), 1972
create_connection() (*asyncio.loop* 메서드), 1161
create_connection() (*socket* 모듈), 1223
create_datagram_endpoint() (*asyncio.loop* 메서드), 1163
create_decimal() (*decimal.Context* 메서드), 378
create_decimal_from_float() (*decimal.Context* 메서드), 379
create_default_context() (*ssl* 모듈), 1243
CREATE_DEFAULT_ERROR_MODE (*subprocess* 모듈), 1090
create_eager_task_factory() (*asyncio* 모듈), 1121
create_empty_file() (*test.support.os_helper* 모듈), 1909
create_function() (*sqlite3.Connection* 메서드), 558
create_future() (*asyncio.loop* 메서드), 1160
create_git_ignore_file() (*venv.EnvBuilder* 메서드), 1972
create_module() (*importlib.abc.Loader* 메서드), 2129
create_module() (*importlib.machinery.ExtensionFileLoader* 메서드), 2138
create_module() (*zipimport.zipimporter* 메서드), 2118
CREATE_NEW_CONSOLE (*subprocess* 모듈), 1089
CREATE_NEW_PROCESS_GROUP (*subprocess* 모듈), 1089
CREATE_NO_WINDOW (*subprocess* 모듈), 1090
create_queue() (*concurrent.interpreters* 모듈), 1075
create_server() (*asyncio.loop* 메서드), 1164
create_server() (*socket* 모듈), 1223
create_stats() (*profile.Profile* 메서드), 1942
create_string_buffer() (*ctypes* 모듈), 868
create_subprocess_exec() (*asyncio* 모듈), 1147
create_subprocess_shell() (*asyncio* 모듈), 1147
create_system (*zipfile.ZipInfo*의 속성), 621
create_task() (*asyncio* 모듈), 1116
create_task() (*asyncio.loop* 메서드), 1160
create_task() (*asyncio.TaskGroup* 메서드), 1117
create_unicode_buffer() (*ctypes* 모듈), 868
create_unix_connection() (*asyncio.loop* 메서드), 1164
create_unix_server() (*asyncio.loop* 메서드), 1166
create_version (*zipfile.ZipInfo*의 속성), 621
create_window_function() (*sqlite3.Connection* 메서드), 559
createAttribute() (*xml.dom.Document* 메서드), 1432
createAttributeNS() (*xml.dom.Document* 메서드), 1432
createComment() (*xml.dom.Document* 메서드), 1431
createDocument() (*xml.dom.DOMImplementation* 메서드), 1428
createDocumentType() (*xml.dom.DOMImplementation* 메서드), 1428
createElement() (*xml.dom.Document* 메서드), 1431
createElementNS() (*xml.dom.Document* 메서드), 1431
createfilehandler() (*_tkinter.Widget.tk* 메서드), 1639
CreateKey() (*winreg* 모듈), 2250
CreateKeyEx() (*winreg* 모듈), 2250
createLock() (*logging.Handler* 메서드), 790
createLock() (*logging.NullHandler* 메서드), 815

createProcessingInstruction() (*xml.dom.Document* 메서드), 1431
createSocket() (*logging.handlers.SocketHandler* 메서드), 820
createSocket() (*logging.handlers.SysLogHandler* 메서드), 821
createTextNode() (*xml.dom.Document* 메서드), 1431
credits (내장 변수), 36
CRITICAL (*logging* 모듈), 790
critical() (*logging* 모듈), 798
critical() (*logging.Logger* 메서드), 788
CRNCYSTR (*locale* 모듈), 1620
CRT_ASSEMBLY_VERSION (*msvcrt* 모듈), 2249
CRT_ASSERT (*msvcrt* 모듈), 2249
CRT_ERROR (*msvcrt* 모듈), 2249
CRT_WARN (*msvcrt* 모듈), 2249
CRTDBG_MODE_DEBUG (*msvcrt* 모듈), 2249
CRTDBG_MODE_FILE (*msvcrt* 모듈), 2249
CRTDBG_MODE_WNDW (*msvcrt* 모듈), 2249
CRTDBG_REPORT_MODE (*msvcrt* 모듈), 2249
CrtSetReportFile() (*msvcrt* 모듈), 2249
CrtSetReportMode() (*msvcrt* 모듈), 2249
crypt
    module, 2294
--css
    calendar 명령줄 옵션, 269
cssclass_month (*calendar.HTMLCalendar*의 속성), 264
cssclass_month_head (*calendar.HTMLCalendar*의 속성), 264
cssclass_noday (*calendar.HTMLCalendar*의 속성), 264
cssclass_year (*calendar.HTMLCalendar*의 속성), 264
cssclass_year_head (*calendar.HTMLCalendar*의 속성), 264
cssclasses (*calendar.HTMLCalendar*의 속성), 263
cssclasses_weekday_head (*calendar.HTMLCalendar*의 속성), 264
csv, 641
    module, 641
cte (*email.headerregistry.ContentTransferEncoding*의 속성), 1329
cte_type (*email.policy.Policy*의 속성), 1320
ctermid() (*os* 모듈), 691
ctime() (*datetime.date* 메서드), 227
ctime() (*datetime.datetime* 메서드), 237
ctime() (*time* 모듈), 773
ctrl() (*curses.ascii* 모듈), 987
CTRL_BREAK_EVENT (*signal* 모듈), 1291
CTRL_C_EVENT (*signal* 모듈), 1291
ctypes
    module, 841
curdir (*os* 모듈), 754
currency() (*locale* 모듈), 1622
current context (현재 컨텍스트), **2305**
current() (*tkinter.ttk.Combobox* 메서드), 1652
current_process() (*multiprocessing* 모듈), 1026
current_task() (*asyncio* 모듈), 1128
current_thread() (*threading* 모듈), 997
CurrentByteIndex (*xml.parsers.expat.xmlparser*의 속성), 1458
CurrentColumnNumber (*xml.parsers.expat.xmlparser*의 속성), 1458
currentframe() (*inspect* 모듈), 2096
CurrentLineNumber (*xml.parsers.expat.xmlparser*의 속성), 1458
curs_set() (*curses* 모듈), 958
curses
    module, 957
curses.ascii
    module, 984
curses.panel
    module, 987
curses.textpad
    module, 982
Cursor (*sqlite3* 클래스), 568
cursor() (*sqlite3.Connection* 메서드), 557
cursyncup() (*curses.window* 메서드), 966
Cut, 1670
cwd() (*ftplib.FTP* 메서드), 1521
cwd() (*pathlib.Path*의 클래스 메서드), 473
cycle() (*itertools* 모듈), 428
CycleError, 346
cyclic isolate, **2305**

D

-d
    compileall 명령줄 옵션, 2219
    gzip 명령줄 옵션, 601
    http.server 명령줄 옵션, 1562
    idle 명령줄 옵션, 1674
D_FMT (*locale* 모듈), 1619
D_T_FMT (*locale* 모듈), 1619
daemon (*multiprocessing.Process*의 속성), 1021
daemon (*threading.Thread*의 속성), 1004
daemon_threads (*socketserver.ThreadingMixIn*의 속성), 1549
data (*collections.UserDict*의 속성), 286
data (*collections.UserList*의 속성), 286
data (*collections.UserString*의 속성), 286
data (*plistlib.UID*의 속성), 671
data (*select.kevent*의 속성), 1284
data (*selectors.SelectorKey*의 속성), 1285
data (*urllib.request.Request*의 속성), 1484
data (*xml.dom.Comment*의 속성), 1433
data (*xml.dom.ProcessingInstruction*의 속성), 1434
data (*xml.dom.Text*의 속성), 1434
data (*xmlrpc.client.Binary*의 속성), 1579
data() (*xml.etree.ElementTree.TreeBuilder* 메서드), 1423
data_filter() (*tarfile* 모듈), 634
data_open() (*urllib.request.DataHandler* 메서드), 1490
data_received() (*asyncio.Protocol* 메서드), 1190

DatabaseError, 572
dataclass() (*dataclasses* 모듈), 2033
dataclass_transform() (*typing* 모듈), 1759
dataclasses
    module, 2033
DataError, 572
datagram_received() (*asyncio.DatagramProtocol* 메서드), 1191
DatagramHandler (*logging.handlers* 클래스), 820
DatagramProtocol (*asyncio* 클래스), 1189
DatagramRequestHandler (*socketserver* 클래스), 1552
DatagramTransport (*asyncio* 클래스), 1185
DataHandler (*urllib.request* 클래스), 1483
date (*datetime* 클래스), 223
date() (*datetime.datetime* 메서드), 234
date_time (*zipfile.ZipInfo*의 속성), 620
date_time_string() (*http.server.BaseHTTPRequestHandler* 메서드), 1560
DateHeader (*email.headerregistry* 클래스), 1328
datetime
    module, 217
datetime (*datetime* 클래스), 229
datetime (*email.headerregistry.DateHeader*의 속성), 1328
DateTime (*xmlrpc.client* 클래스), 1579
Day (*calendar* 클래스), 266
day (*datetime.datetime*의 속성), 232
day (*datetime.date*의 속성), 225
DAY_1 (*locale* 모듈), 1619
DAY_2 (*locale* 모듈), 1619
DAY_3 (*locale* 모듈), 1619
DAY_4 (*locale* 모듈), 1619
DAY_5 (*locale* 모듈), 1619
DAY_6 (*locale* 모듈), 1619
DAY_7 (*locale* 모듈), 1619
day_abbr (*calendar* 모듈), 266
day_name (*calendar* 모듈), 265
daylight (*time* 모듈), 783
days (*datetime.timedelta*의 속성), 221
DbfilenameShelf (*shelve* 클래스), 542
dbm
    module, 545
dbm.dumb
    module, 550
dbm.gnu
    module, 547
    모듈, 541
dbm.ndbm
    module, 549
    모듈, 541
dbm.sqlite3
    module, 547
DC1 (*curses.ascii* 모듈), 985
DC2 (*curses.ascii* 모듈), 985
DC3 (*curses.ascii* 모듈), 985
DC4 (*curses.ascii* 모듈), 985
dcgettext() (*locale* 모듈), 1625
deactivate_stack_trampoline() (*sys* 모듈), 2004
debug (*imaplib.IMAP4*의 속성), 1534
DEBUG (*logging* 모듈), 790
debug (*pdb command*), 1938
DEBUG (*re* 모듈), 149
debug (*shlex.shlex*의 속성), 2266
debug (*sys.flags*의 속성), 1988
debug (*zipfile.ZipFile*의 속성), 617
debug() (*doctest* 모듈), 1798
debug() (*logging* 모듈), 797
debug() (*logging.Logger* 메서드), 787
debug() (*unittest.TestCase* 메서드), 1811
debug() (*unittest.TestSuite* 메서드), 1821
DEBUG_BYTECODE_SUFFIXES (*importlib.machinery* 모듈), 2135
DEBUG_COLLECTABLE (*gc* 모듈), 2081
DEBUG_LEAK (*gc* 모듈), 2081
DEBUG_SAVEALL (*gc* 모듈), 2081
debug_src() (*doctest* 모듈), 1798
DEBUG_STATS (*gc* 모듈), 2081
DEBUG_UNCOLLECTABLE (*gc* 모듈), 2081
debuglevel (*http.client.HTTPResponse*의 속성), 1515
DebugRunner (*doctest* 클래스), 1798
DECEMBER (*calendar* 모듈), 266
decimal
    module, 364
Decimal (*decimal* 클래스), 368
decimal() (*unicodedata* 모듈), 181
DecimalException (*decimal* 클래스), 384
decode
    코덱, 198
decode (*codecs.CodecInfo*의 속성), 199
decode() (*base64* 모듈), 1397
decode() (*bytearray* 메서드), 72
decode() (*bytes* 메서드), 72
decode() (*codecs* 모듈), 199
decode() (*codecs.Codec* 메서드), 204
decode() (*codecs.IncrementalDecoder* 메서드), 205
decode() (*json.JSONDecoder* 메서드), 1366
decode() (*quopri* 모듈), 1400
decode() (*xmlrpc.client.Binary* 메서드), 1579
decode() (*xmlrpc.client.DateTime* 메서드), 1579
decode_header() (*email.header* 모듈), 1353
decode_params() (*email.utils* 모듈), 1359
decode_rfc2231() (*email.utils* 모듈), 1359
decode_source() (*importlib.util* 모듈), 2141
decodebytes() (*base64* 모듈), 1397
DecodedGenerator (*email.generator* 클래스), 1317
decodestring() (*quopri* 모듈), 1400
decomposition() (*unicodedata* 모듈), 182
--decompress
    gzip 명령줄 옵션, 601
decompress() (*bz2* 모듈), 604
decompress() (*bz2.BZ2Decompressor* 메서드), 604
decompress() (*compression.zstd* 모듈), 585

decompress() (*compression.zstd.ZstdDecompressor* 메서드), 587
decompress() (*gzip* 모듈), 600
decompress() (*lzma* 모듈), 609
decompress() (*lzma.LZMADecompressor* 메서드), 609
decompress() (*zlib* 모듈), 595
decompress() (*zlib.Decompress* 메서드), 597
decompressed_size (*compression.zstd.FrameInfo* 의 속성), 593
DecompressionParameter (*compression.zstd* 클래스), 592
decompressobj() (*zlib* 모듈), 596
decorator (데코레이터), **2305**
DEDENT (*token* 모듈), 2205
dedent() (*textwrap* 모듈), 178
deepcopy() (*copy* 모듈), 318
def_prog_mode() (*curses* 모듈), 958
def_shell_mode() (*curses* 모듈), 958
default (*email.policy* 모듈), 1323
default (*inspect.Parameter*의 속성), 2089
default (*optparse.Option*의 속성), 940
DEFAULT (*unittest.mock* 모듈), 1864
default() (*cmd.Cmd* 메서드), 989
default() (*json.JSONEncoder* 메서드), 1367
DEFAULT_BUFFER_SIZE (*io* 모듈), 758
default_bufsize (*xml.dom.pulldom* 모듈), 1442
default_exception_handler() (*asyncio.loop* 메서드), 1173
default_factory (*collections.defaultdict* 의 속성), 279
DEFAULT_FORMAT (*tarfile* 모듈), 627
DEFAULT_IGNORES (*filecmp* 모듈), 500
default_loader() (*xml.etree.ElementInclude* 모듈), 1418
default_max_str_digits (*sys.int_info* 의 속성), 1996
default_open() (*urllib.request.BaseHandler* 메서드), 1487
DEFAULT_PROTOCOL (*pickle* 모듈), 525
DEFAULT_TIMEOUT (*unittest.mock.ThreadingMock* 의 속성), 1848
default_timer() (*timeit* 모듈), 1948
DefaultContext (*decimal* 모듈), 377
DefaultCookiePolicy (*http.cookiejar* 클래스), 1568
defaultdict (*collections* 클래스), 278
DefaultDict (*typing* 클래스), 1767
DefaultEventLoopPolicy (*asyncio* 클래스), 1199
DefaultHandler() (*xml.parsers.expat.xmlparser* 메서드), 1459
DefaultHandlerExpand() (*xml.parsers.expat.xmlparser* 메서드), 1459
--default-pip
    ensurepip 명령줄 옵션, 1966
defaults() (*configparser.ConfigParser* 메서드), 662
DefaultSelector (*selectors* 클래스), 1286
defaultTestLoader (*unittest* 모듈), 1827
defaultTestResult() (*unittest.TestCase* 메서드), 1818
defects (*email.headerregistry.BaseHeader* 의 속성), 1327
defects (*email.message.EmailMessage*의 속성), 1312
defects (*email.message.Message*의 속성), 1349
defpath (*os* 모듈), 755
DefragResult (*urllib.parse* 클래스), 1502
DefragResultBytes (*urllib.parse* 클래스), 1502
degrees() (*math* 모듈), 357
degrees() (*turtle* 모듈), 1691
del
    statement, 49, 93
Del (*ast* 클래스), 2169
DEL (*curses.ascii* 모듈), 986
del_param() (*email.message.EmailMessage* 메서드), 1308
del_param() (*email.message.Message* 메서드), 1347
delattr()
    built-in function, 13
delay() (*turtle* 모듈), 1704
delay_output() (*curses* 모듈), 959
delayload (*http.cookiejar.FileCookieJar* 의 속성), 1570
delch() (*curses.window* 메서드), 966
dele() (*poplib.POP3* 메서드), 1526
Delete (*ast* 클래스), 2178
delete() (*ftplib.FTP* 메서드), 1521
delete() (*imaplib.IMAP4* 메서드), 1529
delete() (*tkinter.ttk.Treeview* 메서드), 1659
DELETE_ATTR (*opcode*), 2234
DELETE_DEREF (*opcode*), 2239
DELETE_FAST (*opcode*), 2239
DELETE_GLOBAL (*opcode*), 2234
DELETE_NAME (*opcode*), 2233
DELETE_SUBSCR (*opcode*), 2230
deleteacl() (*imaplib.IMAP4* 메서드), 1529
deletefilehandler() (*_tkinter.Widget.tk* 메서드), 1640
DeleteKey() (*winreg* 모듈), 2251
DeleteKeyEx() (*winreg* 모듈), 2251
deleteln() (*curses.window* 메서드), 966
deleteMe() (*bdb.Breakpoint* 메서드), 1919
DeleteValue() (*winreg* 모듈), 2251
delimiter (*csv.Dialect*의 속성), 646
delitem() (*operator* 모듈), 454
deliver_challenge() (*multiprocessing.connection* 모듈), 1044
delocalize() (*locale* 모듈), 1622
demo_app() (*wsgiref.simple_server* 모듈), 1471
denominator (*fractions.Fraction*의 속성), 394
denominator (*numbers.Rational*의 속성), 348
deprecated() (*warnings* 모듈), 2031
DeprecationWarning, 120
deque (*collections* 클래스), 275
Deque (*typing* 클래스), 1768
dequeue() (*logging.handlers.QueueListener* 메서드), 827

DER_cert_to_PEM_cert() (*ssl* 모듈), 1246
derive() (*BaseExceptionGroup* 메서드), 122
derwin() (*curses.window* 메서드), 966
description (*inspect.Parameter.kind*의 속성), 2090
description (*sqlite3.Cursor*의 속성), 570
descriptor (디스크립터), **2305**
deserialize() (*sqlite3.Connection* 메서드), 566
dest (*optparse.Option*의 속성), 940
detach() (*io.BufferedIOBase* 메서드), 763
detach() (*io.TextIOBase* 메서드), 767
detach() (*socket.socket* 메서드), 1231
detach() (*tkinter.ttk.Treeview* 메서드), 1659
detach() (*weakref.finalize* 메서드), 307
Detach() (*winreg.PyHKEY* 메서드), 2258
DETACHED_PROCESS (*subprocess* 모듈), 1090
--details
    inspect 명령줄 옵션, 2100
details (*ctypes.COMError*의 속성), 879
detect_api_mismatch() (*test.support* 모듈), 1903
detect_encoding() (*tokenize* 모듈), 2211
dev_mode (*sys.flags*의 속성), 1988
device_encoding() (*os* 모듈), 700
devmajor (*tarfile.TarInfo*의 속성), 632
devminor (*tarfile.TarInfo*의 속성), 632
devnull (*os* 모듈), 755
DEVNULL (*subprocess* 모듈), 1079
devpoll() (*select* 모듈), 1277
DevpollSelector (*selectors* 클래스), 1287
dfast (*compression.zstd.Strategy*의 속성), 592
dgettext() (*gettext* 모듈), 1610
dgettext() (*locale* 모듈), 1625
Dialect (*csv* 클래스), 644
dialect (*csv.csvreader*의 속성), 647
dialect (*csv.csvwriter*의 속성), 647
Dialog (*tkinter.commondialog* 클래스), 1644
Dialog (*tkinter.simpledialog* 클래스), 1642
Dict (*ast* 클래스), 2169
Dict (*typing* 클래스), 1767
dict (내장 클래스), 93
dict() (*multiprocessing.managers.SyncManager* 메서드), 1037
dict_content (*compression.zstd.ZstdDict*의 속성), 588
dict_id (*compression.zstd.ZstdDict*의 속성), 588
dict_id_flag (*compression.zstd.CompressionParameter*의 속성), 591
DICT_MERGE (*opcode*), 2236
DICT_UPDATE (*opcode*), 2236
DictComp (*ast* 클래스), 2174
dictConfig() (*logging.config* 모듈), 802
dictionary
    object, 93
    type, operations on, 93
dictionary (딕셔너리), **2305**
dictionary comprehension (딕셔너리 컴프리헨션), **2305**
dictionary view (딕셔너리 뷰), **2305**
dictionary_id (*compression.zstd.FrameInfo*의 속성), 593
DictReader (*csv* 클래스), 643
DictWriter (*csv* 클래스), 643
diff_bytes() (*difflib* 모듈), 168
diff_files (*filecmp.dircmp*의 속성), 500
Differ (*difflib* 클래스), 165
difference() (*frozenset* 메서드), 92
difference_update() (*frozenset* 메서드), 93
difflib
    module, 164
dig (*sys.float_info*의 속성), 1990
digest() (*hashlib.hash* 메서드), 675
digest() (*hashlib.shake* 메서드), 676
digest() (*hmac* 모듈), 685
digest() (*hmac.HMAC* 메서드), 685
digest_size (*hmac.HMAC*의 속성), 686
digit() (*unicodedata* 모듈), 181
digits (*string* 모듈), 125
dir()
    built-in function, 13
dir() (*ftplib.FTP* 메서드), 1521
dircmp (*filecmp* 클래스), 499
directory
    changing, 713
    compileall 명령줄 옵션, 2219
    creating, 718
    deleting, 720
    traversal, 730, 732
    walking, 730, 732
--directory
    http.server 명령줄 옵션, 1562
Directory (*tkinter.filedialog* 클래스), 1643
DirEntry (*os* 클래스), 722
dirname() (*os.path* 모듈), 487
dirs_double_event() (*tkinter.filedialog.FileDialog* 메서드), 1643
dirs_select_event() (*tkinter.filedialog.FileDialog* 메서드), 1643
DirsOnSysPath (*test.support.import_helper* 클래스), 1911
DIRTYPE (*tarfile* 모듈), 626
dis
    module, 2222
dis 명령줄 옵션
    -C, 2223
    -h, 2223
    --help, 2223
    -O, 2223
    -P, 2223
    -S, 2223
    --show-caches, 2223
    --show-offsets, 2223
    --show-positions, 2223
    --specialized, 2223
dis() (*dis* 모듈), 2225
dis() (*dis.Bytecode* 메서드), 2224
dis() (*pickletools* 모듈), 2246

disable (*pdb command*), 1934
DISABLE (*sys.monitoring* 모듈), 2012
disable() (*bdb.Breakpoint* 메서드), 1919
disable() (*faulthandler* 모듈), 1926
disable() (*gc* 모듈), 2077
disable() (*logging* 모듈), 798
disable() (*profile.Profile* 메서드), 1942
disable_current_event() (*bdb.Bdb* 메서드), 1923
disable_faulthandler() (*test.support* 모듈), 1900
disable_gc() (*test.support* 모듈), 1900
disable_interspersed_args() (*optparse.OptionParser* 메서드), 945
disabled (*logging.Logger*의 속성), 786
DisableReflectionKey() (*winreg* 모듈), 2255
disassemble() (*dis* 모듈), 2226
discard (*http.cookiejar.Cookie*의 속성), 1574
discard() (*frozenset* 메서드), 93
discard() (*mailbox.Mailbox* 메서드), 1371
discard() (*mailbox.MH* 메서드), 1378
disco() (*dis* 모듈), 2226
discover() (*unittest.TestLoader* 메서드), 1823
disk_usage() (*shutil* 모듈), 515
dispatch_call() (*bdb.Bdb* 메서드), 1921
dispatch_exception() (*bdb.Bdb* 메서드), 1922
dispatch_line() (*bdb.Bdb* 메서드), 1921
dispatch_return() (*bdb.Bdb* 메서드), 1921
dispatch_table (*pickle.Pickler*의 속성), 527
DISPLAY, 1629
display (*pdb command*), 1935
display_name (*email.headerregistry.Address*의 속성), 1331
display_name (*email.headerregistry.Group*의 속성), 1331
displayhook() (*sys* 모듈), 1984
dist() (*math* 모듈), 356
distance() (*turtle* 모듈), 1691
distb() (*dis* 모듈), 2225
Distribution (*importlib.metadata* 클래스), 2155
distribution() (*importlib.metadata* 모듈), 2155
distutils
    module, 2294
Div (*ast* 클래스), 2171
divide() (*decimal.Context* 메서드), 380
divide_int() (*decimal.Context* 메서드), 380
division (*__future__* 모듈), 2076
DivisionByZero (*decimal* 클래스), 384
divmod()
    built-in function, 14
divmod() (*decimal.Context* 메서드), 380
DLE (*curses.ascii* 모듈), 985
DllCanUnloadNow() (*ctypes* 모듈), 868
DllGetClassObject() (*ctypes* 모듈), 868
dllhandle (*sys* 모듈), 1984
dllist() (*ctypes.util* 모듈), 869
dnd_start() (*tkinter.dnd* 모듈), 1648
DndHandler (*tkinter.dnd* 클래스), 1647
dngettext() (*gettext* 모듈), 1610
dnpgettext() (*gettext* 모듈), 1610
do_clear() (*bdb.Bdb* 메서드), 1922
do_command() (*curses.textpad.Textbox* 메서드), 983
do_GET() (*http.server.SimpleHTTPRequestHandler* 메서드), 1560
do_handshake() (*ssl.SSLSocket* 메서드), 1255
do_HEAD() (*http.server.SimpleHTTPRequestHandler* 메서드), 1560
do_help() (*cmd.Cmd* 메서드), 989
do_POST() (*http.server.CGIHTTPRequestHandler* 메서드), 1561
doc (*json.JSONDecodeError*의 속성), 1367
doc (*tomllib.TOMLDecodeError*의 속성), 668
doc_header (*cmd.Cmd*의 속성), 990
DocCGIXMLRPCRequestHandler (*xmlrpc.server* 클래스), 1588
DocFileSuite() (*doctest* 모듈), 1789
doClassCleanups() (*unittest.TestCase*의 클래스 메서드), 1819
doCleanups() (*unittest.TestCase* 메서드), 1819
docmd() (*smtplib.SMTP* 메서드), 1537
docstring (*doctest.DocTest*의 속성), 1792
docstring (독스트링), **2306**
doctest
    module, 1777
DocTest (*doctest* 클래스), 1792
doctest 명령줄 옵션
    -f, 1780
    --fail-fast, 1780
    -o, 1780
    --option, 1780
    -v, 1780
    --verbose, 1780
DocTestFailure, 1798
DocTestFinder (*doctest* 클래스), 1793
DocTestParser (*doctest* 클래스), 1794
DocTestRunner (*doctest* 클래스), 1794
DocTestSuite() (*doctest* 모듈), 1790
doctype() (*xml.etree.ElementTree.TreeBuilder* 메서드), 1424
documentElement (*xml.dom.Document*의 속성), 1431
DocXMLRPCRequestHandler (*xmlrpc.server* 클래스), 1588
DocXMLRPCServer (*xmlrpc.server* 클래스), 1588
domain (*email.headerregistry.Address*의 속성), 1331
domain (*http.cookiejar.Cookie*의 속성), 1574
domain (*http.cookies.Morsel*의 속성), 1565
domain (*tracemalloc.DomainFilter*의 속성), 1960
domain (*tracemalloc.Filter*의 속성), 1961
domain (*tracemalloc.Trace*의 속성), 1963
domain_initial_dot (*http.cookiejar.Cookie*의 속성), 1574
domain_return_ok() (*http.cookiejar.CookiePolicy* 메서드), 1571
domain_specified (*http.cookiejar.Cookie*의 속성), 1574
DomainFilter (*tracemalloc* 클래스), 1960
DomainLiberal (*http.cookiejar.DefaultCookiePolicy*의 속성), 1573

DomainRFC2965Match (*http.cookiejar.DefaultCookiePolicy* 의 속성), 1573
DomainStrict (*http.cookiejar.DefaultCookiePolicy* 의 속성), 1574
DomainStrictNoDots (*http.cookiejar.DefaultCookiePolicy* 의 속성), 1573
DomainStrictNonDomain (*http.cookiejar.DefaultCookiePolicy* 의 속성), 1573
DOMEventStream (*xml.dom.pulldom* 클래스), 1442
DOMException, 1434
doModuleCleanups() (*unittest* 모듈), 1830
DomstringSizeErr, 1434
done() (*asyncio.Future* 메서드), 1182
done() (*asyncio.Task* 메서드), 1129
done() (*concurrent.futures.Future* 메서드), 1070
done() (*graphlib.TopologicalSorter* 메서드), 345
done() (*turtle* 모듈), 1706
DONT_ACCEPT_BLANKLINE (*doctest* 모듈), 1784
DONT_ACCEPT_TRUE_FOR_1 (*doctest* 모듈), 1784
dont_write_bytecode (*sys* 모듈), 1985
dont_write_bytecode (*sys.flags*의 속성), 1988
doRollover() (*logging.handlers.RotatingFileHandler* 메서드), 818
doRollover() (*logging.handlers.TimedRotatingFileHandler* 메서드), 819
DOT (*token* 모듈), 2208
dot() (*turtle* 모듈), 1688
DOTALL (*re* 모듈), 150
doublequote (*csv.Dialect*의 속성), 646
DOUBLESLASH (*token* 모듈), 2209
DOUBLESLASHEQUAL (*token* 모듈), 2209
DOUBLESTAR (*token* 모듈), 2208
DOUBLESTAREQUAL (*token* 모듈), 2209
doupdate() (*curses* 모듈), 959
down (*pdb command*), 1933
down() (*turtle* 모듈), 1692
dpgettext() (*gettext* 모듈), 1610
drain() (*asyncio.StreamWriter* 메서드), 1137
drive (*pathlib.PurePath*의 속성), 464
drop_whitespace (*textwrap.TextWrapper* 의 속성), 179
dropwhile() (*itertools* 모듈), 428
dst() (*datetime.datetime* 메서드), 235
dst() (*datetime.time* 메서드), 243
dst() (*datetime.timezone* 메서드), 251
dst() (*datetime.tzinfo* 메서드), 245
DTDHandler (*xml.sax.handler* 클래스), 1444
duck-typing (덕 타이핑), **2306**
dump() (*ast* 모듈), 2198
dump() (*json* 모듈), 1363
dump() (*marshal* 모듈), 544
dump() (*pickle* 모듈), 525
dump() (*pickle.Pickler* 메서드), 527
dump() (*plistlib* 모듈), 671
dump() (*tracemalloc.Snapshot* 메서드), 1962
dump() (*xml.etree.ElementTree* 모듈), 1415
dump_c_stack() (*faulthandler* 모듈), 1926
dump_stats() (*profile.Profile* 메서드), 1942
dump_stats() (*pstats.Stats* 메서드), 1943
dump_traceback() (*faulthandler* 모듈), 1925
dump_traceback_later() (*faulthandler* 모듈), 1926
dumps() (*json* 모듈), 1364
dumps() (*marshal* 모듈), 544
dumps() (*pickle* 모듈), 525
dumps() (*plistlib* 모듈), 671
dumps() (*xmlrpc.client* 모듈), 1582
dunder, **2306**
dup() (*os* 모듈), 700
dup() (*socket.socket* 메서드), 1231
dup2() (*os* 모듈), 700
DuplicateOptionError, 666
DuplicateSectionError, 666
--durations
    unittest 명령줄 옵션, 1803
dwFlags (*subprocess.STARTUPINFO*의 속성), 1088
DynamicClassAttribute() (*types* 모듈), 317

E

-e
    calendar 명령줄 옵션, 268
    compileall 명령줄 옵션, 2220
    idle 명령줄 옵션, 1674
    mimetypes 명령줄 옵션, 1393
    tarfile 명령줄 옵션, 637
    tokenize 명령줄 옵션, 2212
    zipfile 명령줄 옵션, 622
e (*cmath* 모듈), 363
e (*math* 모듈), 359
E2BIG (*errno* 모듈), 833
EACCES (*errno* 모듈), 833
EADDRINUSE (*errno* 모듈), 838
EADDRNOTAVAIL (*errno* 모듈), 838
EADV (*errno* 모듈), 836
EAFNOSUPPORT (*errno* 모듈), 838
EAFP, **2306**
EAGAIN (*errno* 모듈), 833
eager_task_factory() (*asyncio* 모듈), 1121
EALREADY (*errno* 모듈), 838
east_asian_width() (*unicodedata* 모듈), 182
EAUTH (*errno* 모듈), 839
EBADARCH (*errno* 모듈), 839
EBADE (*errno* 모듈), 835
EBADEXEC (*errno* 모듈), 840
EBADF (*errno* 모듈), 833
EBADFD (*errno* 모듈), 837
EBADMACHO (*errno* 모듈), 840
EBADMSG (*errno* 모듈), 836
EBADR (*errno* 모듈), 835
EBADRPC (*errno* 모듈), 840
EBADRQC (*errno* 모듈), 836
EBADSLT (*errno* 모듈), 836
EBFONT (*errno* 모듈), 836
EBUSY (*errno* 모듈), 834

ECANCELED (*errno* 모듈), 841
ECHILD (*errno* 모듈), 833
echo() (*curses* 모듈), 959
echochar() (*curses.window* 메서드), 966
ECHRNG (*errno* 모듈), 835
ECOMM (*errno* 모듈), 836
ECONNABORTED (*errno* 모듈), 838
ECONNREFUSED (*errno* 모듈), 838
ECONNRESET (*errno* 모듈), 838
EDEADLK (*errno* 모듈), 835
EDEADLOCK (*errno* 모듈), 836
EDESTADDRREQ (*errno* 모듈), 837
EDEVERR (*errno* 모듈), 840
edit() (*curses.textpad.Textbox* 메서드), 982
EDOM (*errno* 모듈), 834
EDOTDOT (*errno* 모듈), 836
EDQUOT (*errno* 모듈), 839
EEXIST (*errno* 모듈), 834
EFAULT (*errno* 모듈), 834
EFBIG (*errno* 모듈), 834
EFD_CLOEXEC (*os* 모듈), 734
EFD_NONBLOCK (*os* 모듈), 734
EFD_SEMAPHORE (*os* 모듈), 734
effective() (*bdb* 모듈), 1924
EFTYPE (*errno* 모듈), 840
ehlo() (*smtplib.SMTP* 메서드), 1537
ehlo_or_helo_if_needed() (*smtplib.SMTP* 메서드), 1537
EHOSTDOWN (*errno* 모듈), 838
EHOSTUNREACH (*errno* 모듈), 838
EHWPOISON (*errno* 모듈), 838
EIDRM (*errno* 모듈), 835
EILSEQ (*errno* 모듈), 837
EINPROGRESS (*errno* 모듈), 839
EINTR (*errno* 모듈), 833
EINVAL (*errno* 모듈), 834
EIO (*errno* 모듈), 833
EISCONN (*errno* 모듈), 838
EISDIR (*errno* 모듈), 834
EISNAM (*errno* 모듈), 839
EJECT (*enum.FlagBoundary*의 속성), 340
EKEYEXPIRED (*errno* 모듈), 839
EKEYREJECTED (*errno* 모듈), 839
EKEYREVOKED (*errno* 모듈), 839
EL2HLT (*errno* 모듈), 835
EL2NSYNC (*errno* 모듈), 835
EL3HLT (*errno* 모듈), 835
EL3RST (*errno* 모듈), 835
Element (*xml.etree.ElementTree* 클래스), 1419
element_create() (*tkinter.ttk.Style* 메서드), 1663
element_names() (*tkinter.ttk.Style* 메서드), 1664
element_options() (*tkinter.ttk.Style* 메서드), 1665
ElementDeclHandler() (*xml.parsers.expat.xmlparser* 메서드), 1458
elements() (*collections.Counter* 메서드), 272
ElementTree (*xml.etree.ElementTree* 클래스), 1421
ELIBACC (*errno* 모듈), 837
ELIBBAD (*errno* 모듈), 837
ELIBEXEC (*errno* 모듈), 837
ELIBMAX (*errno* 모듈), 837
ELIBSCN (*errno* 모듈), 837
ELLIPSIS (*doctest* 모듈), 1784
ELLIPSIS (*token* 모듈), 2209
Ellipsis (내장 변수), 36
EllipsisType (*types* 모듈), 314
ELNRNG (*errno* 모듈), 835
ELOCKUNMAPPED (*errno* 모듈), 839
ELOOP (*errno* 모듈), 835
EM (*curses.ascii* 모듈), 985
email
    module, 1303
email.charset
    module, 1354
email.contentmanager
    module, 1332
email.encoders
    module, 1356
email.errors
    module, 1325
email.generator
    module, 1315
email.header
    module, 1351
email.headerregistry
    module, 1326
email.iterators
    module, 1359
email.message
    module, 1304
EmailMessage (*email.message* 클래스), 1304
email.mime
    module, 1349
email.mime.application
    module, 1350
email.mime.audio
    module, 1350
email.mime.base
    module, 1349
email.mime.image
    module, 1350
email.mime.message
    module, 1351
email.mime.multipart
    module, 1349
email.mime.nonmultipart
    module, 1349
email.mime.text
    module, 1351
email.parser
    module, 1312
email.policy
    module, 1318
EmailPolicy (*email.policy* 클래스), 1322
email.utils
    module, 1357
Emax (*decimal.Context*의 속성), 378

EMEDIUMTYPE (*errno* 모듈), 839
EMFILE (*errno* 모듈), 834
Emin (*decimal.Context*의 속성), 378
emit() (*logging.FileHandler* 메서드), 815
emit() (*logging.Handler* 메서드), 791
emit() (*logging.handlers.BufferingHandler* 메서드), 824
emit() (*logging.handlers.DatagramHandler* 메서드), 820
emit() (*logging.handlers.HTTPHandler* 메서드), 825
emit() (*logging.handlers.NTEventLogHandler* 메서드), 823
emit() (*logging.handlers.QueueHandler* 메서드), 826
emit() (*logging.handlers.RotatingFileHandler* 메서드), 818
emit() (*logging.handlers.SMTPHandler* 메서드), 824
emit() (*logging.handlers.SocketHandler* 메서드), 819
emit() (*logging.handlers.SysLogHandler* 메서드), 821
emit() (*logging.handlers.TimedRotatingFileHandler* 메서드), 819
emit() (*logging.handlers.WatchedFileHandler* 메서드), 816
emit() (*logging.NullHandler* 메서드), 815
emit() (*logging.StreamHandler* 메서드), 814
EMLINK (*errno* 모듈), 834
Empty, 1098
empty (*inspect.Parameter*의 속성), 2089
empty (*inspect.Signature*의 속성), 2088
empty() (*asyncio.Queue* 메서드), 1151
empty() (*multiprocessing.Queue* 메서드), 1024
empty() (*multiprocessing.SimpleQueue* 메서드), 1025
empty() (*queue.Queue* 메서드), 1099
empty() (*queue.SimpleQueue* 메서드), 1100
empty() (*sched.scheduler* 메서드), 1097
EMPTY_NAMESPACE (*xml.dom* 모듈), 1427
emptyline() (*cmd.Cmd* 메서드), 989
emscripten_version (*sys._emscripten_info*의 속성), 1985
EMSGSIZE (*errno* 모듈), 837
EMULTIHOP (*errno* 모듈), 836
enable (*pdb command*), 1934
enable() (*bdb.Breakpoint* 메서드), 1919
enable() (*faulthandler* 모듈), 1926
enable() (*gc* 모듈), 2077
enable() (*imaplib.IMAP4* 메서드), 1529
enable() (*profile.Profile* 메서드), 1942
enable_callback_tracebacks() (*sqlite3* 모듈), 555
enable_interspersed_args() (*optparse.OptionParser* 메서드), 945
enable_load_extension() (*sqlite3.Connection* 메서드), 562
enable_long_distance_matching (*compression.zstd.CompressionParameter* 의 속성), 590
enable_traversal() (*tkinter.ttk.Notebook* 메서드), 1655
ENABLE_USER_SITE (*site* 모듈), 2110
enabled (*bdb.Breakpoint*의 속성), 1920
EnableReflectionKey() (*winreg* 모듈), 2255
ENAMETOOLONG (*errno* 모듈), 835
ENAVAIL (*errno* 모듈), 839
enclose() (*curses.window* 메서드), 966
encode
    코덱, 198
encode (*codecs.CodecInfo*의 속성), 199
encode() (*base64* 모듈), 1397
encode() (*codecs* 모듈), 199
encode() (*codecs.Codec* 메서드), 204
encode() (*codecs.IncrementalEncoder* 메서드), 205
encode() (*email.header.Header* 메서드), 1353
encode() (*json.JSONEncoder* 메서드), 1367
encode() (*quopri* 모듈), 1400
encode() (*str* 메서드), 55
encode() (*xmlrpc.client.Binary* 메서드), 1579
encode() (*xmlrpc.client.DateTime* 메서드), 1579
encode_7or8bit() (*email.encoders* 모듈), 1357
encode_base64() (*email.encoders* 모듈), 1357
encode_noop() (*email.encoders* 모듈), 1357
encode_quopri() (*email.encoders* 모듈), 1357
encode_rfc2231() (*email.utils* 모듈), 1359
encodebytes() (*base64* 모듈), 1397
EncodedFile() (*codecs* 모듈), 201
encodePriority() (*logging.handlers.SysLogHandler* 메서드), 822
encodestring() (*quopri* 모듈), 1400
--encoding
    calendar 명령줄 옵션, 268
encoding (*curses.window*의 속성), 967
encoding (*io.TextIOBase*의 속성), 767
ENCODING (*tarfile* 모듈), 626
ENCODING (*token* 모듈), 2206
encoding (*UnicodeError*의 속성), 118
encodings
    module, 214
encodings_map (*mimetypes* 모듈), 1391
encodings_map (*mimetypes.MimeTypes* 의 속성), 1392
encodings.idna
    module, 215
encodings.mbcs
    module, 216
encodings.utf_8_sig
    module, 216
EncodingWarning, 120
end (*UnicodeError*의 속성), 118
end() (*re.Match* 메서드), 158
end() (*xml.etree.ElementTree.TreeBuilder* 메서드), 1423
END_ASYNC_FOR (*opcode*), 2231
end_col_offset (*ast.AST*의 속성), 2165
end_colno (*traceback.FrameSummary*의 속성), 2071
end_fill() (*turtle* 모듈), 1695
END_FOR (*opcode*), 2229
end_headers() (*http.server.BaseHTTPRequestHandler* 메서드), 1559

end_lineno (*ast.AST*의 속성), 2165
end_lineno (*SyntaxError*의 속성), 117
end_lineno (*traceback.FrameSummary*의 속성), 2070
end_lineno (*traceback.TracebackException*의 속성), 2069
end_ns() (*xml.etree.ElementTree.TreeBuilder* 메서드), 1424
end_offset (*SyntaxError*의 속성), 117
end_offset (*traceback.TracebackException*의 속성), 2069
end_poly() (*turtle* 모듈), 1700
END_SEND (*opcode*), 2229
endCDATA() (*xml.sax.handler.LexicalHandler* 메서드), 1449
EndCdataSectionHandler() (*xml.parsers.expat.xmlparser* 메서드), 1459
EndDoctypeDeclHandler() (*xml.parsers.expat.xmlparser* 메서드), 1458
endDocument() (*xml.sax.handler.ContentHandler* 메서드), 1446
endDTD() (*xml.sax.handler.LexicalHandler* 메서드), 1449
endElement() (*xml.sax.handler.ContentHandler* 메서드), 1447
EndElementHandler() (*xml.parsers.expat.xmlparser* 메서드), 1458
endElementNS() (*xml.sax.handler.ContentHandler* 메서드), 1447
endheaders() (*http.client.HTTPConnection* 메서드), 1514
ENDMARKER (*token* 모듈), 2206
EndNamespaceDeclHandler() (*xml.parsers.expat.xmlparser* 메서드), 1459
endpos (*re.Match*의 속성), 159
endPrefixMapping() (*xml.sax.handler.ContentHandler* 메서드), 1447
endswith() (*bytearray* 메서드), 72
endswith() (*bytes* 메서드), 72
endswith() (*str* 메서드), 56
endwin() (*curses* 모듈), 959
ENEEDAUTH (*errno* 모듈), 840
ENETDOWN (*errno* 모듈), 838
ENETRESET (*errno* 모듈), 838
ENETUNREACH (*errno* 모듈), 838
ENFILE (*errno* 모듈), 834
ENOANO (*errno* 모듈), 835
ENOATTR (*errno* 모듈), 840
ENOBUFS (*errno* 모듈), 838
ENOCSI (*errno* 모듈), 835
ENODATA (*errno* 모듈), 836
ENODEV (*errno* 모듈), 834
ENOENT (*errno* 모듈), 833
ENOEXEC (*errno* 모듈), 833
ENOKEY (*errno* 모듈), 839
ENOLCK (*errno* 모듈), 835
ENOLINK (*errno* 모듈), 836
ENOMEDIUM (*errno* 모듈), 839
ENOMEM (*errno* 모듈), 833
ENOMSG (*errno* 모듈), 835
ENONET (*errno* 모듈), 836
ENOPKG (*errno* 모듈), 836
ENOPOLICY (*errno* 모듈), 840
ENOPROTOOPT (*errno* 모듈), 837
ENOSPC (*errno* 모듈), 834
ENOSR (*errno* 모듈), 836
ENOSTR (*errno* 모듈), 836
ENOSYS (*errno* 모듈), 835
ENOTACTIVE (*errno* 모듈), 839
ENOTBLK (*errno* 모듈), 834
ENOTCAPABLE (*errno* 모듈), 841
ENOTCONN (*errno* 모듈), 838
ENOTDIR (*errno* 모듈), 834
ENOTEMPTY (*errno* 모듈), 835
ENOTNAM (*errno* 모듈), 839
ENOTRECOVERABLE (*errno* 모듈), 841
ENOTSOCK (*errno* 모듈), 837
ENOTSUP (*errno* 모듈), 837
ENOTTY (*errno* 모듈), 834
ENOTUNIQ (*errno* 모듈), 836
ENQ (*curses.ascii* 모듈), 984
enqueue() (*logging.handlers.QueueHandler* 메서드), 826
enqueue_sentinel() (*logging.handlers.QueueListener* 메서드), 828
ensure_directories() (*venv.EnvBuilder* 메서드), 1971
ensure_future() (*asyncio* 모듈), 1181
ensurepip
    module, 1965
ensurepip 명령줄 옵션
    --altinstall, 1966
    --default-pip, 1966
    --root, 1966
    --user, 1966
enter() (*sched.scheduler* 메서드), 1097
enter_async_context() (*contextlib.AsyncExitStack* 메서드), 2052
enter_context() (*contextlib.ExitStack* 메서드), 2052
enterabs() (*sched.scheduler* 메서드), 1097
enterAsyncContext() (*unittest.IsolatedAsyncioTestCase* 메서드), 1820
enterClassContext() (*unittest.TestCase*의 클래스 메서드), 1819
enterContext() (*unittest.TestCase* 메서드), 1819
enterModuleContext() (*unittest* 모듈), 1830
entities (*xml.dom.DocumentType*의 속성), 1431
EntityDeclHandler() (*xml.parsers.expat.xmlparser* 메서드), 1459
entitydefs (*html.entities* 모듈), 1406
EntityResolver (*xml.sax.handler* 클래스), 1444
entry_points() (*importlib.metadata* 모듈), 2152
EntryPoint (*importlib.metadata* 클래스), 2152
EntryPoints (*importlib.metadata* 클래스), 2152
enum

module, 327
Enum (*enum* 클래스), 331
enum_certificates() (*ssl* 모듈), 1246
enum_crls() (*ssl* 모듈), 1247
EnumCheck (*enum* 클래스), 338
EnumDict (*enum* 클래스), 340
enumerate()
built-in function, 14
enumerate() (*threading* 모듈), 998
EnumKey() (*winreg* 모듈), 2251
EnumType (*enum* 클래스), 330
EnumValue() (*winreg* 모듈), 2252
ENV_DIR
venv 명령줄 옵션, 1968
EnvBuilder (*venv* 클래스), 1970
environ (*os* 모듈), 691
environ (*posix* 모듈), 2269
environb (*os* 모듈), 692
environment variables
deleting, 698
setting, 695
EnvironmentError, 118
EnvironmentVarGuard (*test.support.os_helper* 클래스), 1909
ENXIO (*errno* 모듈), 833
eof (*bz2.BZ2Decompressor*의 속성), 604
eof (*compression.zstd.ZstdDecompressor*의 속성), 587
eof (*lzma.LZMADecompressor*의 속성), 609
eof (*shlex.shlex*의 속성), 2266
eof (*ssl.MemoryBIO*의 속성), 1274
eof (*zlib.Decompress*의 속성), 597
eof_received() (*asyncio.BufferedProtocol* 메서드), 1191
eof_received() (*asyncio.Protocol* 메서드), 1190
EOFError, 113
EOPNOTSUPP (*errno* 모듈), 837
EOT (*curses.ascii* 모듈), 984
EOVERFLOW (*errno* 모듈), 836
EOWNERDEAD (*errno* 모듈), 841
EPERM (*errno* 모듈), 833
EPFNOSUPPORT (*errno* 모듈), 837
epilogue (*email.message.EmailMessage*의 속성), 1312
epilogue (*email.message.Message*의 속성), 1349
EPIPE (*errno* 모듈), 834
epoll() (*select* 모듈), 1278
EpollSelector (*selectors* 클래스), 1287
EPROCLIM (*errno* 모듈), 840
EPROCUNAVAIL (*errno* 모듈), 840
EPROGMISMATCH (*errno* 모듈), 840
EPROGUNAVAIL (*errno* 모듈), 840
EPROTO (*errno* 모듈), 836
EPROTONOSUPPORT (*errno* 모듈), 837
EPROTOTYPE (*errno* 모듈), 837
epsilon (*sys.float_info*의 속성), 1990
EPWROFF (*errno* 모듈), 840
Eq (*ast* 클래스), 2172
eq() (*operator* 모듈), 451
EQEQUAL (*token* 모듈), 2208
EQFULL (*errno* 모듈), 839
EQUAL (*token* 모듈), 2207
ERA (*locale* 모듈), 1620
ERA_D_FMT (*locale* 모듈), 1620
ERA_D_T_FMT (*locale* 모듈), 1620
ERA_T_FMT (*locale* 모듈), 1620
ERANGE (*errno* 모듈), 834
erase() (*curses.window* 메서드), 967
erasechar() (*curses* 모듈), 959
EREMCHG (*errno* 모듈), 837
EREMOTE (*errno* 모듈), 836
EREMOTEIO (*errno* 모듈), 839
ERESTART (*errno* 모듈), 837
erf() (*math* 모듈), 358
erfc() (*math* 모듈), 358
ERFKILL (*errno* 모듈), 839
EROFS (*errno* 모듈), 834
ERPCMISMATCH (*errno* 모듈), 841
ERR (*curses* 모듈), 971
errcheck (*ctypes._CFuncPtr*의 속성), 864
errcode (*xmlrpc.client.ProtocolError*의 속성), 1581
errmsg (*xmlrpc.client.ProtocolError*의 속성), 1581
errno
module, 833
모듈, 115
errno (*OSError*의 속성), 115
Error, 318, 516, 572, 645, 666, 1388, 1400, 1466, 1605, 1617
error, 192, 545, 547, 549, 550, 594, 689, 957, 1105, 1216, 1277, 1455, 2278, 2290
ERROR (*logging* 모듈), 790
ERROR (*tkinter.messagebox* 모듈), 1646
error() (*argparse.ArgumentParser* 메서드), 911
error() (*logging* 모듈), 798
error() (*logging.Logger* 메서드), 788
error() (*urllib.request.OpenerDirector* 메서드), 1486
error() (*xml.sax.handler.ErrorHandler* 메서드), 1448
error_body (*wsgiref.handlers.BaseHandler*의 속성), 1476
error_content_type (*http.server.BaseHTTPRequestHandler*의 속성), 1558
error_headers (*wsgiref.handlers.BaseHandler*의 속성), 1475
error_leader() (*shlex.shlex* 메서드), 2265
error_message_format (*http.server.BaseHTTPRequestHandler*의 속성), 1558
error_output() (*wsgiref.handlers.BaseHandler* 메서드), 1475
error_perm, 1523
error_proto, 1523, 1525
error_received() (*asyncio.DatagramProtocol* 메서드), 1191
error_reply, 1523
error_status (*wsgiref.handlers.BaseHandler*의 속성), 1475
error_temp, 1523

ErrorByteIndex (*xml.parsers.expat.xmlparser* 의 속성), 1457
errorcode (*errno* 모듈), 833
ErrorCode (*xml.parsers.expat.xmlparser* 의 속성), 1457
ErrorColumnNumber (*xml.parsers.expat.xmlparser* 의 속성), 1457
ErrorHandler (*xml.sax.handler* 클래스), 1444
errorlevel (*tarfile.TarFile* 의 속성), 629
ErrorLineNumber (*xml.parsers.expat.xmlparser* 의 속성), 1457
errors (*io.TextIOBase* 의 속성), 767
errors (*unittest.TestLoader* 의 속성), 1822
errors (*unittest.TestResult* 의 속성), 1824
ErrorStream (*wsgiref.types* 클래스), 1477
ErrorString() (*xml.parsers.expat* 모듈), 1455
ERRORTOKEN (*token* 모듈), 2206
ESC (*curses.ascii* 모듈), 985
escape (*shlex.shlex* 의 속성), 2266
escape() (*glob* 모듈), 507
escape() (*html* 모듈), 1401
escape() (*re* 모듈), 154
escape() (*xml.sax.saxutils* 모듈), 1449
escapechar (*csv.Dialect* 의 속성), 646
escapedquotes (*shlex.shlex* 의 속성), 2266
ESHLIBVERS (*errno* 모듈), 841
ESHUTDOWN (*errno* 모듈), 838
ESOCKTNOSUPPORT (*errno* 모듈), 837
ESPIPE (*errno* 모듈), 834
ESRCH (*errno* 모듈), 833
ESRMNT (*errno* 모듈), 836
ESTALE (*errno* 모듈), 839
ESTRPIPE (*errno* 모듈), 837
ETB (*curses.ascii* 모듈), 985
ETH_P_ALL (*socket* 모듈), 1219
ETHERTYPE_ARP (*socket* 모듈), 1222
ETHERTYPE_IP (*socket* 모듈), 1222
ETHERTYPE_IPV6 (*socket* 모듈), 1222
ETHERTYPE_VLAN (*socket* 모듈), 1222
ETIME (*errno* 모듈), 836
ETIMEDOUT (*errno* 모듈), 838
Etiny() (*decimal.Context* 메서드), 379
ETOOMANYREFS (*errno* 모듈), 838
Etop() (*decimal.Context* 메서드), 379
ETX (*curses.ascii* 모듈), 984
ETXTBSY (*errno* 모듈), 834
EUCLEAN (*errno* 모듈), 839
EUNATCH (*errno* 모듈), 835
EUSERS (*errno* 모듈), 837
eval
    built-in function, 106
    내장 함수, 320, 321
eval()
    built-in function, 14
evaluate function, **2306**
evaluate() (*annotationlib.ForwardRef* 메서드), 2102
evaluate_bound() (*typing.TypeVar* 메서드), 1744
evaluate_constraints() (*typing.TypeVar* 메서드), 1744
evaluate_default() (*typing.ParamSpec* 메서드), 1748
evaluate_default() (*typing.TypeVar* 메서드), 1744
evaluate_default() (*typing.TypeVarTuple* 메서드), 1746
evaluate_forward_ref() (*typing* 모듈), 1765
evaluate_value() (*typing.TypeAliasType* 메서드), 1749
Event (*asyncio* 클래스), 1141
Event (*multiprocessing* 클래스), 1030
Event (*threading* 클래스), 1009
Event() (*multiprocessing.managers.SyncManager* 메서드), 1037
EVENT_READ (*selectors* 모듈), 1285
EVENT_WRITE (*selectors* 모듈), 1285
eventfd() (*os* 모듈), 733
eventfd_read() (*os* 모듈), 734
eventfd_write() (*os* 모듈), 734
EventLoop (*asyncio* 클래스), 1178
events (*selectors.SelectorKey* 의 속성), 1285
EWOULDBLOCK (*errno* 모듈), 835
EX_CANTCREAT (*os* 모듈), 741
EX_CONFIG (*os* 모듈), 741
EX_DATAERR (*os* 모듈), 740
EX_IOERR (*os* 모듈), 741
EX_NOHOST (*os* 모듈), 740
EX_NOINPUT (*os* 모듈), 740
EX_NOPERM (*os* 모듈), 741
EX_NOTFOUND (*os* 모듈), 741
EX_NOUSER (*os* 모듈), 740
EX_OK (*os* 모듈), 740
EX_OSERR (*os* 모듈), 741
EX_OSFILE (*os* 모듈), 741
EX_PROTOCOL (*os* 모듈), 741
EX_SOFTWARE (*os* 모듈), 740
EX_TEMPFAIL (*os* 모듈), 741
EX_UNAVAILABLE (*os* 모듈), 740
EX_USAGE (*os* 모듈), 740
--exact
    tokenize 명령줄 옵션, 2212
EXACT_TOKEN_TYPES (*token* 모듈), 2209
Example (*doctest* 클래스), 1792
example (*doctest.DocTestFailure* 의 속성), 1798
example (*doctest.UnexpectedException* 의 속성), 1799
examples (*doctest.DocTest* 의 속성), 1792
exc_info (*doctest.UnexpectedException* 의 속성), 1799
exc_info() (*sys* 모듈), 1986
exc_msg (*doctest.Example* 의 속성), 1792
exc_type (*traceback.TracebackException* 의 속성), 2068
exc_type_str (*traceback.TracebackException* 의 속성), 2068
excel (*csv* 클래스), 644
excel_tab (*csv* 클래스), 644
except
    문장, 111
ExceptHandler (*ast* 클래스), 2182
excepthook() (*sys* 모듈), 1985

excepthook() (*threading* 모듈), 997
Exception, 113
EXCEPTION (*_tkinter* 모듈), 1640
exception() (*asyncio.Future* 메서드), 1183
exception() (*asyncio.Task* 메서드), 1129
exception() (*concurrent.futures.Future* 메서드), 1070
exception() (*logging* 모듈), 798
exception() (*logging.Logger* 메서드), 788
exception() (*sys* 모듈), 1986
EXCEPTION_HANDLED (*monitoring event*), 2009
ExceptionGroup, 121
exceptions (*BaseExceptionGroup*의 속성), 121
exceptions (*pdb command*), 1938
exceptions (*traceback.TracebackException*의 속성), 2068
excinfo (*concurrent.interpreters.ExecutionFailed*의 속성), 1075
EXCLAMATION (*token* 모듈), 2209
EXDEV (*errno* 모듈), 834
exec
    built-in function, 15, 106
exec()
    built-in function, 15
exec() (*concurrent.interpreters.Interpreter* 메서드), 1075
exec_module() (*importlib.abc.InspectLoader* 메서드), 2131
exec_module() (*importlib.abc.Loader* 메서드), 2129
exec_module() (*importlib.abc.SourceLoader* 메서드), 2132
exec_module() (*importlib.machinery.ExtensionFileLoader* 메서드), 2138
exec_module() (*zipimport.zipimporter* 메서드), 2118
exec_prefix (*sys* 모듈), 1986
execl() (*os* 모듈), 739
execle() (*os* 모듈), 739
execlp() (*os* 모듈), 739
execlpe() (*os* 모듈), 739
executable (*sys* 모듈), 1986
execute() (*sqlite3.Connection* 메서드), 558
execute() (*sqlite3.Cursor* 메서드), 568
executemany() (*sqlite3.Connection* 메서드), 558
executemany() (*sqlite3.Cursor* 메서드), 568
executescript() (*sqlite3.Connection* 메서드), 558
executescript() (*sqlite3.Cursor* 메서드), 569
ExecutionFailed, 1072, 1075
ExecutionLoader (*importlib.abc* 클래스), 2131
Executor (*concurrent.futures* 클래스), 1064
execv() (*os* 모듈), 739
execve() (*os* 모듈), 739
execvp() (*os* 모듈), 739
execvpe() (*os* 모듈), 739
EXFULL (*errno* 모듈), 835
exists() (*os.path* 모듈), 487
exists() (*pathlib.Path* 메서드), 474
exists() (*pathlib.types.PathInfo* 메서드), 485
exists() (*tkinter.ttk.Treeview* 메서드), 1659
exists() (*zipfile.Path* 메서드), 618
exit (내장 변수), 36
exit() (*_thread* 모듈), 1106
exit() (*argparse.ArgumentParser* 메서드), 911
exit() (*sys* 모듈), 1987
exitcode (*multiprocessing.Process*의 속성), 1021
exitonclick() (*turtle* 모듈), 1708
ExitStack (*contextlib* 클래스), 2051
exp() (*cmath* 모듈), 361
exp() (*decimal.Context* 메서드), 380
exp() (*decimal.Decimal* 메서드), 371
exp() (*math* 모듈), 355
exp2() (*math* 모듈), 355
expand() (*re.Match* 메서드), 156
expand_tabs (*textwrap.TextWrapper*의 속성), 179
ExpandEnvironmentStrings() (*winreg* 모듈), 2252
expandNode() (*xml.dom.pulldom.DOMEventStream* 메서드), 1442
expandtabs() (*bytearray* 메서드), 77
expandtabs() (*bytes* 메서드), 77
expandtabs() (*str* 메서드), 56
expanduser() (*os.path* 모듈), 487
expanduser() (*pathlib.Path* 메서드), 473
expandvars() (*os.path* 모듈), 488
Expat, 1454
ExpatError, 1455
expected (*asyncio.IncompleteReadError*의 속성), 1154
expectedFailure() (*unittest* 모듈), 1808
expectedFailures (*unittest.TestResult*의 속성), 1825
expired() (*asyncio.Timeout* 메서드), 1123
expires (*http.cookiejar.Cookie*의 속성), 1574
expires (*http.cookies.Morsel*의 속성), 1565
exploded (*ipaddress.IPv4Address*의 속성), 1591
exploded (*ipaddress.IPv4Network*의 속성), 1596
exploded (*ipaddress.IPv6Address*의 속성), 1593
exploded (*ipaddress.IPv6Network*의 속성), 1599
expm1() (*math* 모듈), 355
expovariate() (*random* 모듈), 400
Expr (*ast* 클래스), 2170
Expression (*ast* 클래스), 2166
expression (*string.templatelib.Interpolation*의 속성), 140
expression (표현식), **2306**
expunge() (*imaplib.IMAP4* 메서드), 1529
extend() (*array.array* 메서드), 302
extend() (*collections.deque* 메서드), 275
extend() (*sequence* 메서드), 49
extend() (*xml.etree.ElementTree.Element* 메서드), 1420
extend_path() (*pkgutil* 모듈), 2119
EXTENDED_ARG (*opcode*), 2241
ExtendedContext (*decimal* 모듈), 377
ExtendedInterpolation (*configparser* 클래스), 654
extendleft() (*collections.deque* 메서드), 275
--extension
    mimetypes 명령줄 옵션, 1393
extension module (확장 모듈), **2306**

EXTENSION_SUFFIXES (*importlib.machinery* 모듈), 2135
ExtensionFileLoader (*importlib.machinery* 클래스), 2138
extensions_map (*http.server.SimpleHTTPRequestHandler*의 속성), 1560
external_attr (*zipfile.ZipInfo*의 속성), 621
ExternalClashError, 1388
ExternalEntityParserCreate() (*xml.parsers.expat.xmlparser* 메서드), 1456
ExternalEntityRefHandler() (*xml.parsers.expat.xmlparser* 메서드), 1459
extra (*zipfile.ZipInfo*의 속성), 621
--extract
    tarfile 명령줄 옵션, 637
    zipfile 명령줄 옵션, 622
extract() (*tarfile.TarFile* 메서드), 629
extract() (*traceback.StackSummary*의 클래스 메서드), 2069
extract() (*zipfile.ZipFile* 메서드), 615
extract_cookies() (*http.cookiejar.CookieJar* 메서드), 1569
extract_stack() (*traceback* 모듈), 2066
extract_tb() (*traceback* 모듈), 2066
extract_version (*zipfile.ZipInfo*의 속성), 621
extractall() (*tarfile.TarFile* 메서드), 628
extractall() (*zipfile.ZipFile* 메서드), 616
ExtractError, 625
extractfile() (*tarfile.TarFile* 메서드), 629
extraction_filter (*tarfile.TarFile*의 속성), 630
extsep (*os* 모듈), 755

F

-f
    calendar 명령줄 옵션, 268
    compileall 명령줄 옵션, 2219
    doctest 명령줄 옵션, 1780
    random 명령줄 옵션, 406
    trace 명령줄 옵션, 1953
    unittest 명령줄 옵션, 1803
f-string (*f*-문자열), **2306**
f-strings, **2306**
F_CONTIGUOUS (*inspect.BufferFlags*의 속성), 2099
f_contiguous (*memoryview*의 속성), 91
F_LOCK (*os* 모듈), 702
F_OK (*os* 모듈), 713
F_TEST (*os* 모듈), 702
F_TLOCK (*os* 모듈), 702
F_ULOCK (*os* 모듈), 702
fabs() (*math* 모듈), 353
factorial() (*math* 모듈), 352
factory() (*importlib.util.LazyLoader*의 클래스 메서드), 2142
fail() (*unittest.TestCase* 메서드), 1818
FAIL_FAST (*doctest* 모듈), 1786
failed (*doctest.TestResults*의 속성), 1794
--failfast
    unittest 명령줄 옵션, 1803
--fail-fast
    doctest 명령줄 옵션, 1780
failfast (*unittest.TestResult*의 속성), 1825
failureException (*unittest.TestCase*의 속성), 1818
failures (*doctest.DocTestRunner*의 속성), 1795
failures (*unittest.TestResult*의 속성), 1825
FakePath (*test.support.os_helper* 클래스), 1909
False, 37, 46
false, 37
False (*Built-in object*), 37
False (내장 변수), 35
families() (*tkinter.font* 모듈), 1641
family (*socket.socket*의 속성), 1237
--fast
    gzip 명령줄 옵션, 601
fast (*compression.zstd.Strategy*의 속성), 592
fast (*pickle.Pickler*의 속성), 527
fatalError() (*xml.sax.handler.ErrorHandler* 메서드), 1449
Fault (*xmlrpc.client* 클래스), 1580
faultCode (*xmlrpc.client.Fault*의 속성), 1580
faulthandler
    module, 1925
faultString (*xmlrpc.client.Fault*의 속성), 1580
fchdir() (*os* 모듈), 715
fchmod() (*os* 모듈), 700
fchown() (*os* 모듈), 701
fcntl
    module, 2275
fcntl() (*fcntl* 모듈), 2276
fd (*selectors.SelectorKey*의 속성), 1285
fd() (*turtle* 모듈), 1685
fd_count() (*test.support.os_helper* 모듈), 1909
fdatasync() (*os* 모듈), 701
fdopen() (*os* 모듈), 699
feature_external_ges (*xml.sax.handler* 모듈), 1445
feature_external_pes (*xml.sax.handler* 모듈), 1445
feature_namespace_prefixes (*xml.sax.handler* 모듈), 1445
feature_namespaces (*xml.sax.handler* 모듈), 1445
feature_string_interning (*xml.sax.handler* 모듈), 1445
feature_validation (*xml.sax.handler* 모듈), 1445
--feature-version
    ast 명령줄 옵션, 2200
FEBRUARY (*calendar* 모듈), 266
feed() (*email.parser.BytesFeedParser* 메서드), 1313
feed() (*html.parser.HTMLParser* 메서드), 1403
feed() (*xml.etree.ElementTree.XMLParser* 메서드), 1424
feed() (*xml.etree.ElementTree.XMLPullParser* 메서드), 1425
feed() (*xml.sax.xmlreader.IncrementalParser* 메서드), 1452
feed_eof() (*asyncio.StreamReader* 메서드), 1135
FeedParser (*email.parser* 클래스), 1313

fetch() (*imaplib.IMAP4* 메서드), 1529
fetchall() (*sqlite3.Cursor* 메서드), 569
fetchmany() (*sqlite3.Cursor* 메서드), 569
fetchone() (*sqlite3.Cursor* 메서드), 569
FF (*curses.ascii* 모듈), 984
fflags (*select.kevent*의 속성), 1283
Field (*dataclasses* 클래스), 2037
field() (*dataclasses* 모듈), 2036
field_size_limit() (*csv* 모듈), 643
fieldnames (*csv.DictReader*의 속성), 647
fields (*uuid.UUID*의 속성), 1542
fields() (*dataclasses* 모듈), 2037
FIFOTYPE (*tarfile* 모듈), 626
file
    compileall 명령줄 옵션, 2219
    gzip 명령줄 옵션, 601
    modes, 23
--file
    trace 명령줄 옵션, 1953
file (*bdb.Breakpoint*의 속성), 1920
file (*pyclbr.Class*의 속성), 2216
file (*pyclbr.Function*의 속성), 2215
file object
    open() built-in function, 23
file object (파일 객체), **2306**
file-like object (파일류 객체), **2306**
FILE_ATTRIBUTE_ARCHIVE (*stat* 모듈), 498
FILE_ATTRIBUTE_COMPRESSED (*stat* 모듈), 498
FILE_ATTRIBUTE_DEVICE (*stat* 모듈), 498
FILE_ATTRIBUTE_DIRECTORY (*stat* 모듈), 498
FILE_ATTRIBUTE_ENCRYPTED (*stat* 모듈), 498
FILE_ATTRIBUTE_HIDDEN (*stat* 모듈), 498
FILE_ATTRIBUTE_INTEGRITY_STREAM (*stat* 모듈), 498
FILE_ATTRIBUTE_NO_SCRUB_DATA (*stat* 모듈), 498
FILE_ATTRIBUTE_NORMAL (*stat* 모듈), 498
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED (*stat* 모듈), 498
FILE_ATTRIBUTE_OFFLINE (*stat* 모듈), 498
FILE_ATTRIBUTE_READONLY (*stat* 모듈), 498
FILE_ATTRIBUTE_REPARSE_POINT (*stat* 모듈), 498
FILE_ATTRIBUTE_SPARSE_FILE (*stat* 모듈), 498
FILE_ATTRIBUTE_SYSTEM (*stat* 모듈), 498
FILE_ATTRIBUTE_TEMPORARY (*stat* 모듈), 498
FILE_ATTRIBUTE_VIRTUAL (*stat* 모듈), 498
file_digest() (*hashlib* 모듈), 676
file_open() (*urllib.request.FileHandler* 메서드), 1490
file_size (*zipfile.ZipInfo*의 속성), 622
filecmp
    module, 499
fileConfig() (*logging.config* 모듈), 803
FileCookieJar (*http.cookiejar* 클래스), 1567
FileDialog (*tkinter.filedialog* 클래스), 1643
FileExistsError, 119
FileFinder (*importlib.machinery* 클래스), 2136
FileHandler (*logging* 클래스), 815
FileHandler (*urllib.request* 클래스), 1483
fileinput
    module, 954
FileInput (*fileinput* 클래스), 956
FileIO (*io* 클래스), 764
filelineno() (*fileinput* 모듈), 955
FileLoader (*importlib.abc* 클래스), 2131
filemode() (*stat* 모듈), 494
filename (*doctest.DocTest*의 속성), 1792
filename (*http.cookiejar.FileCookieJar*의 속성), 1570
filename (*inspect.FrameInfo*의 속성), 2094
filename (*inspect.Traceback*의 속성), 2095
filename (*netrc.NetrcParseError*의 속성), 669
filename (*OSError*의 속성), 115
filename (*SyntaxError*의 속성), 116
filename (*traceback.FrameSummary*의 속성), 2070
filename (*traceback.TracebackException*의 속성), 2068
filename (*tracemalloc.Frame*의 속성), 1961
filename (*zipfile.ZipFile*의 속성), 617
filename (*zipfile.ZipInfo*의 속성), 620
filename() (*fileinput* 모듈), 955
filename2 (*OSError*의 속성), 115
filename_only (*tabnanny* 모듈), 2214
filename_pattern (*tracemalloc.Filter*의 속성), 1961
fileno() (*bz2.BZ2File* 메서드), 602
fileno() (*fileinput* 모듈), 955
fileno() (*http.client.HTTPResponse* 메서드), 1515
fileno() (*io.IOBase* 메서드), 761
fileno() (*multiprocessing.connection.Connection* 메서드), 1028
fileno() (*select.devpoll* 메서드), 1279
fileno() (*select.epoll* 메서드), 1280
fileno() (*select.kqueue* 메서드), 1282
fileno() (*selectors.DevpollSelector* 메서드), 1287
fileno() (*selectors.EpollSelector* 메서드), 1287
fileno() (*selectors.KqueueSelector* 메서드), 1287
fileno() (*socketserver.BaseServer* 메서드), 1549
fileno() (*socket.socket* 메서드), 1231
FileNotFoundError, 119
fileobj (*selectors.SelectorKey*의 속성), 1285
files() (*importlib.abc.TraversableResources* 메서드), 2134
files() (*importlib.metadata* 모듈), 2154
files() (*importlib.resources* 모듈), 2146
files() (*importlib.resources.abc.TraversableResources* 메서드), 2150
files_double_event() (*tkinter.filedialog.FileDialog* 메서드), 1643
files_select_event() (*tkinter.filedialog.FileDialog* 메서드), 1643
filesystem encoding and error handler (파일시스템 인코딩과 에러 처리기), **2306**
FileType (*argparse* 클래스), 907
FileWrapper (*wsgiref.types* 클래스), 1477
FileWrapper (*wsgiref.util* 클래스), 1469
fill() (*textwrap* 모듈), 177
fill() (*textwrap.TextWrapper* 메서드), 180
fill() (*turtle* 모듈), 1695

fillcolor() (*turtle* 모듈), 1694
filling() (*turtle* 모듈), 1695
fillvalue (*reprlib.Repr*의 속성), 325
--filter
    tarfile 명령줄 옵션, 637
Filter (*logging* 클래스), 793
filter (*select.kevent*의 속성), 1282
Filter (*tracemalloc* 클래스), 1961
filter()
    built-in function, 16
filter() (*curses* 모듈), 959
filter() (*fnmatch* 모듈), 509
filter() (*logging.Filter* 메서드), 793
filter() (*logging.Handler* 메서드), 791
filter() (*logging.Logger* 메서드), 789
filter_command() (*tkinter.filedialog.FileDialog* 메서드), 1643
FILTER_DIR (*unittest.mock* 모듈), 1866
filter_traces() (*tracemalloc.Snapshot* 메서드), 1962
FilterError, 625
filterfalse() (*fnmatch* 모듈), 509
filterfalse() (*itertools* 모듈), 428
filterwarnings() (*warnings* 모듈), 2031
Final (*typing* 모듈), 1736
final() (*typing* 모듈), 1761
finalize (*weakref* 클래스), 307
finalize_dict() (*compression.zstd* 모듈), 587
find() (*bytearray* 메서드), 72
find() (*bytes* 메서드), 72
find() (*doctest.DocTestFinder* 메서드), 1793
find() (*gettext* 모듈), 1610
find() (*mmap.mmap* 메서드), 1300
find() (*str* 메서드), 56
find() (*xml.etree.ElementTree.Element* 메서드), 1420
find() (*xml.etree.ElementTree.ElementTree* 메서드), 1421
find_class() (*pickle.Unpickler* 메서드), 528
find_class() (피클 프로토콜), 537
find_library() (*ctypes.util* 모듈), 869
find_longest_match() (*difflib.SequenceMatcher* 메서드), 169
find_msvcrt() (*ctypes.util* 모듈), 869
find_spec() (*importlib.abc.MetaPathFinder* 메서드), 2128
find_spec() (*importlib.abc.PathEntryFinder* 메서드), 2129
find_spec() (*importlib.machinery.FileFinder* 메서드), 2136
find_spec() (*importlib.machinery.PathFinder*의 클래스 메서드), 2136
find_spec() (*importlib.util* 모듈), 2141
find_spec() (*zipimport.zipimporter* 메서드), 2118
find_unused_port() (*test.support.socket_helper* 모듈), 1905
find_user_password() (*urllib.request.HTTPPasswordMgr* 메서드), 1489
find_user_password() (*urllib.request.HTTPPasswordMgrWithPriorAuth* 메서드), 1489
findall() (*re* 모듈), 152
findall() (*re.Pattern* 메서드), 156
findall() (*xml.etree.ElementTree.Element* 메서드), 1420
findall() (*xml.etree.ElementTree.ElementTree* 메서드), 1421
findCaller() (*logging.Logger* 메서드), 789
finder (파인더), **2307**
findfile() (*test.support* 모듈), 1899
finditer() (*re* 모듈), 152
finditer() (*re.Pattern* 메서드), 156
findlabels() (*dis* 모듈), 2227
findlinestarts() (*dis* 모듈), 2226
findtext() (*xml.etree.ElementTree.Element* 메서드), 1420
findtext() (*xml.etree.ElementTree.ElementTree* 메서드), 1422
finish() (*socketserver.BaseRequestHandler* 메서드), 1552
finish() (*tkinter.dnd.DndHandler* 메서드), 1647
finish_request() (*socketserver.BaseServer* 메서드), 1551
FIRST_COMPLETED (*asyncio* 모듈), 1125
FIRST_COMPLETED (*concurrent.futures* 모듈), 1071
FIRST_EXCEPTION (*asyncio* 모듈), 1125
FIRST_EXCEPTION (*concurrent.futures* 모듈), 1071
firstChild (*xml.dom.Node*의 속성), 1429
FirstHeaderLineIsContinuationDefect, 1326
firstkey() (*dbm.gnu.gdbm* 메서드), 548
--first-weekday
    calendar 명령줄 옵션, 268
firstweekday (*calendar.Calendar*의 속성), 261
firstweekday() (*calendar* 모듈), 265
fix_missing_locations() (*ast* 모듈), 2197
fix_sentence_endings (*textwrap.TextWrapper*의 속성), 179
Flag (*enum* 클래스), 336
flag_bits (*zipfile.ZipInfo*의 속성), 621
FlagBoundary (*enum* 클래스), 339
flags (*decimal.Context*의 속성), 378
flags (*re.Pattern*의 속성), 156
flags (*select.kevent*의 속성), 1283
flags (*sys* 모듈), 1987
flash() (*curses* 모듈), 959
flatten() (*email.generator.BytesGenerator* 메서드), 1316
flatten() (*email.generator.Generator* 메서드), 1317
float
    built-in function, 39
--float
    random 명령줄 옵션, 406
float (내장 클래스), 16
float_info (*sys* 모듈), 1989
float_repr_style (*sys* 모듈), 1991
floating-point

literals, 39
object, 38
FloatingPointError, 113
FloatOperation (*decimal* 클래스), 385
flock() (*fcntl* 모듈), 2277
floor division (정수 나눗셈), **2307**
floor() (*in module math*), 39
floor() (*math* 모듈), 353
FloorDiv (*ast* 클래스), 2171
floordiv() (*operator* 모듈), 452
flush() (*bz2.BZ2Compressor* 메서드), 604
flush() (*compression.zstd.ZstdCompressor* 메서드), 586
flush() (*io.BufferedWriter* 메서드), 766
flush() (*io.IOBase* 메서드), 761
flush() (*logging.Handler* 메서드), 791
flush() (*logging.handlers.BufferingHandler* 메서드), 824
flush() (*logging.handlers.MemoryHandler* 메서드), 825
flush() (*logging.StreamHandler* 메서드), 814
flush() (*lzma.LZMACompressor* 메서드), 608
flush() (*mailbox.Mailbox* 메서드), 1373
flush() (*mailbox.Maildir* 메서드), 1376
flush() (*mailbox.MH* 메서드), 1378
flush() (*mmap.mmap* 메서드), 1300
flush() (*xml.etree.ElementTree.XMLParser* 메서드), 1424
flush() (*xml.etree.ElementTree.XMLPullParser* 메서드), 1425
flush() (*zlib.Compress* 메서드), 596
flush() (*zlib.Decompress* 메서드), 597
FLUSH_BLOCK (*compression.zstd.ZstdCompressor*의 속성), 586
FLUSH_FRAME (*compression.zstd.ZstdCompressor*의 속성), 586
flush_headers() (*http.server.BaseHTTPRequestHandler* 메서드), 1559
flush_std_streams() (*test.support* 모듈), 1901
flushinp() (*curses* 모듈), 959
FlushKey() (*winreg* 모듈), 2252
fma() (*decimal.Context* 메서드), 380
fma() (*decimal.Decimal* 메서드), 372
fma() (*math* 모듈), 353
fmean() (*statistics* 모듈), 408
fmod() (*math* 모듈), 353
FMT_BINARY (*plistlib* 모듈), 672
FMT_XML (*plistlib* 모듈), 672
fnmatch
module, 509
fnmatch() (*fnmatch* 모듈), 509
fnmatchcase() (*fnmatch* 모듈), 509
focus() (*tkinter.ttk.Treeview* 메서드), 1659
fold (*datetime.datetime*의 속성), 233
fold (*datetime.time*의 속성), 241
fold() (*email.headerregistry.BaseHeader* 메서드), 1327
fold() (*email.policy.Compat32* 메서드), 1324
fold() (*email.policy.EmailPolicy* 메서드), 1323
fold() (*email.policy.Policy* 메서드), 1322
fold_binary() (*email.policy.Compat32* 메서드), 1324
fold_binary() (*email.policy.EmailPolicy* 메서드), 1323
fold_binary() (*email.policy.Policy* 메서드), 1322
Font (*tkinter.font* 클래스), 1640
For (*ast* 클래스), 2180
FOR_ITER (*opcode*), 2238
forget() (*test.support.import_helper* 모듈), 1910
forget() (*tkinter.ttk.Notebook* 메서드), 1654
fork() (*os* 모듈), 741
fork() (*pty* 모듈), 2273
ForkingMixIn (*socketserver* 클래스), 1548
ForkingTCPServer (*socketserver* 클래스), 1549
ForkingUDPServer (*socketserver* 클래스), 1549
ForkingUnixDatagramServer (*socketserver* 클래스), 1549
ForkingUnixStreamServer (*socketserver* 클래스), 1549
forkpty() (*os* 모듈), 742
Format (*annotationlib* 클래스), 2102
FORMAT (*inspect.BufferFlags*의 속성), 2099
format (*memoryview*의 속성), 90
format (*multiprocessing.shared_memory.ShareableList*의 속성), 1062
format (*struct.Struct*의 속성), 198
format()
built-in function, 17
format() (*inspect.Signature* 메서드), 2089
format() (*logging.BufferingFormatter* 메서드), 793
format() (*logging.Formatter* 메서드), 792
format() (*logging.Handler* 메서드), 791
format() (*pprint.PrettyPrinter* 메서드), 321
format() (*str* 메서드), 57
format() (*string.Formatter* 메서드), 126
format() (*traceback.StackSummary* 메서드), 2070
format() (*traceback.TracebackException* 메서드), 2069
format() (*tracemalloc.Traceback* 메서드), 1964
format_call_graph() (*asyncio* 모듈), 1155
format_datetime() (*email.utils* 모듈), 1359
format_exc() (*traceback* 모듈), 2067
format_exception() (*traceback* 모듈), 2067
format_exception_only() (*traceback* 모듈), 2067
format_exception_only() (*traceback.TracebackException* 메서드), 2069
format_field() (*string.Formatter* 메서드), 127
format_frame_summary() (*traceback.StackSummary* 메서드), 2070
format_help() (*argparse.ArgumentParser* 메서드), 910
format_list() (*traceback* 모듈), 2066
format_map() (*str* 메서드), 57
FORMAT_SIMPLE (*opcode*), 2241
format_spec (*string.templatelib.Interpolation*의 속성), 141

format_stack() (*traceback* 모듈), 2067
format_stack_entry() (*bdb.Bdb* 메서드), 1924
format_string() (*locale* 모듈), 1622
format_tb() (*traceback* 모듈), 2067
format_usage() (*argparse.Action* 메서드), 900
format_usage() (*argparse.ArgumentParser* 메서드), 910
FORMAT_WITH_SPEC (*opcode*), 2241
formataddr() (*email.utils* 모듈), 1358
formatargvalues() (*inspect* 모듈), 2093
formatdate() (*email.utils* 모듈), 1358
formatday() (*calendar.TextCalendar* 메서드), 262
FormatError, 1388
FormatError() (*ctypes* 모듈), 869
formatException() (*logging.Formatter* 메서드), 793
formatFooter() (*logging.BufferingFormatter* 메서드), 793
formatHeader() (*logging.BufferingFormatter* 메서드), 793
formatmonth() (*calendar.HTMLCalendar* 메서드), 263
formatmonth() (*calendar.TextCalendar* 메서드), 263
formatmonthname() (*calendar.HTMLCalendar* 메서드), 263
formatmonthname() (*calendar.TextCalendar* 메서드), 263
formatStack() (*logging.Formatter* 메서드), 793
formatted string literals, 64
FormattedValue (*ast* 클래스), 2167
Formatter (*logging* 클래스), 792
Formatter (*string* 클래스), 126
formatTime() (*logging.Formatter* 메서드), 792
formatting
    bytearray (%), 81
    bytes (%), 81
formatting, string (%), 66
formatwarning() (*warnings* 모듈), 2031
formatweek() (*calendar.TextCalendar* 메서드), 262
formatweekday() (*calendar.TextCalendar* 메서드), 262
formatweekheader() (*calendar.TextCalendar* 메서드), 263
formatyear() (*calendar.HTMLCalendar* 메서드), 263
formatyear() (*calendar.TextCalendar* 메서드), 263
formatyearpage() (*calendar.HTMLCalendar* 메서드), 263
Fortran contiguous, 2304
forward() (*turtle* 모듈), 1685
ForwardRef (*annotationlib* 클래스), 2102
FORWARDREF (*annotationlib.Format*의 속성), 2102
ForwardRef (*typing* 클래스), 1765
fp (*urllib.error.HTTPError*의 속성), 1505
fpathconf() (*os* 모듈), 701
Fraction (*fractions* 클래스), 393
fractions
    module, 393
frame (*inspect.FrameInfo*의 속성), 2094
frame (*tkinter.scrolledtext.ScrolledText*의 속성), 1647
Frame (*tracemalloc* 클래스), 1961
FrameInfo (*compression.zstd* 클래스), 593
FrameInfo (*inspect* 클래스), 2094
FrameSummary (*traceback* 클래스), 2070
FrameType (*types* 모듈), 315
free threading (자유 스레딩), **2307**
free variable (자유 변수), **2307**
free_tool_id() (*sys.monitoring* 모듈), 2009
freedesktop_os_release() (*platform* 모듈), 832
freeze() (*gc* 모듈), 2080
freeze_support() (*multiprocessing* 모듈), 1026
frexp() (*math* 모듈), 354
FRIDAY (*calendar* 모듈), 266
from_address() (*ctypes._CData* 메서드), 871
from_buffer() (*ctypes._CData* 메서드), 871
from_buffer_copy() (*ctypes._CData* 메서드), 871
from_bytes() (*int*의 클래스 메서드), 42
from_callable() (*inspect.Signature*의 클래스 메서드), 2089
from_decimal() (*fractions.Fraction*의 클래스 메서드), 395
from_exception() (*traceback.TracebackException*의 클래스 메서드), 2069
from_file() (*zipfile.ZipInfo*의 클래스 메서드), 620
from_file() (*zoneinfo.ZoneInfo*의 클래스 메서드), 258
from_float() (*decimal.Decimal*의 클래스 메서드), 371
from_float() (*fractions.Fraction*의 클래스 메서드), 395
from_iterable() (*itertools.chain*의 클래스 메서드), 426
from_list() (*traceback.StackSummary*의 클래스 메서드), 2070
from_number() (*complex*의 클래스 메서드), 44
from_number() (*decimal.Decimal*의 클래스 메서드), 371
from_number() (*float*의 클래스 메서드), 43
from_number() (*fractions.Fraction*의 클래스 메서드), 395
from_param() (*ctypes._CData* 메서드), 871
from_samples() (*statistics.NormalDist*의 클래스 메서드), 418
from_traceback() (*dis.Bytecode*의 클래스 메서드), 2224
from_uri() (*pathlib.Path*의 클래스 메서드), 472
frombuf() (*tarfile.TarInfo*의 클래스 메서드), 631
frombytes() (*array.array* 메서드), 302
fromfd() (*select.epoll* 메서드), 1281
fromfd() (*select.kqueue* 메서드), 1282
fromfd() (*socket* 모듈), 1224
fromfile() (*array.array* 메서드), 302
fromhex() (*bytearray*의 클래스 메서드), 69
fromhex() (*bytes*의 클래스 메서드), 68
fromhex() (*float*의 클래스 메서드), 43
fromisocalendar() (*datetime.datetime*의 클래스 메서드), 232

fromisocalendar() (*datetime.date*의 클래스 메서드), 224
fromisoformat() (*datetime.datetime*의 클래스 메서드), 231
fromisoformat() (*datetime.date*의 클래스 메서드), 224
fromisoformat() (*datetime.time*의 클래스 메서드), 241
fromkeys() (*collections.Counter* 메서드), 273
fromkeys() (*dict*의 클래스 메서드), 95
fromlist() (*array.array* 메서드), 303
fromordinal() (*datetime.datetime*의 클래스 메서드), 231
fromordinal() (*datetime.date*의 클래스 메서드), 224
fromshare() (*socket* 모듈), 1224
fromstring() (*xml.etree.ElementTree* 모듈), 1415
fromstringlist() (*xml.etree.ElementTree* 모듈), 1415
fromtarfile() (*tarfile.TarInfo*의 클래스 메서드), 631
fromtimestamp() (*datetime.datetime*의 클래스 메서드), 230
fromtimestamp() (*datetime.date*의 클래스 메서드), 224
fromunicode() (*array.array* 메서드), 303
fromutc() (*datetime.timezone* 메서드), 251
fromutc() (*datetime.tzinfo* 메서드), 246
FrozenImporter (*importlib.machinery* 클래스), 2135
FrozenInstanceError, 2040
FrozenSet (*typing* 클래스), 1767
frozenset (내장 클래스), 91
FS (*curses.ascii* 모듈), 985
fs_is_case_insensitive() (*test.support.os_helper* 모듈), 1909
FS_NONASCII (*test.support.os_helper* 모듈), 1908
fsdecode() (*os* 모듈), 692
fsencode() (*os* 모듈), 692
fspath() (*os* 모듈), 692
fstat() (*os* 모듈), 701
fstatvfs() (*os* 모듈), 701
fstring, 64
FSTRING_END (*token* 모듈), 2206
FSTRING_MIDDLE (*token* 모듈), 2206
FSTRING_START (*token* 모듈), 2205
f-strings, 64
fsum() (*math* 모듈), 356
fsync() (*os* 모듈), 701
FTP, 1495
    ftplib (표준 모듈), 1517
    프로토콜, 1495, 1517
FTP (*ftplib* 클래스), 1518
ftp_open() (*urllib.request.FTPHandler* 메서드), 1490
FTP_TLS (*ftplib* 클래스), 1522
FTPHandler (*urllib.request* 클래스), 1484
ftplib
    module, 1517
ftruncate() (*os* 모듈), 702
Full, 1098
FULL (*inspect.BufferFlags*의 속성), 2100
full() (*asyncio.Queue* 메서드), 1151
full() (*multiprocessing.Queue* 메서드), 1024
full() (*queue.Queue* 메서드), 1099
full_match() (*pathlib.PurePath* 메서드), 468
FULL_RO (*inspect.BufferFlags*의 속성), 2100
full_url (*urllib.request.Request*의 속성), 1484
fullmatch() (*re* 모듈), 151
fullmatch() (*re.Pattern* 메서드), 155
fully_trusted_filter() (*tarfile* 모듈), 634
func (*functools.partial*의 속성), 451
funcname (*bdb.Breakpoint*의 속성), 1920
function (*inspect.FrameInfo*의 속성), 2094
function (*inspect.Traceback*의 속성), 2095
Function (*pyclbr* 클래스), 2215
Function (*symtable* 클래스), 2202
FUNCTION (*symtable.SymbolTableType*의 속성), 2201
function (함수), **2307**
function annotation (함수 어노테이션), **2307**
FunctionDef (*ast* 클래스), 2191
FunctionTestCase (*unittest* 클래스), 1821
FunctionType (*ast* 클래스), 2166
FunctionType (*types* 모듈), 313
functools
    module, 440
funny_files (*filecmp.dircmp*의 속성), 500
Future (*asyncio* 클래스), 1181
Future (*concurrent.futures* 클래스), 1069
future_add_to_awaited_by() (*asyncio* 모듈), 1155
future_discard_from_awaited_by() (*asyncio* 모듈), 1156
FutureWarning, 120
fwalk() (*os* 모듈), 732

G

-g
    trace **명령줄 옵션**, 1953
gaierror, 1216
gamma() (*math* 모듈), 358
gammavariate() (*random* 모듈), 400
garbage (*gc* 모듈), 2080
garbage collection (가비지 수거), **2307**
gather() (*asyncio* 모듈), 1120
gather() (*curses.textpad.Textbox* 메서드), 983
gauss() (*random* 모듈), 400
gc
    module, 2077
gc_collect() (*test.support* 모듈), 1900
gcd() (*math* 모듈), 352
ge() (*operator* 모듈), 451
generate_tokens() (*tokenize* 모듈), 2211
Generator (*collections.abc* 클래스), 290
Generator (*email.generator* 클래스), 1317
Generator (*typing* 클래스), 1771
generator (제너레이터), 2307, **2308**
generator expression (제너레이터 표현식), **2308**

generator iterator (제너레이터 이터레이터), **2308**
generator_stop (__future__ 모듈), 2076
GeneratorExit, 113
GeneratorExp (*ast* 클래스), 2174
generators (__future__ 모듈), 2076
GeneratorType (*types* 모듈), 313
Generic
    Alias, 99
Generic (*typing* 클래스), 1741
generic function (제네릭 함수), **2308**
generic type (제네릭 형), **2308**
generic_visit() (*ast.NodeVisitor* 메서드), 2197
GenericAlias
    object, 99
GenericAlias (*types* 클래스), 314
genops() (*pickletools* 모듈), 2246
geometric_mean() (*statistics* 모듈), 409
get() (*asyncio.Queue* 메서드), 1151
get() (*configparser.ConfigParser* 메서드), 664
get() (*contextvars.Context* 메서드), 1104
get() (*contextvars.ContextVar* 메서드), 1102
get() (*dict* 메서드), 95
get() (*email.message.EmailMessage* 메서드), 1307
get() (*email.message.Message* 메서드), 1345
get() (*mailbox.Mailbox* 메서드), 1372
get() (*multiprocessing.pool.AsyncResult* 메서드), 1043
get() (*multiprocessing.Queue* 메서드), 1025
get() (*multiprocessing.SimpleQueue* 메서드), 1026
get() (*queue.Queue* 메서드), 1099
get() (*queue.SimpleQueue* 메서드), 1101
get() (*tkinter.ttk.Combobox* 메서드), 1652
get() (*tkinter.ttk.Spinbox* 메서드), 1653
get() (*types.MappingProxyType* 메서드), 316
get() (*webbrowser* 모듈), 1466
get() (*xml.etree.ElementTree.Element* 메서드), 1419
GET_AITER (*opcode*), 2231
get_all() (*email.message.EmailMessage* 메서드), 1307
get_all() (*email.message.Message* 메서드), 1345
get_all() (*wsgiref.headers.Headers* 메서드), 1470
get_all_breaks() (*bdb.Bdb* 메서드), 1923
get_all_start_methods() (*multiprocessing* 모듈), 1027
GET_ANEXT (*opcode*), 2231
get_annotate_from_class_namespace() (*annotationlib* 모듈), 2104
get_annotations() (*annotationlib* 모듈), 2104
get_annotations() (*inspect* 모듈), 2094
get_app() (*wsgiref.simple_server.WSGIServer* 메서드), 1471
get_archive_formats() (*shutil* 모듈), 518
get_args() (*typing* 모듈), 1764
get_asyncgen_hooks() (*sys* 모듈), 1994
get_attribute() (*test.support* 모듈), 1903
GET_AWAITABLE (*opcode*), 2231
get_begidx() (*readline* 모듈), 187
get_blocking() (*os* 모듈), 702
get_body() (*email.message.EmailMessage* 메서드), 1310
get_body_encoding() (*email.charset.Charset* 메서드), 1355
get_boundary() (*email.message.EmailMessage* 메서드), 1308
get_boundary() (*email.message.Message* 메서드), 1347
get_bpbynumber() (*bdb.Bdb* 메서드), 1923
get_break() (*bdb.Bdb* 메서드), 1923
get_breaks() (*bdb.Bdb* 메서드), 1923
get_buffer() (*asyncio.BufferedProtocol* 메서드), 1191
get_bytes() (*mailbox.Mailbox* 메서드), 1372
get_bytes() (*mailbox.mbox* 메서드), 1377
get_bytes() (*mailbox.MMDF* 메서드), 1380
get_ca_certs() (*ssl.SSLContext* 메서드), 1260
get_cache_token() (*abc* 모듈), 2063
get_channel_binding() (*ssl.SSLSocket* 메서드), 1257
get_charset() (*email.message.Message* 메서드), 1344
get_charsets() (*email.message.EmailMessage* 메서드), 1309
get_charsets() (*email.message.Message* 메서드), 1347
get_children() (*symtable.SymbolTable* 메서드), 2202
get_children() (*tkinter.ttk.Treeview* 메서드), 1658
get_ciphers() (*ssl.SSLContext* 메서드), 1260
get_clock_info() (*time* 모듈), 773
get_close_matches() (*difflib* 모듈), 166
get_code() (*importlib.abc.InspectLoader* 메서드), 2130
get_code() (*importlib.abc.SourceLoader* 메서드), 2132
get_code() (*importlib.machinery.ExtensionFileLoader* 메서드), 2138
get_code() (*importlib.machinery.SourcelessFileLoader* 메서드), 2137
get_code() (*zipimport.zipimporter* 메서드), 2118
get_completer() (*readline* 모듈), 187
get_completer_delims() (*readline* 모듈), 187
get_completion_type() (*readline* 모듈), 187
get_config_h_filename() (*sysconfig* 모듈), 2018
get_config_var() (*sysconfig* 모듈), 2013
get_config_vars() (*sysconfig* 모듈), 2013
get_content() (*email.contentmanager* 모듈), 1333
get_content() (*email.contentmanager.ContentManager* 메서드), 1332
get_content() (*email.message.EmailMessage* 메서드), 1310
get_content_charset() (*email.message.EmailMessage* 메서드), 1308
get_content_charset() (*email.message.Message* 메서드), 1347
get_content_disposition()

(*email.message.EmailMessage* 메서드), 1309
get_content_disposition() (*email.message.Message* 메서드), 1347
get_content_maintype() (*email.message.EmailMessage* 메서드), 1307
get_content_maintype() (*email.message.Message* 메서드), 1345
get_content_subtype() (*email.message.EmailMessage* 메서드), 1307
get_content_subtype() (*email.message.Message* 메서드), 1346
get_content_type() (*email.message.EmailMessage* 메서드), 1307
get_content_type() (*email.message.Message* 메서드), 1345
get_context() (*asyncio.Handle* 메서드), 1175
get_context() (*asyncio.Task* 메서드), 1130
get_context() (*multiprocessing* 모듈), 1027
get_coro() (*asyncio.Task* 메서드), 1130
get_coroutine_origin_tracking_depth() (*sys* 모듈), 1994
get_count() (*gc* 모듈), 2078
get_current() (*concurrent.interpreters* 모듈), 1074
get_current_history_length() (*readline* 모듈), 186
get_data() (*importlib.abc.FileLoader* 메서드), 2132
get_data() (*importlib.abc.ResourceLoader* 메서드), 2130
get_data() (*pkgutil* 모듈), 2121
get_data() (*zipimport.zipimporter* 메서드), 2118
get_date() (*mailbox.MaildirMessage* 메서드), 1382
get_debug() (*asyncio.loop* 메서드), 1173
get_debug() (*gc* 모듈), 2078
get_default() (*argparse.ArgumentParser* 메서드), 909
get_default_backend() (*pdb* 모듈), 1931
get_default_scheme() (*sysconfig* 모듈), 2017
get_default_type() (*email.message.EmailMessage* 메서드), 1308
get_default_type() (*email.message.Message* 메서드), 1346
get_default_verify_paths() (*ssl* 모듈), 1246
get_dialect() (*csv* 모듈), 642
get_disassembly_as_string() (*test.support.bytecode_helper.BytecodeTestCase* 메서드), 1907
get_docstring() (*ast* 모듈), 2197
get_doctest() (*doctest.DocTestParser* 메서드), 1794
get_endidx() (*readline* 모듈), 187
get_environ() (*wsgiref.simple_server.WSGIRequestHandler* 메서드), 1472
get_errno() (*ctypes* 모듈), 869
get_escdelay() (*curses* 모듈), 962
get_event_loop() (*asyncio* 모듈), 1156
get_event_loop() (*asyncio.AbstractEventLoopPolicy* 메서드), 1198
get_event_loop_policy() (*asyncio* 모듈), 1198
get_events() (*sys.monitoring* 모듈), 2011
get_examples() (*doctest.DocTestParser* 메서드), 1794
get_exception_handler() (*asyncio.loop* 메서드), 1173
get_exec_path() (*os* 모듈), 693
get_extra_info() (*asyncio.BaseTransport* 메서드), 1186
get_extra_info() (*asyncio.StreamWriter* 메서드), 1136
get_field() (*string.Formatter* 메서드), 126
get_file() (*mailbox.Babyl* 메서드), 1379
get_file() (*mailbox.Mailbox* 메서드), 1373
get_file() (*mailbox.Maildir* 메서드), 1376
get_file() (*mailbox.mbox* 메서드), 1377
get_file() (*mailbox.MH* 메서드), 1378
get_file() (*mailbox.MMDF* 메서드), 1380
get_file_breaks() (*bdb.Bdb* 메서드), 1923
get_filename() (*email.message.EmailMessage* 메서드), 1308
get_filename() (*email.message.Message* 메서드), 1347
get_filename() (*importlib.abc.ExecutionLoader* 메서드), 2131
get_filename() (*importlib.abc.FileLoader* 메서드), 2131
get_filename() (*importlib.machinery.ExtensionFileLoader* 메서드), 2138
get_filename() (*zipimport.zipimporter* 메서드), 2118
get_filter() (*tkinter.filedialog.FileDialog* 메서드), 1643
get_flags() (*mailbox.Maildir* 메서드), 1375
get_flags() (*mailbox.MaildirMessage* 메서드), 1381
get_flags() (*mailbox.mboxMessage* 메서드), 1383
get_flags() (*mailbox.MMDFMessage* 메서드), 1387
get_folder() (*mailbox.Maildir* 메서드), 1374
get_folder() (*mailbox.MH* 메서드), 1378
get_frame_info() (*compression.zstd* 모듈), 593
get_frees() (*symtable.Function* 메서드), 2202
get_freeze_count() (*gc* 모듈), 2080
get_from() (*mailbox.mboxMessage* 메서드), 1383
get_from() (*mailbox.MMDFMessage* 메서드), 1387
get_full_url() (*urllib.request.Request* 메서드), 1485
get_globals() (*symtable.Function* 메서드), 2202
get_grouped_opcodes() (*difflib.SequenceMatcher* 메서드), 170
get_handle_inheritable() (*os* 모듈), 712
get_header() (*urllib.request.Request* 메서드), 1485
get_history_item() (*readline* 모듈), 186
get_history_length() (*readline* 모듈), 186
get_id() (*symtable.SymbolTable* 메서드), 2201
get_ident() (*_thread* 모듈), 1106

get_ident() (*threading* 모듈), 997
get_identifiers() (*string.Template* 메서드), 135
get_identifiers() (*symtable.SymbolTable* 메서드), 2202
get_importer() (*pkgutil* 모듈), 2120
get_info() (*mailbox.Maildir* 메서드), 1375
get_info() (*mailbox.MaildirMessage* 메서드), 1382
get_inheritable() (*os* 모듈), 712
get_inheritable() (*socket.socket* 메서드), 1231
get_instructions() (*dis* 모듈), 2226
get_int_max_str_digits() (*sys* 모듈), 1992
get_interpreter() (*zipapp* 모듈), 1978
GET_ITER (*opcode*), 2229
get_key() (*selectors.BaseSelector* 메서드), 1286
get_labels() (*mailbox.Babyl* 메서드), 1379
get_labels() (*mailbox.BabylMessage* 메서드), 1385
get_last_error() (*ctypes* 모듈), 869
GET_LEN (*opcode*), 2233
get_line_buffer() (*readline* 모듈), 185
get_lineno() (*symtable.SymbolTable* 메서드), 2202
get_local_events() (*sys.monitoring* 모듈), 2012
get_locals() (*symtable.Function* 메서드), 2202
get_logger() (*multiprocessing* 모듈), 1047
get_loop() (*asyncio.Future* 메서드), 1183
get_loop() (*asyncio.Runner* 메서드), 1112
get_loop() (*asyncio.Server* 메서드), 1176
get_main() (*concurrent.interpreters* 모듈), 1075
get_makefile_filename() (*sysconfig* 모듈), 2018
get_map() (*selectors.BaseSelector* 메서드), 1286
get_matching_blocks() (*difflib.SequenceMatcher* 메서드), 170
get_message() (*mailbox.Mailbox* 메서드), 1372
get_method() (*urllib.request.Request* 메서드), 1484
get_methods() (*symtable.Class* 메서드), 2202
get_mixed_type_key() (*ipaddress* 모듈), 1603
get_name() (*asyncio.Task* 메서드), 1130
get_name() (*symtable.Symbol* 메서드), 2203
get_name() (*symtable.SymbolTable* 메서드), 2201
get_namespace() (*symtable.Symbol* 메서드), 2204
get_namespaces() (*symtable.Symbol* 메서드), 2204
get_native_id() (*_thread* 모듈), 1106
get_native_id() (*threading* 모듈), 997
get_nonlocals() (*symtable.Function* 메서드), 2202
get_nonstandard_attr() (*http.cookiejar.Cookie* 메서드), 1575
get_nowait() (*asyncio.Queue* 메서드), 1151
get_nowait() (*multiprocessing.Queue* 메서드), 1025
get_nowait() (*queue.Queue* 메서드), 1099
get_nowait() (*queue.SimpleQueue* 메서드), 1101
get_object_traceback() (*tracemalloc* 모듈), 1959
get_objects() (*gc* 모듈), 2078
get_opcodes() (*difflib.SequenceMatcher* 메서드), 170
get_option() (*optparse.OptionParser* 메서드), 945
get_option_group() (*optparse.OptionParser* 메서드), 936
get_origin() (*typing* 모듈), 1764
get_original_bases() (*types* 모듈), 312
get_original_stdout() (*test.support* 모듈), 1899
get_osfhandle() (*msvcrt* 모듈), 2248
get_output_charset() (*email.charset.Charset* 메서드), 1355
get_overloads() (*typing* 모듈), 1761
get_pagesize() (*test.support* 모듈), 1899
get_param() (*email.message.Message* 메서드), 1346
get_parameters() (*symtable.Function* 메서드), 2202
get_params() (*email.message.Message* 메서드), 1346
get_path() (*sysconfig* 모듈), 2017
get_path_names() (*sysconfig* 모듈), 2017
get_paths() (*sysconfig* 모듈), 2017
get_payload() (*email.message.Message* 메서드), 1343
get_pid() (*asyncio.SubprocessTransport* 메서드), 1188
get_pipe_transport() (*asyncio.SubprocessTransport* 메서드), 1188
get_platform() (*sysconfig* 모듈), 2018
get_poly() (*turtle* 모듈), 1700
get_preferred_scheme() (*sysconfig* 모듈), 2017
get_protocol() (*asyncio.BaseTransport* 메서드), 1186
get_protocol_members() (*typing* 모듈), 1764
get_proxy_response_headers() (*http.client.HTTPConnection* 메서드), 1514
get_python_version() (*sysconfig* 모듈), 2018
get_ready() (*graphlib.TopologicalSorter* 메서드), 345
get_referents() (*gc* 모듈), 2079
get_referrers() (*gc* 모듈), 2079
get_request() (*socketserver.BaseServer* 메서드), 1551
get_returncode() (*asyncio.SubprocessTransport* 메서드), 1188
get_running_loop() (*asyncio* 모듈), 1156
get_scheme() (*wsgiref.handlers.BaseHandler* 메서드), 1475
get_scheme_names() (*sysconfig* 모듈), 2017
get_selection() (*tkinter.filedialog.FileDialog* 메서드), 1643
get_sequences() (*mailbox.MH* 메서드), 1378
get_sequences() (*mailbox.MHMessage* 메서드), 1384
get_server() (*multiprocessing.managers.BaseManager* 메서드), 1035
get_server_certificate() (*ssl* 모듈), 1246
get_shapepoly() (*turtle* 모듈), 1699
get_source() (*importlib.abc.InspectLoader* 메서드), 2130
get_source() (*importlib.abc.SourceLoader* 메서드), 2133
get_source() (*importlib.machinery.ExtensionFileLoader* 메서드), 2138
get_source() (*importlib.machinery.SourcelessFileLoader* 메서드), 2137
get_source() (*zipimport.zipimporter* 메서드), 2118
get_source_segment() (*ast* 모듈), 2197
get_stack() (*asyncio.Task* 메서드), 1130

get_stack() (*bdb.Bdb* 메서드), 1924
get_start_method() (*multiprocessing* 모듈), 1027
get_starttag_text() (*html.parser.HTMLParser* 메서드), 1403
get_stats() (*gc* 모듈), 2078
get_stats_profile() (*pstats.Stats* 메서드), 1945
get_stderr() (*wsgiref.handlers.BaseHandler* 메서드), 1474
get_stderr() (*wsgiref.simple_server.WSGIRequestHandler* 메서드), 1472
get_stdin() (*wsgiref.handlers.BaseHandler* 메서드), 1474
get_string() (*mailbox.Mailbox* 메서드), 1372
get_string() (*mailbox.mbox* 메서드), 1377
get_subdir() (*mailbox.MaildirMessage* 메서드), 1381
get_symbols() (*symtable.SymbolTable* 메서드), 2202
get_tabsize() (*curses* 모듈), 963
get_task_factory() (*asyncio.loop* 메서드), 1161
get_terminal_size() (*os* 모듈), 711
get_terminal_size() (*shutil* 모듈), 521
get_threshold() (*gc* 모듈), 2079
get_token() (*shlex.shlex* 메서드), 2265
get_tool() (*sys.monitoring* 모듈), 2009
get_traceback_limit() (*tracemalloc* 모듈), 1959
get_traced_memory() (*tracemalloc* 모듈), 1959
get_tracemalloc_memory() (*tracemalloc* 모듈), 1959
get_type() (*symtable.SymbolTable* 메서드), 2201
get_type_hints() (*typing* 모듈), 1763
get_unixfrom() (*email.message.EmailMessage* 메서드), 1306
get_unixfrom() (*email.message.Message* 메서드), 1343
get_unpack_formats() (*shutil* 모듈), 520
get_unverified_chain() (*ssl.SSLSocket* 메서드), 1256
get_usage() (*optparse.OptionParser* 메서드), 946
get_value() (*string.Formatter* 메서드), 126
get_verified_chain() (*ssl.SSLSocket* 메서드), 1256
get_version() (*optparse.OptionParser* 메서드), 936
get_visible() (*mailbox.BabylMessage* 메서드), 1386
get_wch() (*curses.window* 메서드), 967
get_write_buffer_limits() (*asyncio.WriteTransport* 메서드), 1187
get_write_buffer_size() (*asyncio.WriteTransport* 메서드), 1187
GET_YIELD_FROM_ITER (*opcode*), 2229
getacl() (*imaplib.IMAP4* 메서드), 1529
getaddresses() (*email.utils* 모듈), 1358
getaddrinfo() (*asyncio.loop* 메서드), 1170
getaddrinfo() (*socket* 모듈), 1225
getallocatedblocks() (*sys* 모듈), 1991
getandroidapilevel() (*sys* 모듈), 1991
getannotation() (*imaplib.IMAP4* 메서드), 1530
getargvalues() (*inspect* 모듈), 2092
getasyncgenlocals() (*inspect* 모듈), 2098
getasyncgenstate() (*inspect* 모듈), 2097
getatime() (*os.path* 모듈), 488
getattr()
built-in function, 17
getattr_static() (*inspect* 모듈), 2096
getAttribute() (*xml.dom.Element* 메서드), 1432
getAttributeNode() (*xml.dom.Element* 메서드), 1432
getAttributeNodeNS() (*xml.dom.Element* 메서드), 1432
getAttributeNS() (*xml.dom.Element* 메서드), 1432
GetBase() (*xml.parsers.expat.xmlparser* 메서드), 1456
getbegyx() (*curses.window* 메서드), 967
getbkgd() (*curses.window* 메서드), 967
getblocking() (*socket.socket* 메서드), 1232
getboolean() (*configparser.ConfigParser* 메서드), 664
getbuffer() (*io.BytesIO* 메서드), 765
getByteStream() (*xml.sax.xmlreader.InputSource* 메서드), 1453
getcallargs() (*inspect* 모듈), 2093
getcanvas() (*turtle* 모듈), 1707
getch() (*curses.window* 메서드), 967
getch() (*msvcrt* 모듈), 2248
getCharacterStream() (*xml.sax.xmlreader.InputSource* 메서드), 1454
getche() (*msvcrt* 모듈), 2248
getChild() (*logging.Logger* 메서드), 787
getChildren() (*logging.Logger* 메서드), 787
getclasstree() (*inspect* 모듈), 2092
getclosurevars() (*inspect* 모듈), 2093
getcode() (*http.client.HTTPResponse* 메서드), 1516
getcode() (*urllib.response.addinfourl* 메서드), 1495
getColumnNumber() (*xml.sax.xmlreader.Locator* 메서드), 1453
getcomments() (*inspect* 모듈), 2086
getcompname() (*wave.Wave_read* 메서드), 1606
getcomptype() (*wave.Wave_read* 메서드), 1606
getconfig() (*sqlite3.Connection* 메서드), 565
getContentHandler() (*xml.sax.xmlreader.XMLReader* 메서드), 1451
getcontext() (*decimal* 모듈), 376
getcoroutinelocals() (*inspect* 모듈), 2098
getcoroutinestate() (*inspect* 모듈), 2097
getctime() (*os.path* 모듈), 488
getcwd() (*os* 모듈), 715
getcwdb() (*os* 모듈), 715
getdecoder() (*codecs* 모듈), 200
getdefaultencoding() (*sys* 모듈), 1991
getdefaultlocale() (*locale* 모듈), 1621
getdefaulttimeout() (*socket* 모듈), 1229
getdlopenflags() (*sys* 모듈), 1991
getdoc() (*inspect* 모듈), 2086
getDOMImplementation() (*xml.dom* 모듈), 1427

getDTDHandler() (*xml.sax.xmlreader.XMLReader* 메서드), 1452
getEffectiveLevel() (*logging.Logger* 메서드), 787
getegid() (*os* 모듈), 693
getElementsByTagName() (*xml.dom.Document* 메서드), 1432
getElementsByTagName() (*xml.dom.Element* 메서드), 1432
getElementsByTagNameNS() (*xml.dom.Document* 메서드), 1432
getElementsByTagNameNS() (*xml.dom.Element* 메서드), 1432
getencoder() (*codecs* 모듈), 199
getencoding() (*locale* 모듈), 1621
getEncoding() (*xml.sax.xmlreader.InputSource* 메서드), 1453
getEntityResolver() (*xml.sax.xmlreader.XMLReader* 메서드), 1452
getenv() (*os* 모듈), 693
getenvb() (*os* 모듈), 693
getErrorHandler() (*xml.sax.xmlreader.XMLReader* 메서드), 1452
geteuid() (*os* 모듈), 693
getEvent() (*xml.dom.pulldom.DOMEventStream* 메서드), 1442
getEventCategory() (*logging.handlers.NTEventLogHandler* 메서드), 823
getEventType() (*logging.handlers.NTEventLogHandler* 메서드), 823
getException() (*xml.sax.SAXException* 메서드), 1444
getFeature() (*xml.sax.xmlreader.XMLReader* 메서드), 1452
getfile() (*inspect* 모듈), 2086
getFilesToDelete() (*logging.handlers.TimedRotatingFileHandler* 메서드), 819
getfilesystemencodeerrors() (*sys* 모듈), 1992
getfilesystemencoding() (*sys* 모듈), 1991
getfirstweekday() (*calendar.Calendar* 메서드), 261
getfloat() (*configparser.ConfigParser* 메서드), 664
getfqdn() (*socket* 모듈), 1226
getframeinfo() (*inspect* 모듈), 2095
getframerate() (*wave.Wave_read* 메서드), 1606
getfullargspec() (*inspect* 모듈), 2092
getgeneratorlocals() (*inspect* 모듈), 2098
getgeneratorstate() (*inspect* 모듈), 2097
getgid() (*os* 모듈), 693
getgrall() (*grp* 모듈), 2271
getgrgid() (*grp* 모듈), 2270
getgrnam() (*grp* 모듈), 2270
getgrouplist() (*os* 모듈), 693
getgroups() (*os* 모듈), 693
getHandlerByName() (*logging* 모듈), 799
getHandlerNames() (*logging* 모듈), 799
getheader() (*http.client.HTTPResponse* 메서드), 1515
getheaders() (*http.client.HTTPResponse* 메서드), 1515
gethostbyaddr() (*in module socket*), 698
gethostbyaddr() (*socket* 모듈), 1226
gethostbyname() (*socket* 모듈), 1226
gethostbyname_ex() (*socket* 모듈), 1226
gethostname() (*in module socket*), 698
gethostname() (*socket* 모듈), 1226
getincrementaldecoder() (*codecs* 모듈), 200
getincrementalencoder() (*codecs* 모듈), 200
getinfo() (*zipfile.ZipFile* 메서드), 614
getinnerframes() (*inspect* 모듈), 2096
GetInputContext() (*xml.parsers.expat.xmlparser* 메서드), 1456
getint() (*configparser.ConfigParser* 메서드), 664
getitem() (*operator* 모듈), 454
getitimer() (*signal* 모듈), 1293
getkey() (*curses.window* 메서드), 967
GetLastError() (*ctypes* 모듈), 869
getLength() (*xml.sax.xmlreader.Attributes* 메서드), 1454
getLevelName() (*logging* 모듈), 799
getLevelNamesMapping() (*logging* 모듈), 798
getlimit() (*sqlite3.Connection* 메서드), 565
getline() (*linecache* 모듈), 510
getLineNumber() (*xml.sax.xmlreader.Locator* 메서드), 1453
getloadavg() (*os* 모듈), 754
getlocale() (*locale* 모듈), 1621
getLogger() (*logging* 모듈), 797
getLoggerClass() (*logging* 모듈), 797
getlogin() (*os* 모듈), 694
getLogRecordFactory() (*logging* 모듈), 797
getMandatoryRelease() (*__future__._Feature* 메서드), 2077
getmark() (*wave.Wave_read* 메서드), 1606
getmarkers() (*wave.Wave_read* 메서드), 1606
getmaxyx() (*curses.window* 메서드), 967
getmember() (*tarfile.TarFile* 메서드), 628
getmembers() (*inspect* 모듈), 2083
getmembers() (*tarfile.TarFile* 메서드), 628
getmembers_static() (*inspect* 모듈), 2083
getMessage() (*logging.LogRecord* 메서드), 794
getMessage() (*xml.sax.SAXException* 메서드), 1444
getMessageID() (*logging.handlers.NTEventLogHandler* 메서드), 824
getmodule() (*inspect* 모듈), 2086
getmodulename() (*inspect* 모듈), 2084
getmouse() (*curses* 모듈), 959
getmro() (*inspect* 모듈), 2093
getmtime() (*os.path* 모듈), 488
getName() (*threading.Thread* 메서드), 1003
getNameByQName() (*xml.sax.xmlreader.AttributesNS* 메서드), 1454

getnameinfo() (*asyncio.loop* 메서드), 1170
getnameinfo() (*socket* 모듈), 1226
getnames() (*tarfile.TarFile* 메서드), 628
getNames() (*xml.sax.xmlreader.Attributes* 메서드), 1454
getnchannels() (*wave.Wave_read* 메서드), 1606
getnframes() (*wave.Wave_read* 메서드), 1606
getnode() (*uuid* 모듈), 1543
getobjects() (*sys* 모듈), 1993
getopt
    module, 2289
getopt() (*getopt* 모듈), 2289
GetoptError, 2290
getOptionalRelease() (*__future__._Feature* 메서드), 2076
getouterframes() (*inspect* 모듈), 2095
getoutput() (*subprocess* 모듈), 1095
getpagesize() (*resource* 모듈), 2282
getparams() (*wave.Wave_read* 메서드), 1606
getparyx() (*curses.window* 메서드), 967
getpass
    module, 953
getpass() (*getpass* 모듈), 954
GetPassWarning, 954
getpeercert() (*ssl.SSLSocket* 메서드), 1255
getpeername() (*socket.socket* 메서드), 1232
getpen() (*turtle* 모듈), 1701
getpgid() (*os* 모듈), 694
getpgrp() (*os* 모듈), 694
getpid() (*os* 모듈), 694
getpos() (*html.parser.HTMLParser* 메서드), 1403
getppid() (*os* 모듈), 694
getpreferredencoding() (*locale* 모듈), 1621
getpriority() (*os* 모듈), 694
getprofile() (*sys* 모듈), 1993
getprofile() (*threading* 모듈), 998
getProperty() (*xml.sax.xmlreader.XMLReader* 메서드), 1452
getprotobyname() (*socket* 모듈), 1227
getproxies() (*urllib.request* 모듈), 1481
getPublicId() (*xml.sax.xmlreader.InputSource* 메서드), 1453
getPublicId() (*xml.sax.xmlreader.Locator* 메서드), 1453
getpwall() (*pwd* 모듈), 2270
getpwnam() (*pwd* 모듈), 2270
getpwuid() (*pwd* 모듈), 2270
getQNameByName() (*xml.sax.xmlreader.AttributesNS* 메서드), 1454
getQNames() (*xml.sax.xmlreader.AttributesNS* 메서드), 1454
getquota() (*imaplib.IMAP4* 메서드), 1530
getquotaroot() (*imaplib.IMAP4* 메서드), 1530
getrandbits() (*random* 모듈), 398
getrandbits() (*random.Random* 메서드), 401
getrandom() (*os* 모듈), 755
getreader() (*codecs* 모듈), 200
getrecursionlimit() (*sys* 모듈), 1992
getrefcount() (*sys* 모듈), 1992
GetReparseDeferralEnabled() (*xml.parsers.expat.xmlparser* 메서드), 1457
getresgid() (*os* 모듈), 695
getresponse() (*http.client.HTTPConnection* 메서드), 1513
getresuid() (*os* 모듈), 695
getrlimit() (*resource* 모듈), 2278
getroot() (*xml.etree.ElementTree.ElementTree* 메서드), 1422
getrusage() (*resource* 모듈), 2281
getsampwidth() (*wave.Wave_read* 메서드), 1606
getscreen() (*turtle* 모듈), 1701
getservbyname() (*socket* 모듈), 1227
getservbyport() (*socket* 모듈), 1227
GetSetDescriptorType (*types* 모듈), 315
getshapes() (*turtle* 모듈), 1707
getsid() (*os* 모듈), 697
getsignal() (*signal* 모듈), 1292
getsitepackages() (*site* 모듈), 2111
getsize() (*os.path* 모듈), 488
getsizeof() (*sys* 모듈), 1992
getsockname() (*socket.socket* 메서드), 1232
getsockopt() (*socket.socket* 메서드), 1232
getsource() (*inspect* 모듈), 2087
getsourcefile() (*inspect* 모듈), 2086
getsourcelines() (*inspect* 모듈), 2087
getstate() (*codecs.IncrementalDecoder* 메서드), 205
getstate() (*codecs.IncrementalEncoder* 메서드), 205
getstate() (*random* 모듈), 398
getstate() (*random.Random* 메서드), 401
getstatusoutput() (*subprocess* 모듈), 1094
getstr() (*curses.window* 메서드), 967
getSubject() (*logging.handlers.SMTPHandler* 메서드), 824
getswitchinterval() (*sys* 모듈), 1992
getSystemId() (*xml.sax.xmlreader.InputSource* 메서드), 1453
getSystemId() (*xml.sax.xmlreader.Locator* 메서드), 1453
getsyx() (*curses* 모듈), 959
gettarinfo() (*tarfile.TarFile* 메서드), 630
gettempdir() (*tempfile* 모듈), 504
gettempdirb() (*tempfile* 모듈), 504
gettempprefix() (*tempfile* 모듈), 504
gettempprefixb() (*tempfile* 모듈), 504
getTestCaseNames() (*unittest.TestLoader* 메서드), 1823
gettext
    module, 1609
gettext() (*gettext* 모듈), 1610
gettext() (*gettext.GNUTranslations* 메서드), 1613
gettext() (*gettext.NullTranslations* 메서드), 1611
gettext() (*locale* 모듈), 1625
gettimeout() (*socket.socket* 메서드), 1232
gettrace() (*sys* 모듈), 1993
gettrace() (*threading* 모듈), 998
getturtle() (*turtle* 모듈), 1701

getType() (*xml.sax.xmlreader.Attributes* 메서드), 1454
getuid() (*os* 모듈), 695
getunicodeinternedsize() (*sys* 모듈), 1991
geturl() (*http.client.HTTPResponse* 메서드), 1515
geturl() (*urllib.parse.urllib.parse.SplitResult* 메서드), 1501
geturl() (*urllib.response.addinfourl* 메서드), 1495
getuser() (*getpass* 모듈), 954
getuserbase() (*site* 모듈), 2111
getusersitepackages() (*site* 모듈), 2111
getvalue() (*io.BytesIO* 메서드), 765
getvalue() (*io.StringIO* 메서드), 769
getValue() (*xml.sax.xmlreader.Attributes* 메서드), 1454
getValueByQName() (*xml.sax.xmlreader.AttributesNS* 메서드), 1454
getwch() (*msvcrt* 모듈), 2248
getwche() (*msvcrt* 모듈), 2248
getweakrefcount() (*weakref* 모듈), 305
getweakrefs() (*weakref* 모듈), 305
getwelcome() (*ftplib.FTP* 메서드), 1519
getwelcome() (*poplib.POP3* 메서드), 1525
getwin() (*curses* 모듈), 959
getwindowsversion() (*sys* 모듈), 1993
getwriter() (*codecs* 모듈), 200
getxattr() (*os* 모듈), 737
getyx() (*curses.window* 메서드), 967
gid (*tarfile.TarInfo*의 속성), 632
GIL, **2308**
gil (*sys.flags*의 속성), 1988
glob
    module, 506
    모듈, 509
glob() (*glob* 모듈), 506
glob() (*pathlib.Path* 메서드), 477
Global (*ast* 클래스), 2193
global interpreter lock (전역 인터프리터 록), **2308**
global_enum() (*enum* 모듈), 342
globals()
    built-in function, 18
globs (*doctest.DocTest*의 속성), 1792
gmtime() (*time* 모듈), 774
gname (*tarfile.TarInfo*의 속성), 632
GNOME, 1613
GNU_FORMAT (*tarfile* 모듈), 627
gnu_getopt() (*getopt* 모듈), 2290
GNUTranslations (*gettext* 클래스), 1613
GNUTYPE_LONGLINK (*tarfile* 모듈), 626
GNUTYPE_LONGNAME (*tarfile* 모듈), 626
GNUTYPE_SPARSE (*tarfile* 모듈), 626
go() (*tkinter.filedialog.FileDialog* 메서드), 1643
got (*doctest.DocTestFailure*의 속성), 1798
goto() (*turtle* 모듈), 1686
grantpt() (*os* 모듈), 702
graphlib
    module, 343
GREATER (*token* 모듈), 2207
GREATEREQUAL (*token* 모듈), 2208
greedy (*compression.zstd.Strategy*의 속성), 592
GRND_NONBLOCK (*os* 모듈), 756
GRND_RANDOM (*os* 모듈), 756
Group (*email.headerregistry* 클래스), 1331
group() (*pathlib.Path* 메서드), 482
group() (*re.Match* 메서드), 156
groupby() (*itertools* 모듈), 429
groupdict() (*re.Match* 메서드), 158
groupindex (*re.Pattern*의 속성), 156
groups (*email.headerregistry.AddressHeader*의 속성), 1328
groups (*re.Pattern*의 속성), 156
groups() (*re.Match* 메서드), 158
grp
    module, 2270
GS (*curses.ascii* 모듈), 985
Gt (*ast* 클래스), 2172
gt() (*operator* 모듈), 451
GtE (*ast* 클래스), 2172
guess_all_extensions() (*mimetypes* 모듈), 1390
guess_all_extensions() (*mimetypes.MimeTypes* 메서드), 1392
guess_extension() (*mimetypes* 모듈), 1390
guess_extension() (*mimetypes.MimeTypes* 메서드), 1392
guess_file_type() (*mimetypes* 모듈), 1390
guess_file_type() (*mimetypes.MimeTypes* 메서드), 1392
guess_scheme() (*wsgiref.util* 모듈), 1468
guess_type() (*mimetypes* 모듈), 1390
guess_type() (*mimetypes.MimeTypes* 메서드), 1392
GUI, 1627
gzip
    module, 598
gzip 명령줄 옵션
    --best, 601
    -d, 601
    --decompress, 601
    --fast, 601
    file, 601
    -h, 601
    --help, 601
GzipFile (*gzip* 클래스), 598

H

-h
    ast 명령줄 옵션, 2200
    calendar 명령줄 옵션, 268
    dis 명령줄 옵션, 2223
    gzip 명령줄 옵션, 601
    idle 명령줄 옵션, 1674
    json 명령줄 옵션, 1370
    mimetypes 명령줄 옵션, 1393
    python--m-sqlite3-[-h]-[-v]-[filename]-[sql] 명령줄 옵션, 574
    random 명령줄 옵션, 405

timeit 명령줄 옵션, 1950
tokenize 명령줄 옵션, 2212
uuid 명령줄 옵션, 1545
zipapp 명령줄 옵션, 1977
halfdelay() (*curses* 모듈), 960
Handle (*asyncio* 클래스), 1175
handle() (*http.server.BaseHTTPRequestHandler* 메서드), 1558
handle() (*logging.Handler* 메서드), 791
handle() (*logging.handlers.QueueListener* 메서드), 827
handle() (*logging.Logger* 메서드), 789
handle() (*logging.NullHandler* 메서드), 815
handle() (*socketserver.BaseRequestHandler* 메서드), 1551
handle() (*wsgiref.simple_server.WSGIRequestHandler* 메서드), 1472
handle_charref() (*html.parser.HTMLParser* 메서드), 1403
handle_comment() (*html.parser.HTMLParser* 메서드), 1403
handle_data() (*html.parser.HTMLParser* 메서드), 1403
handle_decl() (*html.parser.HTMLParser* 메서드), 1404
handle_defect() (*email.policy.Policy* 메서드), 1321
handle_endtag() (*html.parser.HTMLParser* 메서드), 1403
handle_entityref() (*html.parser.HTMLParser* 메서드), 1403
handle_error() (*socketserver.BaseServer* 메서드), 1551
handle_expect_100() (*http.server.BaseHTTPRequestHandler* 메서드), 1558
handle_one_request() (*http.server.BaseHTTPRequestHandler* 메서드), 1558
handle_pi() (*html.parser.HTMLParser* 메서드), 1404
handle_request() (*socketserver.BaseServer* 메서드), 1549
handle_request() (*xmlrpc.server.CGIXMLRPCRequestHandler* 메서드), 1588
handle_startendtag() (*html.parser.HTMLParser* 메서드), 1403
handle_starttag() (*html.parser.HTMLParser* 메서드), 1403
handle_timeout() (*socketserver.BaseServer* 메서드), 1551
handleError() (*logging.Handler* 메서드), 791
handleError() (*logging.handlers.SocketHandler* 메서드), 819
Handler (*logging* 클래스), 790
handlers (*logging.Logger*의 속성), 786
Handlers (*signal* 클래스), 1289
hardlink_to() (*pathlib.Path* 메서드), 481
--hardlink-dupes
compileall 명령줄 옵션, 2220
harmonic_mean() (*statistics* 모듈), 409
HAS_ALPN (*ssl* 모듈), 1251
has_children() (*symtable.SymbolTable* 메서드), 2202
has_colors() (*curses* 모듈), 959
has_default() (*typing.ParamSpec* 메서드), 1748
has_default() (*typing.TypeVar* 메서드), 1744
has_default() (*typing.TypeVarTuple* 메서드), 1746
has_dualstack_ipv6() (*socket* 모듈), 1224
HAS_ECDH (*ssl* 모듈), 1251
has_extended_color_support() (*curses* 모듈), 960
has_extn() (*smtplib.SMTP* 메서드), 1537
has_header() (*csv.Sniffer* 메서드), 644
has_header() (*urllib.request.Request* 메서드), 1485
has_ic() (*curses* 모듈), 960
has_il() (*curses* 모듈), 960
has_ipv6 (*socket* 모듈), 1220
has_key() (*curses* 모듈), 960
has_location (*importlib.machinery.ModuleSpec*의 속성), 2139
HAS_NEVER_CHECK_COMMON_NAME (*ssl* 모듈), 1251
has_nonstandard_attr() (*http.cookiejar.Cookie* 메서드), 1575
HAS_NPN (*ssl* 모듈), 1252
has_option() (*configparser.ConfigParser* 메서드), 663
has_option() (*optparse.OptionParser* 메서드), 945
HAS_PHA (*ssl* 모듈), 1252
HAS_PSK (*ssl* 모듈), 1252
has_section() (*configparser.ConfigParser* 메서드), 663
HAS_SNI (*ssl* 모듈), 1252
HAS_SSLv2 (*ssl* 모듈), 1252
HAS_SSLv3 (*ssl* 모듈), 1252
has_ticket (*ssl.SSLSession*의 속성), 1275
HAS_TLSv1 (*ssl* 모듈), 1252
HAS_TLSv1_1 (*ssl* 모듈), 1252
HAS_TLSv1_2 (*ssl* 모듈), 1252
HAS_TLSv1_3 (*ssl* 모듈), 1252
hasarg (*dis* 모듈), 2244
hasattr()
built-in function, 18
hasAttribute() (*xml.dom.Element* 메서드), 1432
hasAttributeNS() (*xml.dom.Element* 메서드), 1432
hasAttributes() (*xml.dom.Node* 메서드), 1429
hasChildNodes() (*xml.dom.Node* 메서드), 1430
hascompare (*dis* 모듈), 2245
hasconst (*dis* 모듈), 2244
hasexc (*dis* 모듈), 2245
hasFeature() (*xml.dom.DOMImplementation* 메서드), 1428
hasfree (*dis* 모듈), 2244
hash
built-in function, 49
hash()
built-in function, 18

hash-based pyc (해시 기반 *pyc*), **2308**
hash_bits (*sys.hash_info*의 속성), 1995
hash_info (*sys* 모듈), 1994
hash_log (*compression.zstd.CompressionParameter* 의 속성), 589
hash_randomization (*sys.flags*의 속성), 1988
Hashable (*collections.abc* 클래스), 290
Hashable (*typing* 클래스), 1771
hashable (해시 가능), **2308**
hasHandlers() (*logging.Logger* 메서드), 789
hash.block_size (*hashlib* 모듈), 675
hash.digest_size (*hashlib* 모듈), 675
hashlib
    module, 673
hasjabs (*dis* 모듈), 2245
hasjrel (*dis* 모듈), 2245
hasjump (*dis* 모듈), 2244
haslocal (*dis* 모듈), 2245
hasname (*dis* 모듈), 2244
HAVE_ARGUMENT (*opcode*), 2242
HAVE_CONTEXTVAR (*decimal* 모듈), 383
HAVE_DOCSTRINGS (*test.support* 모듈), 1898
HAVE_THREADS (*decimal* 모듈), 383
HCI_CHANNEL_CONTROL (*socket* 모듈), 1221
HCI_CHANNEL_LOGGING (*socket* 모듈), 1221
HCI_CHANNEL_MONITOR (*socket* 모듈), 1221
HCI_CHANNEL_RAW (*socket* 모듈), 1221
HCI_CHANNEL_USER (*socket* 모듈), 1221
HCI_DATA_DIR (*socket* 모듈), 1221
HCI_DEV_NONE (*socket* 모듈), 1221
HCI_FILTER (*socket* 모듈), 1221
HCI_TIME_STAMP (*socket* 모듈), 1221
Header (*email.header* 클래스), 1352
header_encode() (*email.charset.Charset* 메서드), 1355
header_encode_lines() (*email.charset.Charset* 메서드), 1355
header_encoding (*email.charset.Charset*의 속성), 1355
header_factory (*email.policy.EmailPolicy*의 속성), 1322
header_fetch_parse() (*email.policy.Compat32* 메서드), 1324
header_fetch_parse() (*email.policy.EmailPolicy* 메서드), 1323
header_fetch_parse() (*email.policy.Policy* 메서드), 1321
header_items() (*urllib.request.Request* 메서드), 1485
header_max_count() (*email.policy.EmailPolicy* 메서드), 1323
header_max_count() (*email.policy.Policy* 메서드), 1321
header_offset (*zipfile.ZipInfo*의 속성), 621
header_source_parse() (*email.policy.Compat32* 메서드), 1324
header_source_parse() (*email.policy.EmailPolicy* 메서드), 1323
header_source_parse() (*email.policy.Policy* 메서드), 1321
header_store_parse() (*email.policy.Compat32* 메서드), 1324
header_store_parse() (*email.policy.EmailPolicy* 메서드), 1323
header_store_parse() (*email.policy.Policy* 메서드), 1321
HeaderDefect, 1325
HeaderError, 625
HeaderParseError, 1325
HeaderParser (*email.parser* 클래스), 1314
HeaderRegistry (*email.headerregistry* 클래스), 1329
headers (*http.client.HTTPResponse*의 속성), 1515
headers (*http.server.BaseHTTPRequestHandler*의 속성), 1557
headers (*urllib.error.HTTPError*의 속성), 1505
headers (*urllib.response.addinfourl*의 속성), 1495
Headers (*wsgiref.headers* 클래스), 1470
headers (*xmlrpc.client.ProtocolError*의 속성), 1581
HeaderWriteError, 1325
heading() (*tkinter.ttk.Treeview* 메서드), 1659
heading() (*turtle* 모듈), 1691
heapify() (*heapq* 모듈), 294
heapify_max() (*heapq* 모듈), 294
heapmin() (*msvcrt* 모듈), 2248
heappop() (*heapq* 모듈), 293
heappop_max() (*heapq* 모듈), 294
heappush() (*heapq* 모듈), 293
heappush_max() (*heapq* 모듈), 294
heappushpop() (*heapq* 모듈), 294
heappushpop_max() (*heapq* 모듈), 294
heapq
    module, 293
heapreplace() (*heapq* 모듈), 294
heapreplace_max() (*heapq* 모듈), 294
helo() (*smtplib.SMTP* 메서드), 1537
--help
    ast 명령줄 옵션, 2200
    calendar 명령줄 옵션, 268
    dis 명령줄 옵션, 2223
    gzip 명령줄 옵션, 601
    json 명령줄 옵션, 1370
    mimetypes 명령줄 옵션, 1393
    python--m-sqlite3-[-h]-[-v]-[filename]-[sql] 명령줄 옵션, 574
    random 명령줄 옵션, 405
    timeit 명령줄 옵션, 1950
    tokenize 명령줄 옵션, 2212
    trace 명령줄 옵션, 1952
    uuid 명령줄 옵션, 1545
    zipapp 명령줄 옵션, 1977
help (*optparse.Option*의 속성), 941
help (*pdb command*), 1933
help()
    built-in function, 18
herror, 1216
hex (*uuid.UUID*의 속성), 1543

hex()
    built-in function, 18
hex() (*bytearray* 메서드), 69
hex() (*bytes* 메서드), 68
hex() (*float* 메서드), 43
hex() (*memoryview* 메서드), 86
hexadecimal
    literals, 39
hexdigest() (*hashlib.hash* 메서드), 675
hexdigest() (*hashlib.shake* 메서드), 676
hexdigest() (*hmac.HMAC* 메서드), 685
hexdigits (*string* 모듈), 125
hexlify() (*binascii* 모듈), 1399
hexversion (*sys* 모듈), 1995
hidden() (*curses.panel.Panel* 메서드), 988
hide() (*curses.panel.Panel* 메서드), 988
hide() (*tkinter.ttk.Notebook* 메서드), 1654
hide_cookie2 (*http.cookiejar.CookiePolicy* 의 속성), 1572
hideturtle() (*turtle* 모듈), 1696
HierarchyRequestErr, 1434
HIGH_PRIORITY_CLASS (*subprocess* 모듈), 1090
HIGHEST_PROTOCOL (*pickle* 모듈), 525
hits (*bdb.Breakpoint* 의 속성), 1920
HKEY_CLASSES_ROOT (*winreg* 모듈), 2255
HKEY_CURRENT_CONFIG (*winreg* 모듈), 2256
HKEY_CURRENT_USER (*winreg* 모듈), 2255
HKEY_DYN_DATA (*winreg* 모듈), 2256
HKEY_LOCAL_MACHINE (*winreg* 모듈), 2256
HKEY_PERFORMANCE_DATA (*winreg* 모듈), 2256
HKEY_USERS (*winreg* 모듈), 2256
hline() (*curses.window* 메서드), 967
hls_to_rgb() (*colorsys* 모듈), 1608
hmac
    module, 685
HMAC (*hmac* 클래스), 685
HOME, 488, 1629
home() (*pathlib.Path* 의 클래스 메서드), 473
home() (*turtle* 모듈), 1687
HOMEDRIVE, 488
HOMEPATH, 488
hook_compressed() (*fileinput* 모듈), 956
hook_encoded() (*fileinput* 모듈), 956
host (*urllib.request.Request* 의 속성), 1484
hostmask (*ipaddress.IPv4Network* 의 속성), 1596
hostmask (*ipaddress.IPv6Network* 의 속성), 1599
hostname_checks_common_name (*ssl.SSLContext* 의 속성), 1265
hosts (*netrc.netrc* 의 속성), 670
hosts() (*ipaddress.IPv4Network* 메서드), 1597
hosts() (*ipaddress.IPv6Network* 메서드), 1599
hour (*datetime.datetime* 의 속성), 232
hour (*datetime.time* 의 속성), 241
HRESULT (*ctypes* 클래스), 874
hresult (*ctypes.COMError* 의 속성), 878
hStdError (*subprocess.STARTUPINFO* 의 속성), 1088
hStdInput (*subprocess.STARTUPINFO* 의 속성), 1088
hStdOutput (*subprocess.STARTUPINFO* 의 속성), 1088
hsv_to_rgb() (*colorsys* 모듈), 1608
HT (*curses.ascii* 모듈), 984
ht() (*turtle* 모듈), 1696
HTML, 1402, 1495
html
    module, 1401
html5 (*html.entities* 모듈), 1406
HTMLCalendar (*calendar* 클래스), 263
HtmlDiff (*difflib* 클래스), 165
html.entities
    module, 1406
html.parser
    module, 1402
HTMLParser (*html.parser* 클래스), 1402
htonl() (*socket* 모듈), 1227
htons() (*socket* 모듈), 1227
HTTP
    http (표준 모듈), 1506
    http.client (표준 모듈), 1510
    프로토콜, 1495, 1506, 1510, 1556
http
    module, 1506
HTTP (*email.policy* 모듈), 1324
http_error_301() (*urllib.request.HTTPRedirectHandler* 메서드), 1488
http_error_302() (*urllib.request.HTTPRedirectHandler* 메서드), 1488
http_error_303() (*urllib.request.HTTPRedirectHandler* 메서드), 1488
http_error_307() (*urllib.request.HTTPRedirectHandler* 메서드), 1488
http_error_308() (*urllib.request.HTTPRedirectHandler* 메서드), 1488
http_error_401() (*urllib.request.HTTPBasicAuthHandler* 메서드), 1489
http_error_401() (*urllib.request.HTTPDigestAuthHandler* 메서드), 1490
http_error_407() (*urllib.request.ProxyBasicAuthHandler* 메서드), 1490
http_error_407() (*urllib.request.ProxyDigestAuthHandler* 메서드), 1490
http_error_auth_reqed() (*urllib.request.AbstractBasicAuthHandler* 메서드), 1489
http_error_auth_reqed() (*urllib.request.AbstractDigestAuthHandler* 메서드), 1490

http_error_default() (*urllib.request.BaseHandler* 메서드), 1487
http_open() (*urllib.request.HTTPHandler* 메서드), 1490
HTTP_PORT (*http.client* 모듈), 1512
http_response() (*urllib.request.HTTPErrorProcessor* 메서드), 1491
http_version (*wsgiref.handlers.BaseHandler* 의 속성), 1476
HTTPBasicAuthHandler (*urllib.request* 클래스), 1483
http.client
    module, 1510
HTTPConnection (*http.client* 클래스), 1510
http.cookiejar
    module, 1567
HTTPCookieProcessor (*urllib.request* 클래스), 1482
http.cookies
    module, 1563
httpd, 1556
HTTPDefaultErrorHandler (*urllib.request* 클래스), 1482
HTTPDigestAuthHandler (*urllib.request* 클래스), 1483
HTTPError, 1505
HTTPErrorProcessor (*urllib.request* 클래스), 1484
HTTPException, 1511
HTTPHandler (*logging.handlers* 클래스), 825
HTTPHandler (*urllib.request* 클래스), 1483
HTTPMessage (*http.client* 클래스), 1517
HTTPMethod (*http* 클래스), 1509
httponly (*http.cookies.Morsel*의 속성), 1565
HTTPPasswordMgr (*urllib.request* 클래스), 1482
HTTPPasswordMgrWithDefaultRealm (*urllib.request* 클래스), 1482
HTTPPasswordMgrWithPriorAuth (*urllib.request* 클래스), 1482
HTTPRedirectHandler (*urllib.request* 클래스), 1482
HTTPResponse (*http.client* 클래스), 1511
https_open() (*urllib.request.HTTPSHandler* 메서드), 1490
HTTPS_PORT (*http.client* 모듈), 1512
https_response() (*urllib.request.HTTPErrorProcessor* 메서드), 1491
HTTPSConnection (*http.client* 클래스), 1510
http.server
    module, 1556
    보안, 1563
HTTPServer (*http.server* 클래스), 1556
http.server 명령줄 옵션
    -b, 1562
    --bind, 1562
    --cgi, 1562
    -d, 1562
    --directory, 1562
    -p, 1562
    port, 1562
    --protocol, 1562
    --tls-cert, 1562
    --tls-key, 1563
    --tls-password-file, 1563
HTTPSHandler (*urllib.request* 클래스), 1483
HTTPSServer (*http.server* 클래스), 1556
HTTPStatus (*http* 클래스), 1507
HV_GUID_BROADCAST (*socket* 모듈), 1222
HV_GUID_CHILDREN (*socket* 모듈), 1222
HV_GUID_LOOPBACK (*socket* 모듈), 1222
HV_GUID_PARENT (*socket* 모듈), 1222
HV_GUID_WILDCARD (*socket* 모듈), 1222
HV_GUID_ZERO (*socket* 모듈), 1222
HV_PROTOCOL_RAW (*socket* 모듈), 1222
HVSOCKET_ADDRESS_FLAG_PASSTHRU (*socket* 모듈), 1222
HVSOCKET_CONNECT_TIMEOUT (*socket* 모듈), 1222
HVSOCKET_CONNECT_TIMEOUT_MAX (*socket* 모듈), 1222
HVSOCKET_CONNECTED_SUSPEND (*socket* 모듈), 1222
hypot() (*math* 모듈), 357

I

-i
    ast 명령줄 옵션, 2200
    compileall 명령줄 옵션, 2219
    idle 명령줄 옵션, 1674
    random 명령줄 옵션, 406
I (*re* 모듈), 149
I/O control
    buffering, 25, 1232
I/O 제어
    POSIX, 2271
    tty, 2271
    UNIX, 2275
iadd() (*operator* 모듈), 457
iand() (*operator* 모듈), 457
iconcat() (*operator* 모듈), 457
id (*concurrent.interpreters.Interpreter*의 속성), 1075
id (*concurrent.interpreters.Queue*의 속성), 1076
id (*ssl.SSLSession*의 속성), 1275
id()
    built-in function, 19
id() (*unittest.TestCase* 메서드), 1818
idcok() (*curses.window* 메서드), 967
ident (*select.kevent*의 속성), 1282
ident (*threading.Thread*의 속성), 1003
identchars (*cmd.Cmd*의 속성), 990
identify() (*tkinter.ttk.Notebook* 메서드), 1654
identify() (*tkinter.ttk.Treeview* 메서드), 1659
identify() (*tkinter.ttk.Widget* 메서드), 1651
identify_column() (*tkinter.ttk.Treeview* 메서드), 1659
identify_element() (*tkinter.ttk.Treeview* 메서드), 1660
identify_region() (*tkinter.ttk.Treeview* 메서드), 1659
identify_row() (*tkinter.ttk.Treeview* 메서드), 1659

IDLE, 1666, **2309**
idle 명령줄 옵션
    -, 1674
    -c, 1673
    -d, 1674
    -e, 1674
    -h, 1674
    -i, 1674
    -r, 1674
    -s, 1674
    -t, 1674
idle() (*imaplib.IMAP4* 메서드), 1530
IDLE_PRIORITY_CLASS (*subprocess* 모듈), 1090
idlelib
    module, 1677
IDLESTARTUP, 1673, 1674
idlok() (*curses.window* 메서드), 968
IEEE_CONTEXT_MAX_BITS (*decimal* 모듈), 383
IEEEContext() (*decimal* 모듈), 377
if
    statement, 37
If (*ast* 클래스), 2179
if_indextoname() (*socket* 모듈), 1229
if_nameindex() (*socket* 모듈), 1229
if_nametoindex() (*socket* 모듈), 1229
IfExp (*ast* 클래스), 2172
ifloordiv() (*operator* 모듈), 457
iglob() (*glob* 모듈), 507
ignorableWhitespace() (*xml.sax.handler.ContentHandler* 메서드), 1448
ignore
    에러 처리기의 이름, 202
ignore (*bdb.Breakpoint*의 속성), 1920
ignore (*pdb command*), 1934
IGNORE (*tkinter.messagebox* 모듈), 1646
ignore_environment (*sys.flags*의 속성), 1988
ignore_errors() (*codecs* 모듈), 203
IGNORE_EXCEPTION_DETAIL (*doctest* 모듈), 1785
ignore_patterns() (*shutil* 모듈), 513
ignore_warnings() (*test.support.warnings_helper* 모듈), 1912
IGNORECASE (*re* 모듈), 149
--ignore-dir
    trace 명령줄 옵션, 1953
--ignore-module
    trace 명령줄 옵션, 1953
IISCGIHandler (*wsgiref.handlers* 클래스), 1473
IllegalMonthError, 267
IllegalWeekdayError, 267
ilshift() (*operator* 모듈), 457
imag (*numbers.Complex*의 속성), 347
imag (*sys.hash_info*의 속성), 1995
imap() (*multiprocessing.pool.Pool* 메서드), 1043
IMAP4
    프로토콜, 1527
IMAP4 (*imaplib* 클래스), 1527
IMAP4_SSL
    프로토콜, 1527
IMAP4_SSL (*imaplib* 클래스), 1527
IMAP4_stream
    프로토콜, 1527
IMAP4_stream (*imaplib* 클래스), 1528
IMAP4.abort, 1527
IMAP4.error, 1527
IMAP4.readonly, 1527
imap_unordered() (*multiprocessing.pool.Pool* 메서드), 1043
imaplib
    module, 1527
imatmul() (*operator* 모듈), 457
imghdr
    module, 2295
immedok() (*curses.window* 메서드), 968
immortal (불멸), **2309**
immutable
    sequence types, 49
immutable (불변), **2309**
imod() (*operator* 모듈), 457
imp
    module, 2295
impl_detail() (*test.support* 모듈), 1902
implementation (*sys* 모듈), 1995
import
    statement, 33
Import (*ast* 클래스), 2178
import path (임포트 경로), **2309**
import_fresh_module() (*test.support.import_helper* 모듈), 1910
IMPORT_FROM (*opcode*), 2237
import_module() (*importlib* 모듈), 2127
import_module() (*test.support.import_helper* 모듈), 1911
IMPORT_NAME (*opcode*), 2237
importer (임포터), **2309**
ImportError, 113
ImportFrom (*ast* 클래스), 2179
importing (임포팅), **2309**
importlib
    module, 2125
importlib.abc
    module, 2128
importlib.machinery
    module, 2135
importlib.metadata
    module, 2150
importlib.resources
    module, 2145
importlib.resources.abc
    module, 2148
importlib.util
    module, 2140
ImportWarning, 120
ImproperConnectionState, 1511
imul() (*operator* 모듈), 457
in

operator, 38, 47
In (*ast* 클래스), 2172
in_dll() (*ctypes._CData* 메서드), 871
in_table_a1() (*stringprep* 모듈), 183
in_table_b1() (*stringprep* 모듈), 183
in_table_c3() (*stringprep* 모듈), 184
in_table_c4() (*stringprep* 모듈), 184
in_table_c5() (*stringprep* 모듈), 184
in_table_c6() (*stringprep* 모듈), 184
in_table_c7() (*stringprep* 모듈), 184
in_table_c8() (*stringprep* 모듈), 184
in_table_c9() (*stringprep* 모듈), 184
in_table_c11() (*stringprep* 모듈), 183
in_table_c11_c12() (*stringprep* 모듈), 183
in_table_c12() (*stringprep* 모듈), 183
in_table_c21() (*stringprep* 모듈), 183
in_table_c21_c22() (*stringprep* 모듈), 184
in_table_c22() (*stringprep* 모듈), 184
in_table_d1() (*stringprep* 모듈), 184
in_table_d2() (*stringprep* 모듈), 184
in_transaction (*sqlite3.Connection*의 속성), 567
inch() (*curses.window* 메서드), 968
include() (*xml.etree.ElementInclude* 모듈), 1418
--include-attributes
ast 명령줄 옵션, 2200
inclusive (*tracemalloc.DomainFilter*의 속성), 1960
inclusive (*tracemalloc.Filter*의 속성), 1961
Incomplete, 1400
IncompleteRead, 1511
IncompleteReadError, 1154
increment_lineno() (*ast* 모듈), 2197
IncrementalDecoder (*codecs* 클래스), 205
incrementaldecoder (*codecs.CodecInfo*의 속성), 199
IncrementalEncoder (*codecs* 클래스), 204
incrementalencoder (*codecs.CodecInfo*의 속성), 199
IncrementalNewlineDecoder (*io* 클래스), 770
IncrementalParser (*xml.sax.xmlreader* 클래스), 1450
--indent
ast 명령줄 옵션, 2200
json 명령줄 옵션, 1370
indent (*doctest.Example*의 속성), 1793
indent (*reprlib.Repr*의 속성), 326
INDENT (*token* 모듈), 2205
indent() (*textwrap* 모듈), 178
indent() (*xml.etree.ElementTree* 모듈), 1415
IndentationError, 117
--indentlevel
pickletools 명령줄 옵션, 2246
index (*inspect.FrameInfo*의 속성), 2094
index (*inspect.Traceback*의 속성), 2095
index() (*array.array* 메서드), 303
index() (*bytearray* 메서드), 72
index() (*bytes* 메서드), 72
index() (*collections.abc.ByteString* 메서드), 290
index() (*collections.deque* 메서드), 276
index() (*memoryview* 메서드), 89
index() (*multiprocessing.shared_memory.ShareableList* 메서드), 1062
index() (*operator* 모듈), 452
index() (*sequence* 메서드), 48
index() (*str* 메서드), 57
index() (*tkinter.ttk.Notebook* 메서드), 1654
index() (*tkinter.ttk.Treeview* 메서드), 1660
IndexError, 114
indexOf() (*operator* 모듈), 454
IndexSizeErr, 1434
INDIRECT (*inspect.BufferFlags*의 속성), 2100
inet_aton() (*socket* 모듈), 1227
inet_ntoa() (*socket* 모듈), 1228
inet_ntop() (*socket* 모듈), 1228
inet_pton() (*socket* 모듈), 1228
Inexact (*decimal* 클래스), 384
inf (*cmath* 모듈), 363
inf (*math* 모듈), 359
inf (*sys.hash_info*의 속성), 1994
infile
json 명령줄 옵션, 1370
infile (*shlex.shlex*의 속성), 2266
Infinity, 16
infj (*cmath* 모듈), 363
--info
zipapp 명령줄 옵션, 1977
INFO (*logging* 모듈), 790
info (*pathlib.Path*의 속성), 476
INFO (*tkinter.messagebox* 모듈), 1646
info() (*dis.Bytecode* 메서드), 2224
info() (*gettext.NullTranslations* 메서드), 1612
info() (*http.client.HTTPResponse* 메서드), 1515
info() (*logging* 모듈), 798
info() (*logging.Logger* 메서드), 788
info() (*urllib.response.addinfourl* 메서드), 1495
infolist() (*zipfile.ZipFile* 메서드), 614
.ini
파일, 649
ini 파일, 649
init() (*mimetypes* 모듈), 1390
init_color() (*curses* 모듈), 960
init_pair() (*curses* 모듈), 960
inited (*mimetypes* 모듈), 1391
initgroups() (*os* 모듈), 695
initial_indent (*textwrap.TextWrapper*의 속성), 179
initscr() (*curses* 모듈), 960
InitVar (*dataclasses* 클래스), 2037
inode() (*os.DirEntry* 메서드), 723
input()
built-in function, 19
input() (*fileinput* 모듈), 955
input_charset (*email.charset.Charset*의 속성), 1354
input_codec (*email.charset.Charset*의 속성), 1355
InputSource (*xml.sax.xmlreader* 클래스), 1451
InputStream (*wsgiref.types* 클래스), 1476
insch() (*curses.window* 메서드), 968
insdelln() (*curses.window* 메서드), 968

insert() (*array.array* 메서드), 303
insert() (*collections.deque* 메서드), 276
insert() (*sequence* 메서드), 50
insert() (*tkinter.ttk.Notebook* 메서드), 1654
insert() (*tkinter.ttk.Treeview* 메서드), 1660
insert() (*xml.etree.ElementTree.Element* 메서드), 1420
insert_text() (*readline* 모듈), 185
insertBefore() (*xml.dom.Node* 메서드), 1430
insertln() (*curses.window* 메서드), 968
insnstr() (*curses.window* 메서드), 968
insort() (*bisect* 모듈), 298
insort_left() (*bisect* 모듈), 298
insort_right() (*bisect* 모듈), 298
inspect
    module, 2081
inspect (*sys.flags*의 속성), 1988
inspect 명령줄 옵션
    --details, 2100
InspectLoader (*importlib.abc* 클래스), 2130
insstr() (*curses.window* 메서드), 968
install() (*gettext* 모듈), 1611
install() (*gettext.NullTranslations* 메서드), 1612
install_opener() (*urllib.request* 모듈), 1479
install_scripts() (*venv.EnvBuilder* 메서드), 1972
installHandler() (*unittest* 모듈), 1831
instate() (*tkinter.ttk.Widget* 메서드), 1651
instr() (*curses.window* 메서드), 968
instream (*shlex.shlex*의 속성), 2266
Instruction (*dis* 클래스), 2227
INSTRUCTION (*monitoring event*), 2009
Instruction.arg (*dis* 모듈), 2227
Instruction.argrepr (*dis* 모듈), 2227
Instruction.argval (*dis* 모듈), 2227
Instruction.baseopcode (*dis* 모듈), 2227
Instruction.baseopname (*dis* 모듈), 2227
Instruction.cache_info (*dis* 모듈), 2228
Instruction.cache_offset (*dis* 모듈), 2228
Instruction.end_offset (*dis* 모듈), 2228
Instruction.is_jump_target (*dis* 모듈), 2228
Instruction.jump_target (*dis* 모듈), 2228
Instruction.line_number (*dis* 모듈), 2228
Instruction.offset (*dis* 모듈), 2227
Instruction.oparg (*dis* 모듈), 2227
Instruction.opcode (*dis* 모듈), 2227
Instruction.opname (*dis* 모듈), 2227
Instruction.positions (*dis* 모듈), 2228
Instruction.start_offset (*dis* 모듈), 2227
Instruction.starts_line (*dis* 모듈), 2228
int
    built-in function, 39
int (*uuid.UUID*의 속성), 1543
int (내장 클래스), 19
Int2AP() (*imaplib* 모듈), 1528
int_info (*sys* 모듈), 1996
int_max_str_digits (*sys.flags*의 속성), 1988
integer
    literals, 39
    object, 38
    types, operations on, 40
--integer
    random 명령줄 옵션, 406
Integral (*numbers* 클래스), 348
IntegrityError, 572
IntEnum (*enum* 클래스), 335
interact (*pdb command*), 1936
interact() (*code* 모듈), 2113
interact() (*code.InteractiveConsole* 메서드), 2115
Interactive (*ast* 클래스), 2166
interactive (*sys.flags*의 속성), 1988
interactive (대화형), **2309**
InteractiveConsole (*code* 클래스), 2113
InteractiveInterpreter (*code* 클래스), 2113
InterfaceError, 572
intern() (*sys* 모듈), 1996
internal_attr (*zipfile.ZipInfo*의 속성), 621
Internaldate2tuple() (*imaplib* 모듈), 1528
InternalError, 572
internalSubset (*xml.dom.DocumentType*의 속성), 1431
INTERNET_TIMEOUT (*test.support* 모듈), 1896
interpolated string literal, 64
interpolation
    bytearray (%), 81
    bytes (%), 81
Interpolation (*ast* 클래스), 2168
Interpolation (*string.templatelib* 클래스), 140
interpolation, string (%), 66
InterpolationDepthError, 666
InterpolationError, 666
InterpolationMissingOptionError, 666
interpolations (*string.templatelib.Template*의 속성), 138
InterpolationSyntaxError, 666
interpreted (인터프리티드), **2309**
Interpreter (*concurrent.interpreters* 클래스), 1075
interpreter shutdown (인터프리터 종료), **2309**
interpreter_requires_environment() (*test.support.script_helper* 모듈), 1906
InterpreterError, 1075
InterpreterNotFoundError, 1075
InterpreterPoolExecutor (*concurrent.futures* 클래스), 1067
interrupt() (*multiprocessing.Process* 메서드), 1022
interrupt() (*sqlite3.Connection* 메서드), 561
interrupt_main() (*_thread* 모듈), 1106
InterruptedError, 119
intersection() (*frozenset* 메서드), 92
intersection_update() (*frozenset* 메서드), 93
IntFlag (*enum* 클래스), 338
intro (*cmd.Cmd*의 속성), 990
InuseAttributeErr, 1434
inv() (*operator* 모듈), 452
inv_cdf() (*statistics.NormalDist* 메서드), 418
InvalidAccessErr, 1434
invalidate_caches() (*importlib* 모듈), 2127

invalidate_caches() (*importlib.abc.MetaPathFinder* 메서드), 2128
invalidate_caches() (*importlib.abc.PathEntryFinder* 메서드), 2129
invalidate_caches() (*importlib.machinery.FileFinder* 메서드), 2136
invalidate_caches() (*importlib.machinery.PathFinder* 의 클래스 메서드), 2136
invalidate_caches() (*platform* 모듈), 830
invalidate_caches() (*zipimport.zipimporter* 메서드), 2119
--invalidation-mode
compileall 명령줄 옵션, 2219
InvalidBase64CharactersDefect, 1326
InvalidBase64LengthDefect, 1326
InvalidBase64PaddingDefect, 1326
InvalidCharacterErr, 1434
InvalidDateDefect, 1326
InvalidFileException, 672
InvalidModificationErr, 1435
InvalidOperation (*decimal* 클래스), 384
InvalidStateErr, 1435
InvalidStateError, 1072, 1154
InvalidTZPathWarning, 261
InvalidURL, 1511
InvalidWriteError, 667
Invert (*ast* 클래스), 2171
invert() (*operator* 모듈), 452
io
module, 757
IO (*typing* 클래스), 1757
IO_REPARSE_TAG_APPEXECLINK (*stat* 모듈), 498
IO_REPARSE_TAG_MOUNT_POINT (*stat* 모듈), 498
IO_REPARSE_TAG_SYMLINK (*stat* 모듈), 498
IOBase (*io* 클래스), 760
ioctl() (*fcntl* 모듈), 2276
ioctl() (*socket.socket* 메서드), 1232
IOCTL_VM_SOCKETS_GET_LOCAL_CID (*socket* 모듈), 1220
IOError, 118
ior() (*operator* 모듈), 457
ios_ver() (*platform* 모듈), 831
io.StringIO
object, 54
ip (*ipaddress.IPv4Interface*의 속성), 1601
ip (*ipaddress.IPv6Interface*의 속성), 1602
ip_address() (*ipaddress* 모듈), 1589
ip_interface() (*ipaddress* 모듈), 1590
ip_network() (*ipaddress* 모듈), 1590
ipaddress
module, 1589
ipow() (*operator* 모듈), 458
ipv4_mapped (*ipaddress.IPv6Address*의 속성), 1594
IPv4Address (*ipaddress* 클래스), 1590
IPv4Interface (*ipaddress* 클래스), 1601
IPv4Network (*ipaddress* 클래스), 1595
IPV6_ENABLED (*test.support.socket_helper* 모듈), 1905
ipv6_mapped (*ipaddress.IPv4Address*의 속성), 1592
IPv6Address (*ipaddress* 클래스), 1593
IPv6Interface (*ipaddress* 클래스), 1601
IPv6Network (*ipaddress* 클래스), 1598
irshift() (*operator* 모듈), 458
is
operator, 38
Is (*ast* 클래스), 2172
is not
operator, 38
is_() (*operator* 모듈), 452
is_absolute() (*pathlib.PurePath* 메서드), 467
is_active() (*graphlib.TopologicalSorter* 메서드), 344
is_alive() (*multiprocessing.Process* 메서드), 1021
is_alive() (*threading.Thread* 메서드), 1003
is_android (*test.support* 모듈), 1896
is_annotated() (*symtable.Symbol* 메서드), 2203
is_anonymous (*ctypes.CField*의 속성), 878
is_apple (*test.support* 모듈), 1896
is_apple_mobile (*test.support* 모듈), 1896
is_assigned() (*symtable.Symbol* 메서드), 2204
is_async (*pyclbr.Function*의 속성), 2216
is_attachment() (*email.message.EmailMessage* 메서드), 1309
is_authenticated() (*urllib.request.HTTPPasswordMgrWithPriorAuth* 메서드), 1489
is_bitfield (*ctypes.CField*의 속성), 877
is_block_device() (*pathlib.Path* 메서드), 475
is_blocked() (*http.cookiejar.DefaultCookiePolicy* 메서드), 1572
is_canonical() (*decimal.Context* 메서드), 380
is_canonical() (*decimal.Decimal* 메서드), 372
is_char_device() (*pathlib.Path* 메서드), 475
IS_CHARACTER_JUNK() (*difflib* 모듈), 168
is_check_supported() (*lzma* 모듈), 610
is_closed() (*asyncio.loop* 메서드), 1158
is_closing() (*asyncio.BaseTransport* 메서드), 1186
is_closing() (*asyncio.StreamWriter* 메서드), 1137
is_comp_cell() (*symtable.Symbol* 메서드), 2204
is_comp_iter() (*symtable.Symbol* 메서드), 2204
is_dataclass() (*dataclasses* 모듈), 2039
is_declared_global() (*symtable.Symbol* 메서드), 2203
is_dir() (*importlib.abc.Traversable* 메서드), 2134
is_dir() (*importlib.resources.abc.Traversable* 메서드), 2149
is_dir() (*os.DirEntry* 메서드), 723
is_dir() (*pathlib.Path* 메서드), 475
is_dir() (*pathlib.types.PathInfo* 메서드), 486
is_dir() (*zipfile.Path* 메서드), 618
is_dir() (*zipfile.ZipInfo* 메서드), 620
is_emscripten (*test.support* 모듈), 1896
is_enabled() (*faulthandler* 모듈), 1926
is_expired() (*http.cookiejar.Cookie* 메서드), 1575
is_fifo() (*pathlib.Path* 메서드), 475
is_file() (*importlib.abc.Traversable* 메서드), 2134

is_file() (*importlib.resources.abc.Traversable* 메서드), 2149
is_file() (*os.DirEntry* 메서드), 723
is_file() (*pathlib.Path* 메서드), 474
is_file() (*pathlib.types.PathInfo* 메서드), 486
is_file() (*zipfile.Path* 메서드), 618
is_finalized() (*gc* 모듈), 2079
is_finalizing() (*sys* 모듈), 1997
is_finite() (*decimal.Context* 메서드), 380
is_finite() (*decimal.Decimal* 메서드), 372
is_free() (*symtable.Symbol* 메서드), 2203
is_free_class() (*symtable.Symbol* 메서드), 2204
is_global (*ipaddress.IPv4Address*의 속성), 1592
is_global (*ipaddress.IPv6Address*의 속성), 1593
is_global() (*symtable.Symbol* 메서드), 2203
is_hop_by_hop() (*wsgiref.util* 모듈), 1469
is_imported() (*symtable.Symbol* 메서드), 2203
is_infinite() (*decimal.Context* 메서드), 380
is_infinite() (*decimal.Decimal* 메서드), 372
is_integer() (*float* 메서드), 43
is_integer() (*fractions.Fraction* 메서드), 395
is_integer() (*int* 메서드), 43
is_junction() (*os.DirEntry* 메서드), 724
is_junction() (*pathlib.Path* 메서드), 475
is_jython (*test.support* 모듈), 1896
IS_LINE_JUNK() (*difflib* 모듈), 168
is_linetouched() (*curses.window* 메서드), 968
is_link_local (*ipaddress.IPv4Address* 의 속성), 1592
is_link_local (*ipaddress.IPv4Network* 의 속성), 1596
is_link_local (*ipaddress.IPv6Address* 의 속성), 1594
is_link_local (*ipaddress.IPv6Network* 의 속성), 1599
is_local() (*symtable.Symbol* 메서드), 2203
is_loopback (*ipaddress.IPv4Address*의 속성), 1592
is_loopback (*ipaddress.IPv4Network*의 속성), 1596
is_loopback (*ipaddress.IPv6Address*의 속성), 1594
is_loopback (*ipaddress.IPv6Network*의 속성), 1599
is_mount() (*pathlib.Path* 메서드), 475
is_multicast (*ipaddress.IPv4Address*의 속성), 1591
is_multicast (*ipaddress.IPv4Network*의 속성), 1596
is_multicast (*ipaddress.IPv6Address*의 속성), 1593
is_multicast (*ipaddress.IPv6Network*의 속성), 1599
is_multipart() (*email.message.EmailMessage* 메서드), 1305
is_multipart() (*email.message.Message* 메서드), 1342
is_namespace() (*symtable.Symbol* 메서드), 2204
is_nan() (*decimal.Context* 메서드), 380
is_nan() (*decimal.Decimal* 메서드), 372
is_nested() (*symtable.SymbolTable* 메서드), 2202
is_none() (*operator* 모듈), 452
is_nonlocal() (*symtable.Symbol* 메서드), 2203
is_normal() (*decimal.Context* 메서드), 380
is_normal() (*decimal.Decimal* 메서드), 372
is_normalized() (*unicodedata* 모듈), 182
is_not() (*operator* 모듈), 452
is_not_allowed() (*http.cookiejar.DefaultCookiePolicy* 메서드), 1573
is_not_none() (*operator* 모듈), 452
IS_OP (*opcode*), 2237
is_optimized() (*symtable.SymbolTable* 메서드), 2202
is_package() (*importlib.abc.InspectLoader* 메서드), 2130
is_package() (*importlib.abc.SourceLoader* 메서드), 2133
is_package() (*importlib.machinery.ExtensionFileLoader* 메서드), 2138
is_package() (*importlib.machinery.SourceFileLoader* 메서드), 2137
is_package() (*importlib.machinery.SourcelessFileLoader* 메서드), 2137
is_package() (*zipimport.zipimporter* 메서드), 2118
is_parameter() (*symtable.Symbol* 메서드), 2203
is_private (*ipaddress.IPv4Address*의 속성), 1591
is_private (*ipaddress.IPv4Network*의 속성), 1596
is_private (*ipaddress.IPv6Address*의 속성), 1593
is_private (*ipaddress.IPv6Network*의 속성), 1599
is_protocol() (*typing* 모듈), 1764
is_python_build() (*sysconfig* 모듈), 2018
is_qnan() (*decimal.Context* 메서드), 380
is_qnan() (*decimal.Decimal* 메서드), 372
is_reading() (*asyncio.ReadTransport* 메서드), 1187
is_referenced() (*symtable.Symbol* 메서드), 2203
is_relative_to() (*pathlib.PurePath* 메서드), 467
is_reserved (*ipaddress.IPv4Address*의 속성), 1592
is_reserved (*ipaddress.IPv4Network*의 속성), 1596
is_reserved (*ipaddress.IPv6Address*의 속성), 1594
is_reserved (*ipaddress.IPv6Network*의 속성), 1599
is_reserved() (*pathlib.PurePath* 메서드), 467
is_resource() (*importlib.abc.ResourceReader* 메서드), 2133
is_resource() (*importlib.resources* 모듈), 2148
is_resource() (*importlib.resources.abc.ResourceReader* 메서드), 2149
is_resource_enabled() (*test.support* 모듈), 1899
is_running() (*asyncio.loop* 메서드), 1158
is_running() (*concurrent.interpreters.Interpreter* 메서드), 1075
is_safe (*uuid.UUID*의 속성), 1543
is_serving() (*asyncio.Server* 메서드), 1177
is_set() (*asyncio.Event* 메서드), 1142
is_set() (*threading.Event* 메서드), 1010
is_signed() (*decimal.Context* 메서드), 380
is_signed() (*decimal.Decimal* 메서드), 372
is_site_local (*ipaddress.IPv6Address* 의 속성), 1594
is_site_local (*ipaddress.IPv6Network* 의 속성), 1600
is_skipped_line() (*bdb.Bdb* 메서드), 1922
is_snan() (*decimal.Context* 메서드), 380
is_snan() (*decimal.Decimal* 메서드), 372

is_socket() (*pathlib.Path* 메서드), 475
is_stack_trampoline_active() (*sys* 모듈), 2004
is_subnormal() (*decimal.Context* 메서드), 380
is_subnormal() (*decimal.Decimal* 메서드), 372
is_symlink() (*os.DirEntry* 메서드), 723
is_symlink() (*pathlib.Path* 메서드), 475
is_symlink() (*pathlib.types.PathInfo* 메서드), 486
is_symlink() (*zipfile.Path* 메서드), 618
is_tarfile() (*tarfile* 모듈), 625
is_term_resized() (*curses* 모듈), 960
is_tracing() (*tracemalloc* 모듈), 1960
is_tracked() (*gc* 모듈), 2079
is_type_parameter() (*symtable.Symbol* 메서드), 2203
is_typeddict() (*typing* 모듈), 1765
is_unspecified (*ipaddress.IPv4Address* 의 속성), 1592
is_unspecified (*ipaddress.IPv4Network* 의 속성), 1596
is_unspecified (*ipaddress.IPv6Address* 의 속성), 1594
is_unspecified (*ipaddress.IPv6Network* 의 속성), 1599
is_valid() (*string.Template* 메서드), 135
is_wasi (*test.support* 모듈), 1896
is_wintouched() (*curses.window* 메서드), 968
is_zero() (*decimal.Context* 메서드), 380
is_zero() (*decimal.Decimal* 메서드), 372
is_zipfile() (*zipfile* 모듈), 612
isabs() (*os.path* 모듈), 488
isabstract() (*inspect* 모듈), 2086
IsADirectoryError, 119
isalnum() (*bytearray* 메서드), 77
isalnum() (*bytes* 메서드), 77
isalnum() (*curses.ascii* 모듈), 986
isalnum() (*str* 메서드), 57
isalpha() (*bytearray* 메서드), 78
isalpha() (*bytes* 메서드), 78
isalpha() (*curses.ascii* 모듈), 986
isalpha() (*str* 메서드), 57
isascii() (*bytearray* 메서드), 78
isascii() (*bytes* 메서드), 78
isascii() (*curses.ascii* 모듈), 986
isascii() (*str* 메서드), 58
isasyncgen() (*inspect* 모듈), 2085
isasyncgenfunction() (*inspect* 모듈), 2085
isatty() (*io.IOBase* 메서드), 761
isatty() (*os* 모듈), 702
isawaitable() (*inspect* 모듈), 2085
isblank() (*curses.ascii* 모듈), 986
isblk() (*tarfile.TarInfo* 메서드), 633
isbuiltin() (*inspect* 모듈), 2085
ischr() (*tarfile.TarInfo* 메서드), 633
isclass() (*inspect* 모듈), 2084
isclose() (*cmath* 모듈), 363
isclose() (*math* 모듈), 354
iscntrl() (*curses.ascii* 모듈), 986
iscode() (*inspect* 모듈), 2085
iscoroutine() (*asyncio* 모듈), 1128
iscoroutine() (*inspect* 모듈), 2084
iscoroutinefunction() (*inspect* 모듈), 2084
isctrl() (*curses.ascii* 모듈), 986
isDaemon() (*threading.Thread* 메서드), 1004
isdatadescriptor() (*inspect* 모듈), 2086
isdecimal() (*str* 메서드), 58
isdev() (*tarfile.TarInfo* 메서드), 633
isdevdrive() (*os.path* 모듈), 489
isdigit() (*bytearray* 메서드), 78
isdigit() (*bytes* 메서드), 78
isdigit() (*curses.ascii* 모듈), 986
isdigit() (*str* 메서드), 58
isdir() (*os.path* 모듈), 488
isdir() (*tarfile.TarInfo* 메서드), 633
isdisjoint() (*frozenset* 메서드), 92
isdown() (*turtle* 모듈), 1693
iselement() (*xml.etree.ElementTree* 모듈), 1416
isenabled() (*gc* 모듈), 2077
isEnabledFor() (*logging.Logger* 메서드), 786
isendwin() (*curses* 모듈), 960
ISEOF() (*token* 모듈), 2205
isfifo() (*tarfile.TarInfo* 메서드), 633
isfile() (*os.path* 모듈), 488
isfile() (*tarfile.TarInfo* 메서드), 632
isfinite() (*cmath* 모듈), 362
isfinite() (*math* 모듈), 354
isfirstline() (*fileinput* 모듈), 955
isframe() (*inspect* 모듈), 2085
isfunction() (*inspect* 모듈), 2084
isfuture() (*asyncio* 모듈), 1181
isgenerator() (*inspect* 모듈), 2084
isgeneratorfunction() (*inspect* 모듈), 2084
isgetsetdescriptor() (*inspect* 모듈), 2086
isgraph() (*curses.ascii* 모듈), 986
isidentifier() (*str* 메서드), 58
isinf() (*cmath* 모듈), 362
isinf() (*math* 모듈), 354
isinstance()
built-in function, 20
isjunction() (*os.path* 모듈), 489
iskeyword() (*keyword* 모듈), 2210
isleap() (*calendar* 모듈), 265
islice() (*itertools* 모듈), 430
islink() (*os.path* 모듈), 489
islnk() (*tarfile.TarInfo* 메서드), 633
islower() (*bytearray* 메서드), 78
islower() (*bytes* 메서드), 78
islower() (*curses.ascii* 모듈), 986
islower() (*str* 메서드), 58
ismemberdescriptor() (*inspect* 모듈), 2086
ismeta() (*curses.ascii* 모듈), 986
ismethod() (*inspect* 모듈), 2084
ismethoddescriptor() (*inspect* 모듈), 2086
ismethodwrapper() (*inspect* 모듈), 2085
ismodule() (*inspect* 모듈), 2084
ismount() (*os.path* 모듈), 489
isnan() (*cmath* 모듈), 362

isnan() (*math* 모듈), 354
ISNONTERMINAL() (*token* 모듈), 2205
IsNot (*ast* 클래스), 2172
isnumeric() (*str* 메서드), 58
isocalendar() (*datetime.date* 메서드), 226
isocalendar() (*datetime.datetime* 메서드), 236
isoformat() (*datetime.date* 메서드), 227
isoformat() (*datetime.datetime* 메서드), 236
isoformat() (*datetime.time* 메서드), 242
isolated (*sys.flags*의 속성), 1988
IsolatedAsyncioTestCase (*unittest* 클래스), 1819
isolation_level (*sqlite3.Connection*의 속성), 567
isoweekday() (*datetime.date* 메서드), 226
isoweekday() (*datetime.datetime* 메서드), 236
ispackage() (*inspect* 모듈), 2084
isprint() (*curses.ascii* 모듈), 986
isprintable() (*str* 메서드), 58
ispunct() (*curses.ascii* 모듈), 986
isqrt() (*math* 모듈), 352
isreadable() (*pprint* 모듈), 320
isreadable() (*pprint.PrettyPrinter* 메서드), 321
isrecursive() (*pprint* 모듈), 320
isrecursive() (*pprint.PrettyPrinter* 메서드), 321
isreg() (*tarfile.TarInfo* 메서드), 633
isreserved() (*os.path* 모듈), 489
isReservedKey() (*http.cookies.Morsel* 메서드), 1565
isroutine() (*inspect* 모듈), 2085
isSameNode() (*xml.dom.Node* 메서드), 1430
issoftkeyword() (*keyword* 모듈), 2210
isspace() (*bytearray* 메서드), 78
isspace() (*bytes* 메서드), 78
isspace() (*curses.ascii* 모듈), 986
isspace() (*str* 메서드), 58
isstdin() (*fileinput* 모듈), 955
issubclass()
    built-in function, 20
issubset() (*frozenset* 메서드), 92
issuperset() (*frozenset* 메서드), 92
issym() (*tarfile.TarInfo* 메서드), 633
ISTERMINAL() (*token* 모듈), 2205
istitle() (*bytearray* 메서드), 78
istitle() (*bytes* 메서드), 78
istitle() (*str* 메서드), 59
istraceback() (*inspect* 모듈), 2085
isub() (*operator* 모듈), 458
isupper() (*bytearray* 메서드), 79
isupper() (*bytes* 메서드), 79
isupper() (*curses.ascii* 모듈), 986
isupper() (*str* 메서드), 59
isvisible() (*turtle* 모듈), 1696
isxdigit() (*curses.ascii* 모듈), 986
ITALIC (*tkinter.font* 모듈), 1640
item() (*tkinter.ttk.Treeview* 메서드), 1660
item() (*xml.dom.NamedNodeMap* 메서드), 1433
item() (*xml.dom.NodeList* 메서드), 1430
itemgetter() (*operator* 모듈), 455
items() (*configparser.ConfigParser* 메서드), 664
items() (*contextvars.Context* 메서드), 1104
items() (*dict* 메서드), 96
items() (*email.message.EmailMessage* 메서드), 1306
items() (*email.message.Message* 메서드), 1345
items() (*mailbox.Mailbox* 메서드), 1372
items() (*types.MappingProxyType* 메서드), 316
items() (*xml.etree.ElementTree.Element* 메서드), 1419
itemsize (*array.array*의 속성), 302
itemsize (*memoryview*의 속성), 90
ItemsView (*collections.abc* 클래스), 291
ItemsView (*typing* 클래스), 1769
iter()
    built-in function, 20
iter() (*xml.etree.ElementTree.Element* 메서드), 1420
iter() (*xml.etree.ElementTree.ElementTree* 메서드), 1422
iter_attachments() (*email.message.EmailMessage* 메서드), 1310
iter_child_nodes() (*ast* 모듈), 2197
iter_fields() (*ast* 모듈), 2197
iter_importers() (*pkgutil* 모듈), 2120
iter_modules() (*pkgutil* 모듈), 2120
iter_parts() (*email.message.EmailMessage* 메서드), 1310
iter_unpack() (*struct* 모듈), 192
iter_unpack() (*struct.Struct* 메서드), 198
Iterable (*collections.abc* 클래스), 290
Iterable (*typing* 클래스), 1771
iterable (이터러블), **2309**
Iterator (*collections.abc* 클래스), 290
Iterator (*typing* 클래스), 1771
iterator (이터레이터), **2310**
iterator protocol, 46
iterdecode() (*codecs* 모듈), 201
iterdir() (*importlib.abc.Traversable* 메서드), 2134
iterdir() (*importlib.resources.abc.Traversable* 메서드), 2149
iterdir() (*pathlib.Path* 메서드), 477
iterdir() (*zipfile.Path* 메서드), 618
iterdump() (*sqlite3.Connection* 메서드), 563
iterencode() (*codecs* 모듈), 201
iterencode() (*json.JSONEncoder* 메서드), 1367
iterfind() (*xml.etree.ElementTree.Element* 메서드), 1420
iterfind() (*xml.etree.ElementTree.ElementTree* 메서드), 1422
iteritems() (*mailbox.Mailbox* 메서드), 1372
iterkeys() (*mailbox.Mailbox* 메서드), 1372
itermonthdates() (*calendar.Calendar* 메서드), 261
itermonthdays() (*calendar.Calendar* 메서드), 261
itermonthdays2() (*calendar.Calendar* 메서드), 262
itermonthdays3() (*calendar.Calendar* 메서드), 262
itermonthdays4() (*calendar.Calendar* 메서드), 262
iterparse() (*xml.etree.ElementTree* 모듈), 1416
itertext() (*xml.etree.ElementTree.Element* 메서드), 1420
itertools
    module, 423
itervalues() (*mailbox.Mailbox* 메서드), 1372

iterweekdays() (*calendar.Calendar* 메서드), 261
ITIMER_PROF (*signal* 모듈), 1291
ITIMER_REAL (*signal* 모듈), 1291
ITIMER_VIRTUAL (*signal* 모듈), 1291
ItimerError, 1291
itruediv() (*operator* 모듈), 458
ixor() (*operator* 모듈), 458

J

-j
    compileall 명령줄 옵션, 2219
JANUARY (*calendar* 모듈), 266
java_ver() (*platform* 모듈), 830
job_size (*compression.zstd.CompressionParameter* 의 속성), 591
join() (*asyncio.Queue* 메서드), 1151
join() (*bytearray* 메서드), 73
join() (*bytes* 메서드), 73
join() (*multiprocessing.JoinableQueue* 메서드), 1026
join() (*multiprocessing.pool.Pool* 메서드), 1043
join() (*multiprocessing.Process* 메서드), 1021
join() (*os.path* 모듈), 489
join() (*queue.Queue* 메서드), 1100
join() (*shlex* 모듈), 2263
join() (*str* 메서드), 59
join() (*threading.Thread* 메서드), 1002
join_thread() (*multiprocessing.Queue* 메서드), 1025
join_thread() (*test.support.threading_helper* 모듈), 1907
JoinableQueue (*multiprocessing* 클래스), 1026
JoinedStr (*ast* 클래스), 2167
joinpath() (*importlib.abc.Traversable* 메서드), 2134
joinpath() (*importlib.resources.abc.Traversable* 메서드), 2150
joinpath() (*pathlib.PurePath* 메서드), 468
joinpath() (*zipfile.Path* 메서드), 619
js_output() (*http.cookies.BaseCookie* 메서드), 1564
js_output() (*http.cookies.Morsel* 메서드), 1565
json
    module, 1360
json 명령줄 옵션
    --compact, 1370
    -h, 1370
    --help, 1370
    --indent, 1370
    infile, 1370
    --json-lines, 1370
    --no-ensure-ascii, 1370
    --no-indent, 1370
    outfile, 1370
    --sort-keys, 1370
    --tab, 1370
JSONDecodeError, 1367
JSONDecoder (*json* 클래스), 1365
JSONEncoder (*json* 클래스), 1366
--json-lines
    json 명령줄 옵션, 1370
json.tool
    module, 1369
JULY (*calendar* 모듈), 266
JUMP (*monitoring event*), 2009
JUMP (*opcode*), 2244
jump (*pdb command*), 1935
JUMP_BACKWARD (*opcode*), 2237
JUMP_BACKWARD_NO_INTERRUPT (*opcode*), 2237
JUMP_FORWARD (*opcode*), 2237
JUMP_IF_FALSE (*opcode*), 2244
JUMP_IF_TRUE (*opcode*), 2244
JUMP_NO_INTERRUPT (*opcode*), 2244
JUNE (*calendar* 모듈), 266

K

-k
    unittest 명령줄 옵션, 1803
kbhit() (*msvcrt* 모듈), 2248
kde() (*statistics* 모듈), 410
kde_random() (*statistics* 모듈), 410
KEEP (*enum.FlagBoundary* 의 속성), 340
kevent() (*select* 모듈), 1278
key (*http.cookies.Morsel* 의 속성), 1565
key (*zoneinfo.ZoneInfo* 의 속성), 259
key function (키 함수), **2310**
KEY_A1 (*curses* 모듈), 974
KEY_A3 (*curses* 모듈), 975
KEY_ALL_ACCESS (*winreg* 모듈), 2256
KEY_B2 (*curses* 모듈), 975
KEY_BACKSPACE (*curses* 모듈), 973
KEY_BEG (*curses* 모듈), 975
KEY_BREAK (*curses* 모듈), 973
KEY_BTAB (*curses* 모듈), 975
KEY_C1 (*curses* 모듈), 975
KEY_C3 (*curses* 모듈), 975
KEY_CANCEL (*curses* 모듈), 975
KEY_CATAB (*curses* 모듈), 974
KEY_CLEAR (*curses* 모듈), 974
KEY_CLOSE (*curses* 모듈), 975
KEY_COMMAND (*curses* 모듈), 975
KEY_COPY (*curses* 모듈), 975
KEY_CREATE (*curses* 모듈), 975
KEY_CREATE_LINK (*winreg* 모듈), 2256
KEY_CREATE_SUB_KEY (*winreg* 모듈), 2256
KEY_CTAB (*curses* 모듈), 974
KEY_DC (*curses* 모듈), 973
KEY_DL (*curses* 모듈), 973
KEY_DOWN (*curses* 모듈), 973
KEY_EIC (*curses* 모듈), 974
KEY_END (*curses* 모듈), 975
KEY_ENTER (*curses* 모듈), 974
KEY_ENUMERATE_SUB_KEYS (*winreg* 모듈), 2256
KEY_EOL (*curses* 모듈), 974
KEY_EOS (*curses* 모듈), 974
KEY_EXECUTE (*winreg* 모듈), 2256
KEY_EXIT (*curses* 모듈), 975
KEY_F0 (*curses* 모듈), 973
KEY_FIND (*curses* 모듈), 975
KEY_Fn (*curses* 모듈), 973

KEY_HELP (*curses* 모듈), 975
KEY_HOME (*curses* 모듈), 973
KEY_IC (*curses* 모듈), 974
KEY_IL (*curses* 모듈), 973
KEY_LEFT (*curses* 모듈), 973
KEY_LL (*curses* 모듈), 974
KEY_MARK (*curses* 모듈), 975
KEY_MAX (*curses* 모듈), 978
KEY_MESSAGE (*curses* 모듈), 975
KEY_MIN (*curses* 모듈), 973
KEY_MOUSE (*curses* 모듈), 978
KEY_MOVE (*curses* 모듈), 975
KEY_NEXT (*curses* 모듈), 976
KEY_NOTIFY (*winreg* 모듈), 2256
KEY_NPAGE (*curses* 모듈), 974
KEY_OPEN (*curses* 모듈), 976
KEY_OPTIONS (*curses* 모듈), 976
KEY_PPAGE (*curses* 모듈), 974
KEY_PREVIOUS (*curses* 모듈), 976
KEY_PRINT (*curses* 모듈), 974
KEY_QUERY_VALUE (*winreg* 모듈), 2256
KEY_READ (*winreg* 모듈), 2256
KEY_REDO (*curses* 모듈), 976
KEY_REFERENCE (*curses* 모듈), 976
KEY_REFRESH (*curses* 모듈), 976
KEY_REPLACE (*curses* 모듈), 976
KEY_RESET (*curses* 모듈), 974
KEY_RESIZE (*curses* 모듈), 978
KEY_RESTART (*curses* 모듈), 976
KEY_RESUME (*curses* 모듈), 976
KEY_RIGHT (*curses* 모듈), 973
KEY_SAVE (*curses* 모듈), 976
KEY_SBEG (*curses* 모듈), 976
KEY_SCANCEL (*curses* 모듈), 976
KEY_SCOMMAND (*curses* 모듈), 976
KEY_SCOPY (*curses* 모듈), 976
KEY_SCREATE (*curses* 모듈), 976
KEY_SDC (*curses* 모듈), 976
KEY_SDL (*curses* 모듈), 976
KEY_SELECT (*curses* 모듈), 977
KEY_SEND (*curses* 모듈), 977
KEY_SEOL (*curses* 모듈), 977
KEY_SET_VALUE (*winreg* 모듈), 2256
KEY_SEXIT (*curses* 모듈), 977
KEY_SF (*curses* 모듈), 974
KEY_SFIND (*curses* 모듈), 977
KEY_SHELP (*curses* 모듈), 977
KEY_SHOME (*curses* 모듈), 977
KEY_SIC (*curses* 모듈), 977
KEY_SLEFT (*curses* 모듈), 977
KEY_SMESSAGE (*curses* 모듈), 977
KEY_SMOVE (*curses* 모듈), 977
KEY_SNEXT (*curses* 모듈), 977
KEY_SOPTIONS (*curses* 모듈), 977
KEY_SPREVIOUS (*curses* 모듈), 977
KEY_SPRINT (*curses* 모듈), 977
KEY_SR (*curses* 모듈), 974
KEY_SREDO (*curses* 모듈), 977
KEY_SREPLACE (*curses* 모듈), 977
KEY_SRESET (*curses* 모듈), 974
KEY_SRIGHT (*curses* 모듈), 977
KEY_SRSUME (*curses* 모듈), 978
KEY_SSAVE (*curses* 모듈), 978
KEY_SSUSPEND (*curses* 모듈), 978
KEY_STAB (*curses* 모듈), 974
KEY_SUNDO (*curses* 모듈), 978
KEY_SUSPEND (*curses* 모듈), 978
KEY_UNDO (*curses* 모듈), 978
KEY_UP (*curses* 모듈), 973
KEY_WOW64_32KEY (*winreg* 모듈), 2257
KEY_WOW64_64KEY (*winreg* 모듈), 2257
KEY_WRITE (*winreg* 모듈), 2256
KeyboardInterrupt, 114
KeyError, 114
keylog_filename (*ssl.SSLContext*의 속성), 1264
keyname() (*curses* 모듈), 960
keypad() (*curses.window* 메서드), 968
keyrefs() (*weakref.WeakKeyDictionary* 메서드), 306
keys() (*contextvars.Context* 메서드), 1104
keys() (*dict* 메서드), 96
keys() (*email.message.EmailMessage* 메서드), 1306
keys() (*email.message.Message* 메서드), 1344
keys() (*mailbox.Mailbox* 메서드), 1372
keys() (*sqlite3.Row* 메서드), 570
keys() (*types.MappingProxyType* 메서드), 316
keys() (*xml.etree.ElementTree.Element* 메서드), 1419
KeysView (*collections.abc* 클래스), 291
KeysView (*typing* 클래스), 1769
keyword
    module, 2210
keyword (*ast* 클래스), 2172
keyword argument (키워드 인자), **2310**
keywords (*functools.partial*의 속성), 451
kill() (*asyncio.subprocess.Process* 메서드), 1149
kill() (*asyncio.SubprocessTransport* 메서드), 1188
kill() (*multiprocessing.Process* 메서드), 1022
kill() (*os* 모듈), 742
kill() (*subprocess.Popen* 메서드), 1087
kill_python() (*test.support.script_helper* 모듈), 1907
kill_workers() (*concurrent.futures.ProcessPoolExecutor* 메서드), 1069
killchar() (*curses* 모듈), 961
killpg() (*os* 모듈), 742
kind (*inspect.Parameter*의 속성), 2090
knownfiles (*mimetypes* 모듈), 1391
kqueue() (*select* 모듈), 1278
KqueueSelector (*selectors* 클래스), 1287
KW_ONLY (*dataclasses* 모듈), 2039
kwargs (*inspect.BoundArguments*의 속성), 2091
kwargs (*typing.ParamSpec*의 속성), 1747
kwlist (*keyword* 모듈), 2210

L

-L
    calendar 명령줄 옵션, 268

-l
    calendar 명령줄 옵션, 268
    compileall 명령줄 옵션, 2219
    mimetypes 명령줄 옵션, 1393
    pickletools 명령줄 옵션, 2246
    tarfile 명령줄 옵션, 637
    trace 명령줄 옵션, 1952
    zipfile 명령줄 옵션, 622
L (*re* 모듈), 149
L2CAP_LM (*socket* 모듈), 1220
Lambda (*ast* 클래스), 2192
lambda (람다), **2310**
LambdaType (*types* 모듈), 313
LANG, 1609, 1610, 1617, 1621
LANGUAGE, 1609, 1610
language
    C, 38
LARGEST (*test.support* 모듈), 1898
LargeZipFile, 612
last_accepted (*multiprocessing.connection.Listener* 의 속성), 1045
last_exc (*sys* 모듈), 1997
last_mode (*compression.zstd.ZstdCompressor*의 속성), 586
last_traceback (*sys* 모듈), 1998
last_type (*sys* 모듈), 1998
last_value (*sys* 모듈), 1998
lastChild (*xml.dom.Node*의 속성), 1429
lastcmd (*cmd.Cmd*의 속성), 990
lastgroup (*re.Match*의 속성), 159
lastindex (*re.Match*의 속성), 159
lastResort (*logging* 모듈), 801
lastrowid (*sqlite3.Cursor*의 속성), 570
layout() (*tkinter.ttk.Style* 메서드), 1663
lazy (*compression.zstd.Strategy*의 속성), 592
lazy2 (*compression.zstd.Strategy*의 속성), 592
lazycache() (*linecache* 모듈), 511
LazyLoader (*importlib.util* 클래스), 2142
LBRACE (*token* 모듈), 2208
LBYL, **2310**
LC_ALL, 1609, 1610
LC_ALL (*locale* 모듈), 1623
LC_COLLATE (*locale* 모듈), 1623
LC_CTYPE (*locale* 모듈), 1623
LC_MESSAGES, 1609, 1610
LC_MESSAGES (*locale* 모듈), 1623
LC_MONETARY (*locale* 모듈), 1623
LC_NUMERIC (*locale* 모듈), 1623
LC_TIME (*locale* 모듈), 1623
lchflags() (*os* 모듈), 715
lchmod() (*os* 모듈), 716
lchmod() (*pathlib.Path* 메서드), 483
lchown() (*os* 모듈), 716
lcm() (*math* 모듈), 352
ldexp() (*math* 모듈), 354
ldm_bucket_size_log (*compression.zstd.CompressionParameter* 의 속성), 591
ldm_hash_log (*compression.zstd.CompressionParameter* 의 속성), 590
ldm_hash_rate_log (*compression.zstd.CompressionParameter* 의 속성), 591
ldm_min_match (*compression.zstd.CompressionParameter* 의 속성), 590
le() (*operator* 모듈), 451
leapdays() (*calendar* 모듈), 265
leaveok() (*curses.window* 메서드), 968
left (*filecmp.dircmp*의 속성), 500
left() (*turtle* 모듈), 1685
left_list (*filecmp.dircmp*의 속성), 500
left_only (*filecmp.dircmp*의 속성), 500
LEFTSHIFT (*token* 모듈), 2208
LEFTSHIFTEQUAL (*token* 모듈), 2209
LEGACY_TRANSACTION_CONTROL (*sqlite3* 모듈), 555
len
    built-in function, 47, 93
len()
    built-in function, 21
length (*xml.dom.NamedNodeMap*의 속성), 1433
length (*xml.dom.NodeList*의 속성), 1430
length_hint() (*operator* 모듈), 454
--lenient
    mimetypes 명령줄 옵션, 1393
LESS (*token* 모듈), 2207
LESSEQUAL (*token* 모듈), 2208
level (*logging.Logger*의 속성), 785
lexical analyzer (어휘 분석기), **2310**
LexicalHandler (*xml.sax.handler* 클래스), 1445
lexists() (*os.path* 모듈), 487
LF (*curses.ascii* 모듈), 984
lgamma() (*math* 모듈), 358
libc_ver() (*platform* 모듈), 831
LIBRARIES_ASSEMBLY_NAME_PREFIX (*msvcrt* 모듈), 2249
library (*dbm.ndbm* 모듈), 549
library (*ssl.SSLError*의 속성), 1244
LibraryLoader (*ctypes* 클래스), 863
license (내장 변수), 36
LifoQueue (*asyncio* 클래스), 1152
LifoQueue (*queue* 클래스), 1098
limit_denominator() (*fractions.Fraction* 메서드), 395
LimitOverrunError, 1154
line (*bdb.Breakpoint*의 속성), 1920
LINE (*monitoring event*), 2010
line (*traceback.FrameSummary*의 속성), 2070
line_buffering (*io.TextIOWrapper*의 속성), 768
line_num (*csv.csvreader*의 속성), 647
linear_regression() (*statistics* 모듈), 416
line-buffered I/O, 25
linecache
    module, 510
lineno (*ast.AST*의 속성), 2165

lineno (*doctest.DocTest*의 속성), 1792
lineno (*doctest.Example*의 속성), 1793
lineno (*inspect.FrameInfo*의 속성), 2094
lineno (*inspect.Traceback*의 속성), 2095
lineno (*json.JSONDecodeError*의 속성), 1368
lineno (*netrc.NetrcParseError*의 속성), 669
lineno (*pyclbr.Class*의 속성), 2216
lineno (*pyclbr.Function*의 속성), 2216
lineno (*re.PatternError*의 속성), 154
lineno (*shlex.shlex*의 속성), 2266
lineno (*SyntaxError*의 속성), 116
lineno (*tomllib.TOMLDecodeError*의 속성), 668
lineno (*traceback.FrameSummary*의 속성), 2070
lineno (*traceback.TracebackException*의 속성), 2068
lineno (*tracemalloc.Filter*의 속성), 1961
lineno (*tracemalloc.Frame*의 속성), 1961
lineno (*xml.parsers.expat.ExpatError*의 속성), 1460
lineno() (*fileinput* 모듈), 955
LINES, 959, 964
--lines
    calendar 명령줄 옵션, 268
LINES (*curses* 모듈), 971
lines (*os.terminal_size*의 속성), 711
linesep (*email.policy.Policy*의 속성), 1320
linesep (*os* 모듈), 755
lineterminator (*csv.Dialect*의 속성), 646
LineTooLong, 1512
link() (*os* 모듈), 716
linked_to_musl() (*test.support* 모듈), 1902
LinkFallbackError, 626
linkname (*tarfile.TarInfo*의 속성), 631
LinkOutsideDestinationError, 626
list
    object, 49, 50
    type, operations on, 49
--list
    tarfile 명령줄 옵션, 637
    zipfile 명령줄 옵션, 622
List (*ast* 클래스), 2168
list (*pdb command*), 1935
List (*typing* 클래스), 1767
list (내장 클래스), 50
list (리스트), **2310**
list comprehension (리스트 컴프리헨션), **2310**
list() (*imaplib.IMAP4* 메서드), 1531
list() (*multiprocessing.managers.SyncManager* 메서드), 1037
list() (*poplib.POP3* 메서드), 1525
list() (*tarfile.TarFile* 메서드), 628
list_all() (*concurrent.interpreters* 모듈), 1074
LIST_APPEND (*opcode*), 2231
list_dialects() (*csv* 모듈), 643
LIST_EXTEND (*opcode*), 2235
list_folders() (*mailbox.Maildir* 메서드), 1374
list_folders() (*mailbox.MH* 메서드), 1378
ListComp (*ast* 클래스), 2174
listdir() (*os* 모듈), 716
listdrives() (*os* 모듈), 717
listen() (*logging.config* 모듈), 803
listen() (*socket.socket* 메서드), 1232
listen() (*turtle* 모듈), 1704
listener (*logging.handlers.QueueHandler*의 속성), 827
Listener (*multiprocessing.connection* 클래스), 1045
--listfuncs
    trace 명령줄 옵션, 1952
listMethods() (*xmlrpc.client.ServerProxy.system* 메서드), 1578
listmounts() (*os* 모듈), 717
listvolumes() (*os* 모듈), 717
listxattr() (*os* 모듈), 737
Literal (*typing* 모듈), 1734
literal_eval() (*ast* 모듈), 2196
literals
    binary, 39
    complex number, 39
    floating-point, 39
    hexadecimal, 39
    integer, 39
    numeric, 39
    octal, 39
LiteralString (*typing* 모듈), 1730
LittleEndianStructure (*ctypes* 클래스), 875
LittleEndianUnion (*ctypes* 클래스), 874
ljust() (*bytearray* 메서드), 74
ljust() (*bytes* 메서드), 74
ljust() (*str* 메서드), 59
LK_LOCK (*msvcrt* 모듈), 2247
LK_NBLCK (*msvcrt* 모듈), 2247
LK_NBRLCK (*msvcrt* 모듈), 2247
LK_RLCK (*msvcrt* 모듈), 2247
LK_UNLCK (*msvcrt* 모듈), 2247
ll (*pdb command*), 1935
LMTP (*smtplib* 클래스), 1535
ln() (*decimal.Context* 메서드), 380
ln() (*decimal.Decimal* 메서드), 372
LNKTYPE (*tarfile* 모듈), 626
Load (*ast* 클래스), 2169
load() (*http.cookiejar.FileCookieJar* 메서드), 1570
load() (*http.cookies.BaseCookie* 메서드), 1564
load() (*json* 모듈), 1364
load() (*marshal* 모듈), 544
load() (*pickle* 모듈), 526
load() (*pickle.Unpickler* 메서드), 528
load() (*plistlib* 모듈), 670
load() (*tomllib* 모듈), 667
load() (*tracemalloc.Snapshot*의 클래스 메서드), 1962
LOAD_ATTR (*opcode*), 2236
LOAD_BUILD_CLASS (*opcode*), 2233
load_cert_chain() (*ssl.SSLContext* 메서드), 1259
LOAD_CLOSURE (*opcode*), 2244
LOAD_COMMON_CONSTANT (*opcode*), 2233
LOAD_CONST (*opcode*), 2234
LOAD_CONST_IMMORTAL (*opcode*), 2244
load_default_certs() (*ssl.SSLContext* 메서드), 1260

LOAD_DEREF (*opcode*), 2239
load_dh_params() (*ssl.SSLContext* 메서드), 1262
load_extension() (*sqlite3.Connection* 메서드), 563
LOAD_FAST (*opcode*), 2238
LOAD_FAST_AND_CLEAR (*opcode*), 2238
LOAD_FAST_BORROW (*opcode*), 2238
LOAD_FAST_BORROW_LOAD_FAST_BORROW (*opcode*), 2238
LOAD_FAST_CHECK (*opcode*), 2238
LOAD_FAST_LOAD_FAST (*opcode*), 2238
LOAD_FROM_DICT_OR_DEREF (*opcode*), 2239
LOAD_FROM_DICT_OR_GLOBALS (*opcode*), 2234
LOAD_GLOBAL (*opcode*), 2238
LOAD_LOCALS (*opcode*), 2234
load_module() (*importlib.abc.FileLoader* 메서드), 2131
load_module() (*importlib.abc.InspectLoader* 메서드), 2131
load_module() (*importlib.abc.Loader* 메서드), 2129
load_module() (*importlib.abc.SourceLoader* 메서드), 2132
load_module() (*importlib.machinery.SourceFileLoader* 메서드), 2137
load_module() (*importlib.machinery.SourcelessFileLoader* 메서드), 2137
load_module() (*zipimport.zipimporter* 메서드), 2118
LOAD_NAME (*opcode*), 2234
load_package_tests() (*test.support* 모듈), 1903
LOAD_SMALL_INT (*opcode*), 2234
LOAD_SPECIAL (*opcode*), 2243
LOAD_SUPER_ATTR (*opcode*), 2236
load_verify_locations() (*ssl.SSLContext* 메서드), 1260
Loader (*importlib.abc* 클래스), 2129
loader (*importlib.machinery.ModuleSpec* 의 속성), 2139
loader (로더), **2311**
loader_state (*importlib.machinery.ModuleSpec*의 속성), 2139
LoadError, 1567
LoadFileDialog (*tkinter.filedialog* 클래스), 1644
LoadKey() (*winreg* 모듈), 2252
LoadLibrary() (*ctypes.LibraryLoader* 메서드), 863
loads() (*json* 모듈), 1365
loads() (*marshal* 모듈), 544
loads() (*pickle* 모듈), 526
loads() (*plistlib* 모듈), 671
loads() (*tomllib* 모듈), 668
loads() (*xmlrpc.client* 모듈), 1582
loadTestsFromModule() (*unittest.TestLoader* 메서드), 1822
loadTestsFromName() (*unittest.TestLoader* 메서드), 1822
loadTestsFromNames() (*unittest.TestLoader* 메서드), 1823
loadTestsFromTestCase() (*unittest.TestLoader* 메서드), 1822
local (*threading* 클래스), 1001
LOCAL_CREDS (*socket* 모듈), 1221
LOCAL_CREDS_PERSISTENT (*socket* 모듈), 1221
localcontext() (*decimal* 모듈), 376
locale
    module, 1617
--locale
    calendar 명령줄 옵션, 268
LOCALE (*re* 모듈), 149
locale encoding (로케일 인코딩), **2311**
localeconv() (*locale* 모듈), 1617
LocaleHTMLCalendar (*calendar* 클래스), 264
LocaleTextCalendar (*calendar* 클래스), 264
localize() (*locale* 모듈), 1622
localName (*xml.dom.Attr*의 속성), 1433
localName (*xml.dom.Node*의 속성), 1429
--locals
    unittest 명령줄 옵션, 1803
locals()
    built-in function, 21
localtime() (*email.utils* 모듈), 1357
localtime() (*time* 모듈), 774
Locator (*xml.sax.xmlreader* 클래스), 1451
Lock (*asyncio* 클래스), 1141
Lock (*multiprocessing* 클래스), 1030
lock (*sys.thread_info*의 속성), 2006
Lock (*threading* 클래스), 1004
lock() (*mailbox.Babyl* 메서드), 1379
lock() (*mailbox.Mailbox* 메서드), 1373
lock() (*mailbox.Maildir* 메서드), 1376
lock() (*mailbox.mbox* 메서드), 1377
lock() (*mailbox.MH* 메서드), 1378
lock() (*mailbox.MMDF* 메서드), 1380
Lock() (*multiprocessing.managers.SyncManager* 메서드), 1037
LOCK_EX (*fcntl* 모듈), 2277
LOCK_NB (*fcntl* 모듈), 2277
LOCK_SH (*fcntl* 모듈), 2277
LOCK_UN (*fcntl* 모듈), 2277
locked() (*_thread.lock* 메서드), 1107
locked() (*asyncio.Condition* 메서드), 1143
locked() (*asyncio.Lock* 메서드), 1141
locked() (*asyncio.Semaphore* 메서드), 1144
locked() (*multiprocessing.Lock* 메서드), 1031
locked() (*multiprocessing.RLock* 메서드), 1031
locked() (*threading.Condition* 메서드), 1007
locked() (*threading.Lock* 메서드), 1005
locked() (*threading.RLock* 메서드), 1006
lockf() (*fcntl* 모듈), 2277
lockf() (*os* 모듈), 702
locking() (*msvcrt* 모듈), 2247
LockType (*_thread* 모듈), 1105
log() (*cmath* 모듈), 361
log() (*logging* 모듈), 798
log() (*logging.Logger* 메서드), 788
log() (*math* 모듈), 356
log1p() (*math* 모듈), 356

log2() (*math* 모듈), 356
log10() (*cmath* 모듈), 361
log10() (*decimal.Context* 메서드), 380
log10() (*decimal.Decimal* 메서드), 372
log10() (*math* 모듈), 356
LOG_ALERT (*syslog* 모듈), 2283
LOG_AUTH (*syslog* 모듈), 2284
LOG_AUTHPRIV (*syslog* 모듈), 2284
LOG_CONS (*syslog* 모듈), 2284
LOG_CRIT (*syslog* 모듈), 2283
LOG_CRON (*syslog* 모듈), 2284
LOG_DAEMON (*syslog* 모듈), 2284
log_date_time_string() (*http.server.BaseHTTPRequestHandler* 메서드), 1560
LOG_DEBUG (*syslog* 모듈), 2283
LOG_EMERG (*syslog* 모듈), 2283
LOG_ERR (*syslog* 모듈), 2283
log_error() (*http.server.BaseHTTPRequestHandler* 메서드), 1559
log_exception() (*wsgiref.handlers.BaseHandler* 메서드), 1475
LOG_FTP (*syslog* 모듈), 2284
LOG_INFO (*syslog* 모듈), 2283
LOG_INSTALL (*syslog* 모듈), 2284
LOG_KERN (*syslog* 모듈), 2284
LOG_LAUNCHD (*syslog* 모듈), 2284
LOG_LOCAL0 (*syslog* 모듈), 2284
LOG_LOCAL1 (*syslog* 모듈), 2284
LOG_LOCAL2 (*syslog* 모듈), 2284
LOG_LOCAL3 (*syslog* 모듈), 2284
LOG_LOCAL4 (*syslog* 모듈), 2284
LOG_LOCAL5 (*syslog* 모듈), 2284
LOG_LOCAL6 (*syslog* 모듈), 2284
LOG_LOCAL7 (*syslog* 모듈), 2284
LOG_LPR (*syslog* 모듈), 2284
LOG_MAIL (*syslog* 모듈), 2284
log_message() (*http.server.BaseHTTPRequestHandler* 메서드), 1559
LOG_NDELAY (*syslog* 모듈), 2284
LOG_NETINFO (*syslog* 모듈), 2284
LOG_NEWS (*syslog* 모듈), 2284
LOG_NOTICE (*syslog* 모듈), 2283
LOG_NOWAIT (*syslog* 모듈), 2284
LOG_ODELAY (*syslog* 모듈), 2284
LOG_PERROR (*syslog* 모듈), 2284
LOG_PID (*syslog* 모듈), 2284
LOG_RAS (*syslog* 모듈), 2284
LOG_REMOTEAUTH (*syslog* 모듈), 2284
log_request() (*http.server.BaseHTTPRequestHandler* 메서드), 1559
LOG_SYSLOG (*syslog* 모듈), 2284
log_to_stderr() (*multiprocessing* 모듈), 1048
LOG_USER (*syslog* 모듈), 2284
LOG_UUCP (*syslog* 모듈), 2284
LOG_WARNING (*syslog* 모듈), 2283
logb() (*decimal.Context* 메서드), 380
logb() (*decimal.Decimal* 메서드), 372
Logger (*logging* 클래스), 785
LoggerAdapter (*logging* 클래스), 796
logging
    module, 784
    예러, 784
logging.config
    module, 802
logging.handlers
    module, 814
logical_and() (*decimal.Context* 메서드), 380
logical_and() (*decimal.Decimal* 메서드), 373
logical_invert() (*decimal.Context* 메서드), 380
logical_invert() (*decimal.Decimal* 메서드), 373
logical_or() (*decimal.Context* 메서드), 381
logical_or() (*decimal.Decimal* 메서드), 373
logical_xor() (*decimal.Context* 메서드), 381
logical_xor() (*decimal.Decimal* 메서드), 373
login() (*ftplib.FTP* 메서드), 1519
login() (*imaplib.IMAP4* 메서드), 1531
login() (*smtplib.SMTP* 메서드), 1538
login_cram_md5() (*imaplib.IMAP4* 메서드), 1531
login_tty() (*os* 모듈), 702
LOGNAME, 694, 954
lognormvariate() (*random* 모듈), 401
logout() (*imaplib.IMAP4* 메서드), 1531
LogRecord (*logging* 클래스), 794
LONG_TIMEOUT (*test.support* 모듈), 1897
longMessage (*unittest.TestCase*의 속성), 1818
longname() (*curses* 모듈), 961
lookup() (*codecs* 모듈), 199
lookup() (*symtable.SymbolTable* 메서드), 2202
lookup() (*tkinter.ttk.Style* 메서드), 1662
lookup() (*unicodedata* 모듈), 181
lookup_error() (*codecs* 모듈), 203
LookupError, 113
loop
    over mutable sequence, 47
loop_factory (*unittest.IsolatedAsyncioTestCase*의 속성), 1819
LOOPBACK_TIMEOUT (*test.support* 모듈), 1896
lower() (*bytearray* 메서드), 79
lower() (*bytes* 메서드), 79
lower() (*str* 메서드), 59
LPAR (*token* 모듈), 2207
lpAttributeList (*subprocess.STARTUPINFO*의 속성), 1088
lru_cache() (*functools* 모듈), 442
lseek() (*os* 모듈), 703
LShift (*ast* 클래스), 2171
lshift() (*operator* 모듈), 452
LSQB (*token* 모듈), 2207
lstat() (*os* 모듈), 717
lstat() (*pathlib.Path* 메서드), 474
lstrip() (*bytearray* 메서드), 74
lstrip() (*bytes* 메서드), 74
lstrip() (*str* 메서드), 59
lsub() (*imaplib.IMAP4* 메서드), 1531
Lt (*ast* 클래스), 2172

lt() (*operator* 모듈), 451
lt() (*turtle* 모듈), 1685
LtE (*ast* 클래스), 2172
LWPCookieJar (*http.cookiejar* 클래스), 1571
lzma
    module, 606
LZMACompressor (*lzma* 클래스), 607
LZMADecompressor (*lzma* 클래스), 608
LZMAError, 606
LZMAFile (*lzma* 클래스), 606

M

-m
    ast 명령줄 옵션, 2200
    calendar 명령줄 옵션, 269
    cProfile 명령줄 옵션, 1940
    pdb 명령줄 옵션, 1929
    pickletools 명령줄 옵션, 2246
    trace 명령줄 옵션, 1953
    zipapp 명령줄 옵션, 1977
M (*re* 모듈), 150
mac_ver() (*platform* 모듈), 831
machine() (*platform* 모듈), 829
macros (*netrc.netrc*의 속성), 670
MADV_AUTOSYNC (*mmap* 모듈), 1301
MADV_CORE (*mmap* 모듈), 1301
MADV_DODUMP (*mmap* 모듈), 1301
MADV_DOFORK (*mmap* 모듈), 1301
MADV_DONTDUMP (*mmap* 모듈), 1301
MADV_DONTFORK (*mmap* 모듈), 1301
MADV_DONTNEED (*mmap* 모듈), 1301
MADV_FREE (*mmap* 모듈), 1301
MADV_FREE_REUSABLE (*mmap* 모듈), 1301
MADV_FREE_REUSE (*mmap* 모듈), 1301
MADV_HUGEPAGE (*mmap* 모듈), 1301
MADV_HWPOISON (*mmap* 모듈), 1301
MADV_MERGEABLE (*mmap* 모듈), 1301
MADV_NOCORE (*mmap* 모듈), 1301
MADV_NOHUGEPAGE (*mmap* 모듈), 1301
MADV_NORMAL (*mmap* 모듈), 1301
MADV_NOSYNC (*mmap* 모듈), 1301
MADV_PROTECT (*mmap* 모듈), 1301
MADV_RANDOM (*mmap* 모듈), 1301
MADV_REMOVE (*mmap* 모듈), 1301
MADV_SEQUENTIAL (*mmap* 모듈), 1301
MADV_SOFT_OFFLINE (*mmap* 모듈), 1301
MADV_UNMERGEABLE (*mmap* 모듈), 1301
MADV_WILLNEED (*mmap* 모듈), 1301
madvise() (*mmap.mmap* 메서드), 1300
magic
    method (메서드), 2311
magic method (매직 메서드), **2311**
MAGIC_NUMBER (*importlib.util* 모듈), 2140
MagicMock (*unittest.mock* 클래스), 1862
mailbox
    module, 1370
Mailbox (*mailbox* 클래스), 1371
mailcap
    module, 2295
Maildir (*mailbox* 클래스), 1374
MaildirMessage (*mailbox* 클래스), 1381
--main
    zipapp 명령줄 옵션, 1977
main() (*site* 모듈), 2111
main() (*unittest* 모듈), 1827
main_thread() (*threading* 모듈), 998
mainloop() (*turtle* 모듈), 1706
maintype (*email.headerregistry.ContentTypeHeader*의 속성), 1329
major (*email.headerregistry.MIMEVersionHeader*의 속성), 1329
major() (*os* 모듈), 719
make_alternative() (*email.message.EmailMessage* 메서드), 1311
make_archive() (*shutil* 모듈), 518
make_bad_fd() (*test.support.os_helper* 모듈), 1909
MAKE_CELL (*opcode*), 2239
make_cookies() (*http.cookiejar.CookieJar* 메서드), 1569
make_dataclass() (*dataclasses* 모듈), 2038
make_file() (*difflib.HtmlDiff* 메서드), 165
MAKE_FUNCTION (*opcode*), 2240
make_header() (*email.header* 모듈), 1354
make_legacy_pyc() (*test.support.import_helper* 모듈), 1911
make_mixed() (*email.message.EmailMessage* 메서드), 1311
make_msgid() (*email.utils* 모듈), 1357
make_parser() (*xml.sax* 모듈), 1443
make_pkg() (*test.support.script_helper* 모듈), 1907
make_related() (*email.message.EmailMessage* 메서드), 1311
make_script() (*test.support.script_helper* 모듈), 1907
make_server() (*wsgiref.simple_server* 모듈), 1471
make_table() (*difflib.HtmlDiff* 메서드), 166
make_zip_pkg() (*test.support.script_helper* 모듈), 1907
make_zip_script() (*test.support.script_helper* 모듈), 1907
makedev() (*os* 모듈), 719
makedirs() (*os* 모듈), 718
makeelement() (*xml.etree.ElementTree.Element* 메서드), 1420
makefile() (*socket.socket* 메서드), 1232
makeLogRecord() (*logging* 모듈), 799
makePickle() (*logging.handlers.SocketHandler* 메서드), 820
makeRecord() (*logging.Logger* 메서드), 789
makeSocket() (*logging.handlers.DatagramHandler* 메서드), 821
makeSocket() (*logging.handlers.SocketHandler* 메서드), 819
maketrans() (*bytearray*의 정적 메서드), 73
maketrans() (*bytes*의 정적 메서드), 73
maketrans() (*str*의 정적 메서드), 59
MalformedHeaderDefect, 1326

manager (*logging.LoggerAdapter*의 속성), 796
mangle_from_ (*email.policy.Compat32*의 속성), 1324
mangle_from_ (*email.policy.Policy*의 속성), 1320
MANPAGER, 1773
mant_dig (*sys.float_info*의 속성), 1990
map()
    built-in function, 21
map() (*concurrent.futures.Executor* 메서드), 1064
map() (*multiprocessing.pool.Pool* 메서드), 1042
map() (*tkinter.ttk.Style* 메서드), 1662
MAP_32BIT (*mmap* 모듈), 1302
MAP_ADD (*opcode*), 2232
MAP_ALIGNED_SUPER (*mmap* 모듈), 1302
MAP_ANON (*mmap* 모듈), 1302
MAP_ANONYMOUS (*mmap* 모듈), 1302
map_async() (*multiprocessing.pool.Pool* 메서드), 1042
MAP_CONCEAL (*mmap* 모듈), 1302
MAP_DENYWRITE (*mmap* 모듈), 1302
MAP_EXECUTABLE (*mmap* 모듈), 1302
MAP_HASSEMAPHORE (*mmap* 모듈), 1302
MAP_JIT (*mmap* 모듈), 1302
MAP_NOCACHE (*mmap* 모듈), 1302
MAP_NOEXTEND (*mmap* 모듈), 1302
MAP_NORESERVE (*mmap* 모듈), 1302
MAP_POPULATE (*mmap* 모듈), 1302
MAP_PRIVATE (*mmap* 모듈), 1302
MAP_RESILIENT_CODESIGN (*mmap* 모듈), 1302
MAP_RESILIENT_MEDIA (*mmap* 모듈), 1302
MAP_SHARED (*mmap* 모듈), 1302
MAP_STACK (*mmap* 모듈), 1302
map_table_b2() (*stringprep* 모듈), 183
map_table_b3() (*stringprep* 모듈), 183
map_to_type() (*email.headerregistry.HeaderRegistry* 메서드), 1330
MAP_TPRO (*mmap* 모듈), 1302
MAP_TRANSLATED_ALLOW_EXECUTE (*mmap* 모듈), 1302
MAP_UNIX03 (*mmap* 모듈), 1302
mapLogRecord() (*logging.handlers.HTTPHandler* 메서드), 825
mapping
    object, 93
    types, operations on, 93
Mapping (*collections.abc* 클래스), 291
Mapping (*typing* 클래스), 1769
mapping (매핑), **2311**
MappingProxyType (*types* 클래스), 315
MappingView (*collections.abc* 클래스), 291
MappingView (*typing* 클래스), 1769
mapPriority() (*logging.handlers.SysLogHandler* 메서드), 823
maps (*collections.ChainMap*의 속성), 269
MARCH (*calendar* 모듈), 266
markcoroutinefunction() (*inspect* 모듈), 2084
marshal
    module, 543
masking
    operations, 40
master (*tkinter.Tk*의 속성), 1629
Match (*ast* 클래스), 2184
Match (*re* 클래스), 156
Match (*typing* 클래스), 1768
match() (*pathlib.PurePath* 메서드), 468
match() (*re* 모듈), 151
match() (*re.Pattern* 메서드), 155
match_case (*ast* 클래스), 2184
MATCH_CLASS (*opcode*), 2242
MATCH_KEYS (*opcode*), 2233
MATCH_MAPPING (*opcode*), 2233
MATCH_SEQUENCE (*opcode*), 2233
match_value() (*test.support.Matcher* 메서드), 1905
MatchAs (*ast* 클래스), 2188
MatchClass (*ast* 클래스), 2187
Matcher (*test.support* 클래스), 1905
matches() (*test.support.Matcher* 메서드), 1905
MatchMapping (*ast* 클래스), 2186
MatchOr (*ast* 클래스), 2189
MatchSequence (*ast* 클래스), 2185
MatchSingleton (*ast* 클래스), 2185
MatchStar (*ast* 클래스), 2186
MatchValue (*ast* 클래스), 2184
math
    module, 39, 350
matmul() (*operator* 모듈), 453
MatMult (*ast* 클래스), 2171
max
    built-in function, 47
max (*datetime.datetime*의 속성), 232
max (*datetime.date*의 속성), 225
max (*datetime.timedelta*의 속성), 220
max (*datetime.time*의 속성), 240
max (*sys.float_info*의 속성), 1990
MAX (*uuid* 모듈), 1545
max()
    built-in function, 22
max() (*decimal.Context* 메서드), 381
max() (*decimal.Decimal* 메서드), 373
max_10_exp (*sys.float_info*의 속성), 1990
max_children (*socketserver.ThreadingMixIn*의 속성), 1549
max_count (*email.headerregistry.BaseHeader*의 속성), 1327
MAX_EMAX (*decimal* 모듈), 383
max_exp (*sys.float_info*의 속성), 1990
MAX_INTERPOLATION_DEPTH (*configparser* 모듈), 665
max_line_length (*email.policy.Policy*의 속성), 1320
max_lines (*textwrap.TextWrapper*의 속성), 180
max_mag() (*decimal.Context* 메서드), 381
max_mag() (*decimal.Decimal* 메서드), 373
max_memuse (*test.support* 모듈), 1897
MAX_PREC (*decimal* 모듈), 383
max_prefixlen (*ipaddress.IPv4Address* 의 속성), 1591
max_prefixlen (*ipaddress.IPv4Network* 의 속성), 1596

max_prefixlen (*ipaddress.IPv6Address* 의 속성), 1593
max_prefixlen (*ipaddress.IPv6Network* 의 속성), 1599
MAX_Py_ssize_t (*test.support* 모듈), 1897
maxarray (*reprlib.Repr* 의 속성), 325
maxdeque (*reprlib.Repr* 의 속성), 325
maxdict (*reprlib.Repr* 의 속성), 325
maxDiff (*unittest.TestCase* 의 속성), 1818
maxfrozenset (*reprlib.Repr* 의 속성), 325
MAXIMUM_SUPPORTED (*ssl.TLSVersion* 의 속성), 1254
maximum_version (*ssl.SSLContext* 의 속성), 1265
maxlen (*collections.deque* 의 속성), 276
maxlevel (*reprlib.Repr* 의 속성), 325
maxlist (*reprlib.Repr* 의 속성), 325
maxlong (*reprlib.Repr* 의 속성), 326
maxother (*reprlib.Repr* 의 속성), 326
maxset (*reprlib.Repr* 의 속성), 325
maxsize (*asyncio.Queue* 의 속성), 1151
maxsize (*sys* 모듈), 1998
maxstring (*reprlib.Repr* 의 속성), 326
maxtuple (*reprlib.Repr* 의 속성), 325
maxunicode (*sys* 모듈), 1998
MAXYEAR (*datetime* 모듈), 218
MAY (*calendar* 모듈), 266
MB_ICONASTERISK (*winsound* 모듈), 2260
MB_ICONERROR (*winsound* 모듈), 2260
MB_ICONEXCLAMATION (*winsound* 모듈), 2260
MB_ICONHAND (*winsound* 모듈), 2260
MB_ICONINFORMATION (*winsound* 모듈), 2260
MB_ICONQUESTION (*winsound* 모듈), 2260
MB_ICONSTOP (*winsound* 모듈), 2261
MB_ICONWARNING (*winsound* 모듈), 2261
MB_OK (*winsound* 모듈), 2260
mbox (*mailbox* 클래스), 1376
mboxMessage (*mailbox* 클래스), 1382
md5() (*hashlib* 모듈), 674
mean (*statistics.NormalDist* 의 속성), 417
mean() (*statistics* 모듈), 408
measure() (*tkinter.font.Font* 메서드), 1641
median (*statistics.NormalDist* 의 속성), 417
median() (*statistics* 모듈), 411
median_grouped() (*statistics* 모듈), 412
median_high() (*statistics* 모듈), 411
median_low() (*statistics* 모듈), 411
member() (*enum* 모듈), 342
member_names (*enum.EnumDict* 의 속성), 340
MemberDescriptorType (*types* 모듈), 315
memfd_create() (*os* 모듈), 733
memmove() (*ctypes* 모듈), 869
--memo
    pickletools 명령줄 옵션, 2246
MemoryBIO (*ssl* 클래스), 1274
MemoryError, 114
MemoryHandler (*logging.handlers* 클래스), 825
memoryview
    object, 67
memoryview (내장 클래스), 84
memoryview_at() (*ctypes* 모듈), 870
memset() (*ctypes* 모듈), 869
merge() (*heapq* 모듈), 294
message (*BaseExceptionGroup* 의 속성), 121
Message (*email.message* 클래스), 1341
Message (*mailbox* 클래스), 1380
Message (*tkinter.messagebox* 클래스), 1644
message_factory (*email.policy.Policy* 의 속성), 1320
message_from_binary_file() (*email* 모듈), 1314
message_from_bytes() (*email* 모듈), 1314
message_from_file() (*email* 모듈), 1314
message_from_string() (*email* 모듈), 1314
MessageBeep() (*winsound* 모듈), 2259
MessageClass (*http.server.BaseHTTPRequestHandler* 의 속성), 1558
MessageDefect, 1325
MessageError, 1325
MessageParseError, 1325
messages (*xml.parsers.expat.errors* 모듈), 1462
meta path finder (메타 경로 파인더), **2311**
meta() (*curses* 모듈), 961
meta_path (*sys* 모듈), 1998
metaclass (메타 클래스), **2311**
metadata() (*importlib.metadata* 모듈), 2153
--metadata-encoding
    zipfile 명령줄 옵션, 622
MetaPathFinder (*importlib.abc* 클래스), 2128
metavar (*optparse.Option* 의 속성), 941
MetavarTypeHelpFormatter (*argparse* 클래스), 885
method
    object, 106
method (*urllib.request.Request* 의 속성), 1484
method (메서드), **2311**
    magic, 2311
    special, 2316
method resolution order (메서드 결정 순서), **2311**
method_calls (*unittest.mock.Mock* 의 속성), 1841
methodcaller() (*operator* 모듈), 455
MethodDescriptorType (*types* 모듈), 314
methodHelp() (*xmlrpc.client.ServerProxy.system* 메서드), 1578
methods
    bytearray, 70
    bytes, 70
    string, 54
methods (*pyclbr.Class* 의 속성), 2216
methodSignature() (*xmlrpc.client.ServerProxy.system* 메서드), 1578
MethodType (*types* 모듈), 314
MethodWrapperType (*types* 모듈), 314
metrics() (*tkinter.font.Font* 메서드), 1641
MFD_ALLOW_SEALING (*os* 모듈), 733
MFD_CLOEXEC (*os* 모듈), 733
MFD_HUGE_1GB (*os* 모듈), 733
MFD_HUGE_1MB (*os* 모듈), 733

MFD_HUGE_2GB (*os* 모듈), 733
MFD_HUGE_2MB (*os* 모듈), 733
MFD_HUGE_8MB (*os* 모듈), 733
MFD_HUGE_16GB (*os* 모듈), 733
MFD_HUGE_16MB (*os* 모듈), 733
MFD_HUGE_32MB (*os* 모듈), 733
MFD_HUGE_64KB (*os* 모듈), 733
MFD_HUGE_256MB (*os* 모듈), 733
MFD_HUGE_512KB (*os* 모듈), 733
MFD_HUGE_512MB (*os* 모듈), 733
MFD_HUGE_MASK (*os* 모듈), 733
MFD_HUGE_SHIFT (*os* 모듈), 733
MFD_HUGETLB (*os* 모듈), 733
MH (*mailbox* 클래스), 1377
MHMessage (*mailbox* 클래스), 1384
microsecond (*datetime.datetime*의 속성), 233
microsecond (*datetime.time*의 속성), 241
microseconds (*datetime.timedelta*의 속성), 221
MIME
    base64 인코딩, 1394
    quoted-printable 인코딩, 1400
    콘텐츠 유형, 1389
    헤더, 1390
MIMEApplication (*email.mime.application* 클래스), 1350
MIMEAudio (*email.mime.audio* 클래스), 1350
MIMEBase (*email.mime.base* 클래스), 1349
MIMEImage (*email.mime.image* 클래스), 1350
MIMEMessage (*email.mime.message* 클래스), 1351
MIMEMultipart (*email.mime.multipart* 클래스), 1349
MIMENonMultipart (*email.mime.nonmultipart* 클래스), 1349
MIMEPart (*email.message* 클래스), 1312
MIMEText (*email.mime.text* 클래스), 1351
mimetypes
    module, 1389
MimeTypes (*mimetypes* 클래스), 1391
mimetypes 명령줄 옵션
    -e, 1393
    --extension, 1393
    -h, 1393
    --help, 1393
    -l, 1393
    --lenient, 1393
MIMEVersionHeader (*email.headerregistry* 클래스), 1329
min
    built-in function, 47
min (*datetime.datetime*의 속성), 232
min (*datetime.date*의 속성), 225
min (*datetime.timedelta*의 속성), 220
min (*datetime.time*의 속성), 240
min (*sys.float_info*의 속성), 1990
min()
    built-in function, 22
min() (*decimal.Context* 메서드), 381
min() (*decimal.Decimal* 메서드), 373
min_10_exp (*sys.float_info*의 속성), 1990
MIN_EMIN (*decimal* 모듈), 383
MIN_ETINY (*decimal* 모듈), 383
min_exp (*sys.float_info*의 속성), 1990
min_mag() (*decimal.Context* 메서드), 381
min_mag() (*decimal.Decimal* 메서드), 373
min_match (*compression.zstd.CompressionParameter*의 속성), 590
MINEQUAL (*token* 모듈), 2208
MINIMUM_SUPPORTED (*ssl.TLSVersion*의 속성), 1254
minimum_version (*ssl.SSLContext*의 속성), 1265
minor (*email.headerregistry.MIMEVersionHeader*의 속성), 1329
minor() (*os* 모듈), 719
MINUS (*token* 모듈), 2207
minus() (*decimal.Context* 메서드), 381
minute (*datetime.datetime*의 속성), 233
minute (*datetime.time*의 속성), 241
MINYEAR (*datetime* 모듈), 218
mirrored() (*unicodedata* 모듈), 182
misc_header (*cmd.Cmd*의 속성), 990
MisplacedEnvelopeHeaderDefect, 1326
--missing
    trace 명령줄 옵션, 1953
MISSING (*contextvars.Token*의 속성), 1102
MISSING (*dataclasses* 모듈), 2039
MISSING (*sys.monitoring* 모듈), 2012
MISSING_C_DOCSTRINGS (*test.support* 모듈), 1898
missing_compiler_executable() (*test.support* 모듈), 1904
MissingHeaderBodySeparatorDefect, 1326
MissingSectionHeaderError, 667
mkd() (*ftplib.FTP* 메서드), 1521
mkdir() (*os* 모듈), 718
mkdir() (*pathlib.Path* 메서드), 480
mkdir() (*zipfile.ZipFile* 메서드), 617
mkdtemp() (*tempfile* 모듈), 504
mkfifo() (*os* 모듈), 718
mknod() (*os* 모듈), 719
mkstemp() (*tempfile* 모듈), 503
mktemp() (*tempfile* 모듈), 506
mktime() (*time* 모듈), 774
mktime_tz() (*email.utils* 모듈), 1358
mlsd() (*ftplib.FTP* 메서드), 1521
mmap
    module, 1297
mmap (*mmap* 클래스), 1298
MMDF (*mailbox* 클래스), 1380
MMDFMessage (*mailbox* 클래스), 1386
Mock (*unittest.mock* 클래스), 1834
mock_add_spec() (*unittest.mock.Mock* 메서드), 1837
mock_calls (*unittest.mock.Mock*의 속성), 1841
mock_open() (*unittest.mock* 모듈), 1867
Mod (*ast* 클래스), 2171
mod() (*operator* 모듈), 453
--mode
    ast 명령줄 옵션, 2200
mode (*bz2.BZ2File*의 속성), 603
mode (*compression.zstd.ZstdFile*의 속성), 585

mode (*gzip.GzipFile*의 속성), 599
mode (*io.FileIO*의 속성), 764
mode (*lzma.LZMAFile*의 속성), 607
mode (*statistics.NormalDist*의 속성), 417
mode (*tarfile.TarInfo*의 속성), 631
mode() (*statistics* 모듈), 412
mode() (*turtle* 모듈), 1707
modes
    file, 23
modf() (*math* 모듈), 353
modified() (*urllib.robotparser.RobotFileParser* 메서드), 1506
modify() (*select.devpoll* 메서드), 1279
modify() (*select.epoll* 메서드), 1281
modify() (*selectors.BaseSelector* 메서드), 1286
modify() (*select.poll* 메서드), 1281
module
    __future__, 2075
    __main__, 2020
    _thread, 1105
    _tkinter, 1630
    abc, 2058
    aifc, 2293
    annotationlib, 2100
    argparse, 881
    array, 67, 301
    ast, 2161
    asynchat, 2293
    asyncio, 1109
    asyncore, 2293
    atexit, 2063
    audioop, 2294
    base64, 1394
    bdb, 1919
    binascii, 1398
    bisect, 297
    builtins, 33, 2019
    bz2, 601
    calendar, 261
    cgi, 2294
    cgitb, 2294
    chunk, 2294
    cmath, 360
    cmd, 988
    code, 2113
    codecs, 198
    codeop, 2115
    collections, 269
    collections.abc, 287
    colorsys, 1608
    compileall, 2218
    compression.zstd, 583
    concurrent.futures, 1063
    concurrent.interpreters, 1072
    configparser, 649
    contextlib, 2044
    contextvars, 1101
    copy, 317
    copyreg, 540
    cProfile, 1941
    crypt, 2294
    csv, 641
    ctypes, 841
    curses, 957
    curses.ascii, 984
    curses.panel, 987
    curses.textpad, 982
    dataclasses, 2033
    datetime, 217
    dbm, 545
    dbm.dumb, 550
    dbm.gnu, 547
    dbm.ndbm, 549
    dbm.sqlite3, 547
    decimal, 364
    difflib, 164
    dis, 2222
    distutils, 2294
    doctest, 1777
    email, 1303
    email.charset, 1354
    email.contentmanager, 1332
    email.encoders, 1356
    email.errors, 1325
    email.generator, 1315
    email.header, 1351
    email.headerregistry, 1326
    email.iterators, 1359
    email.message, 1304
    email.mime, 1349
    email.mime.application, 1350
    email.mime.audio, 1350
    email.mime.base, 1349
    email.mime.image, 1350
    email.mime.message, 1351
    email.mime.multipart, 1349
    email.mime.nonmultipart, 1349
    email.mime.text, 1351
    email.parser, 1312
    email.policy, 1318
    email.utils, 1357
    encodings, 214
    encodings.idna, 215
    encodings.mbcs, 216
    encodings.utf_8_sig, 216
    ensurepip, 1965
    enum, 327
    errno, 833
    faulthandler, 1925
    fcntl, 2275
    filecmp, 499
    fileinput, 954
    fnmatch, 509
    fractions, 393
    ftplib, 1517
    functools, 440

gc, 2077
getopt, 2289
getpass, 953
gettext, 1609
glob, 506
graphlib, 343
grp, 2270
gzip, 598
hashlib, 673
heapq, 293
hmac, 685
html, 1401
html.entities, 1406
html.parser, 1402
http, 1506
http.client, 1510
http.cookiejar, 1567
http.cookies, 1563
http.server, 1556
idlelib, 1677
imaplib, 1527
imghdr, 2295
imp, 2295
importlib, 2125
importlib.abc, 2128
importlib.machinery, 2135
importlib.metadata, 2150
importlib.resources, 2145
importlib.resources.abc, 2148
importlib.util, 2140
inspect, 2081
io, 757
ipaddress, 1589
itertools, 423
json, 1360
json.tool, 1369
keyword, 2210
linecache, 510
locale, 1617
logging, 784
logging.config, 802
logging.handlers, 814
lzma, 606
mailbox, 1370
mailcap, 2295
marshal, 543
math, 39, 350
mimetypes, 1389
mmap, 1297
modulefinder, 2122
msilib, 2295
msvcrt, 2247
multiprocessing, 1012
multiprocessing.connection, 1044
multiprocessing.dummy, 1048
multiprocessing.managers, 1035
multiprocessing.pool, 1041
multiprocessing.shared_memory, 1057
multiprocessing.sharedctypes, 1033
netrc, 669
nis, 2295
nntplib, 2295
numbers, 347
operator, 451
optparse, 926
os, 689
os.path, 486
ossaudiodev, 2296
pathlib, 459
pathlib.types, 485
pdb, 1928
pickle, 523
pickletools, 2245
pipes, 2296
pkgutil, 2119
platform, 828
plistlib, 670
poplib, 1524
posix, 2268
pprint, 319
profile, 1941
pstats, 1943
pty, 705, 2273
pwd, 2269
py_compile, 2217
pyclbr, 2215
pydoc, 1773
queue, 1098
quopri, 1400
random, 397
re, 55, 142
readline, 184
reprlib, 324
resource, 2278
rlcompleter, 189
runpy, 2123
sched, 1096
secrets, 686
select, 1277
selectors, 1284
shelve, 540
shlex, 2263
shutil, 511
signal, 1288
site, 2108
sitecustomize, 2110
smtpd, 2296
smtplib, 1534
sndhdr, 2296
socket, 1212
socketserver, 1547
spwd, 2296
sqlite3, 551
ssl, 1241
stat, 493, 724
statistics, 406

string, 125
stringprep, 183
string.templatelib, 136
struct, 191, 1236
subprocess, 1077
sunau, 2296
symtable, 2200
sys, 25, 1981
sysconfig, 2013
syslog, 2282
sys.monitoring, 2008
tabnanny, 2214
tarfile, 623
telnetlib, 2297
tempfile, 501
termios, 2271
test, 1893
test.regrtest, 1895
test.support, 1896
test.support.bytecode_helper, 1907
test.support.import_helper, 1910
test.support.os_helper, 1908
test.support.script_helper, 1906
test.support.socket_helper, 1905
test.support.threading_helper, 1907
test.support.warnings_helper, 1912
textwrap, 177
threading, 995
time, 771
timeit, 1947
tkinter, 1627
tkinter.colorchooser, 1640
tkinter.commondialog, 1644
tkinter.dnd, 1647
tkinter.filedialog, 1642
tkinter.font, 1640
tkinter.messagebox, 1644
tkinter.scrolledtext, 1647
tkinter.simpledialog, 1641
tkinter.ttk, 1648
token, 2205
tokenize, 2210
tomllib, 667
trace, 1952
traceback, 2065
tracemalloc, 1954
tty, 2272
turtle, 1678
turtledemo, 1714
types, 106, 311
typing, 1717
unicodedata, 180
unittest, 1800
unittest.mock, 1831
urllib, 1478
urllib.error, 1504
urllib.parse, 1496
urllib.request, 1478
urllib.response, 1495
urllib.robotparser, 1505
usercustomize, 2110
uu, 2297
uuid, 1541
venv, 1967
warnings, 2025
wave, 1605
weakref, 304
webbrowser, 1465
winreg, 2250
winsound, 2258
wsgiref, 1468
wsgiref.handlers, 1473
wsgiref.headers, 1470
wsgiref.simple_server, 1471
wsgiref.types, 1476
wsgiref.util, 1468
wsgiref.validate, 1472
xdrlib, 2297
xml, 1407
xml.dom, 1426
xml.dom.minidom, 1436
xml.dom.pulldom, 1440
xml.etree.ElementInclude, 1418
xml.etree.ElementTree, 1408
xml.parsers.expat, 1454
xml.parsers.expat.errors, 1461
xml.parsers.expat.model, 1461
xmlrpc, 1575
xmlrpc.client, 1576
xmlrpc.server, 1583
xml.sax, 1442
xml.sax.handler, 1444
xml.sax.saxutils, 1449
xml.sax.xmlreader, 1450
zipapp, 1976
zipfile, 611
zipimport, 2117
zlib, 594
zoneinfo, 256
Module (*ast* 클래스), 2166
module (*pyclbr.Class*의 속성), 2216
module (*pyclbr.Function*의 속성), 2215
MODULE (*symtable.SymbolTableType*의 속성), 2201
module (모듈), **2311**
module_spec (모듈 스펙), **2311**
module_from_spec() (*importlib.util* 모듈), 2141
modulefinder
module, 2122
ModuleFinder (*modulefinder* 클래스), 2122
ModuleInfo (*pkgutil* 클래스), 2119
ModuleNotFoundError, 114
modules (*modulefinder.ModuleFinder*의 속성), 2122
modules (*sys* 모듈), 1999
modules_cleanup() (*test.support.import_helper* 모듈), 1911

modules_setup() (*test.support.import_helper* 모듈), 1911
ModuleSpec (*importlib.machinery* 클래스), 2138
ModuleType (*types* 클래스), 314
modulus (*sys.hash_info*의 속성), 1994
MON_1 (*locale* 모듈), 1619
MON_2 (*locale* 모듈), 1619
MON_3 (*locale* 모듈), 1619
MON_4 (*locale* 모듈), 1619
MON_5 (*locale* 모듈), 1619
MON_6 (*locale* 모듈), 1619
MON_7 (*locale* 모듈), 1619
MON_8 (*locale* 모듈), 1619
MON_9 (*locale* 모듈), 1619
MON_10 (*locale* 모듈), 1619
MON_11 (*locale* 모듈), 1619
MON_12 (*locale* 모듈), 1619
MONDAY (*calendar* 모듈), 266
monotonic() (*time* 모듈), 774
monotonic_ns() (*time* 모듈), 774
month
    calendar 명령줄 옵션, 268
Month (*calendar* 클래스), 267
month (*calendar.IllegalMonthError*의 속성), 267
month (*datetime.datetime*의 속성), 232
month (*datetime.date*의 속성), 225
month() (*calendar* 모듈), 265
month_abbr (*calendar* 모듈), 266
month_name (*calendar* 모듈), 266
monthcalendar() (*calendar* 모듈), 265
monthdatescalendar() (*calendar.Calendar* 메서드), 262
monthdays2calendar() (*calendar.Calendar* 메서드), 262
monthdayscalendar() (*calendar.Calendar* 메서드), 262
monthrange() (*calendar* 모듈), 265
--months
    calendar 명령줄 옵션, 269
Morsel (*http.cookies* 클래스), 1564
most_common() (*collections.Counter* 메서드), 273
mouseinterval() (*curses* 모듈), 961
mousemask() (*curses* 모듈), 961
move() (*curses.panel.Panel* 메서드), 988
move() (*curses.window* 메서드), 969
move() (*mmap.mmap* 메서드), 1300
move() (*pathlib.Path* 메서드), 482
move() (*shutil* 모듈), 515
move() (*tkinter.ttk.Treeview* 메서드), 1660
move_into() (*pathlib.Path* 메서드), 482
move_to_end() (*collections.OrderedDict* 메서드), 284
MozillaCookieJar (*http.cookiejar* 클래스), 1570
MRO, **2312**
msg (*http.client.HTTPResponse*의 속성), 1515
msg (*json.JSONDecodeError*의 속성), 1367
msg (*netrc.NetrcParseError*의 속성), 669
msg (*re.PatternError*의 속성), 154
msg (*tomllib.TOMLDecodeError*의 속성), 668
msg (*traceback.TracebackException*의 속성), 2069
msilib
    module, 2295
msvcrt
    module, 2247
mtime (*gzip.GzipFile*의 속성), 599
mtime (*tarfile.TarInfo*의 속성), 631
mtime() (*urllib.robotparser.RobotFileParser* 메서드), 1506
mul() (*operator* 모듈), 453
Mult (*ast* 클래스), 2171
MultiCall (*xmlrpc.client* 클래스), 1581
MULTILINE (*re* 모듈), 150
MultilineContinuationError, 667
multimode() (*statistics* 모듈), 413
MultipartConversionError, 1325
MultipartInvariantViolationDefect, 1326
multiply() (*decimal.Context* 메서드), 381
multiprocessing
    module, 1012
multiprocessing.connection
    module, 1044
multiprocessing.dummy
    module, 1048
multiprocessing.Manager()
    built-in function, 1035
multiprocessing.managers
    module, 1035
multiprocessing.pool
    module, 1041
multiprocessing.shared_memory
    module, 1057
multiprocessing.sharedctypes
    module, 1033
mutable
    sequence types, 49
mutable (가변), **2312**
mutable sequence
    loop over, 47
MutableMapping (*collections.abc* 클래스), 291
MutableMapping (*typing* 클래스), 1769
MutableSequence (*collections.abc* 클래스), 290
MutableSequence (*typing* 클래스), 1770
MutableSet (*collections.abc* 클래스), 291
MutableSet (*typing* 클래스), 1770
mvderwin() (*curses.window* 메서드), 969
mvwin() (*curses.window* 메서드), 969
myrights() (*imaplib.IMAP4* 메서드), 1531

N

-N
    uuid 명령줄 옵션, 1545
-n
    timeit 명령줄 옵션, 1949
    uuid 명령줄 옵션, 1545
    webbrowser 명령줄 옵션, 1465
N_TOKENS (*token* 모듈), 2209
n_waiting (*asyncio.Barrier*의 속성), 1146

n_waiting (*threading.Barrier*의 속성), 1011
NAK (*curses.ascii* 모듈), 985
--name
    uuid 명령줄 옵션, 1545
Name (*ast* 클래스), 2169
name (*AttributeError*의 속성), 113
name (*bz2.BZ2File*의 속성), 603
name (*codecs.CodecInfo*의 속성), 199
name (*compression.zstd.ZstdFile*의 속성), 585
name (*contextvars.ContextVar*의 속성), 1101
name (*ctypes.CField*의 속성), 877
name (*doctest.DocTest*의 속성), 1792
name (*email.headerregistry.BaseHeader*의 속성), 1327
name (*enum.Enum*의 속성), 331
name (*gzip.GzipFile*의 속성), 599
name (*hashlib.hash*의 속성), 675
name (*hmac.HMAC*의 속성), 686
name (*http.cookiejar.Cookie*의 속성), 1574
name (*ImportError*의 속성), 113
name (*importlib.abc.FileLoader*의 속성), 2131
name (*importlib.abc.Traversable*의 속성), 2134
name (*importlib.machinery.AppleFrameworkLoader* 의 속성), 2140
name (*importlib.machinery.ExtensionFileLoader* 의 속성), 2138
name (*importlib.machinery.ModuleSpec*의 속성), 2139
name (*importlib.machinery.SourceFileLoader*의 속성), 2137
name (*importlib.machinery.SourcelessFileLoader*의 속성), 2137
name (*importlib.resources.abc.Traversable* 의 속성), 2149
name (*inspect.Parameter*의 속성), 2089
name (*io.FileIO*의 속성), 764
name (*logging.Logger*의 속성), 785
name (*lzma.LZMAFile*의 속성), 607
name (*multiprocessing.Process*의 속성), 1021
name (*multiprocessing.shared_memory.SharedMemory* 의 속성), 1059
name (*NameError*의 속성), 114
name (*os* 모듈), 690
name (*os.DirEntry*의 속성), 722
name (*pathlib.PurePath*의 속성), 466
name (*pyclbr.Class*의 속성), 2216
name (*pyclbr.Function*의 속성), 2216
name (*sys.thread_info*의 속성), 2006
name (*tarfile.TarInfo*의 속성), 631
name (*tempfile.TemporaryDirectory*의 속성), 503
name (*threading.Thread*의 속성), 1003
NAME (*token* 모듈), 2205
name (*traceback.FrameSummary*의 속성), 2070
name (*webbrowser.controller*의 속성), 1467
name (*xml.dom.Attr*의 속성), 1433
name (*xml.dom.DocumentType*의 속성), 1431
name (*zipfile.Path*의 속성), 618
name() (*unicodedata* 모듈), 181
name2codepoint (*html.entities* 모듈), 1406
named tuple (네임드 튜플), **2312**
NAMED_FLAGS (*enum.EnumCheck*의 속성), 339
NamedExpr (*ast* 클래스), 2173
NamedTemporaryFile() (*tempfile* 모듈), 501
NamedTuple (*typing* 클래스), 1750
namedtuple() (*collections* 모듈), 280
NameError, 114
namelist() (*zipfile.ZipFile* 메서드), 614
nameprep() (*encodings.idna* 모듈), 215
namer (*logging.handlers.BaseRotatingHandler*의 속성), 816
namereplace
    에러 처리기의 이름, 202
namereplace_errors() (*codecs* 모듈), 203
names() (*tkinter.font* 모듈), 1641
--namespace
    uuid 명령줄 옵션, 1545
Namespace (*argparse* 클래스), 903
Namespace (*multiprocessing.managers* 클래스), 1037
namespace (이름 공간), **2312**
namespace package (이름 공간 패키지), **2312**
namespace() (*imaplib.IMAP4* 메서드), 1531
Namespace() (*multiprocessing.managers.SyncManager* 메서드), 1037
NAMESPACE_DNS (*uuid* 모듈), 1544
NAMESPACE_OID (*uuid* 모듈), 1544
NAMESPACE_URL (*uuid* 모듈), 1544
NAMESPACE_X500 (*uuid* 모듈), 1544
NamespaceErr, 1435
NamespaceLoader (*importlib.machinery* 클래스), 2138
namespaceURI (*xml.dom.Node*의 속성), 1429
nametofont() (*tkinter.font* 모듈), 1641
NaN, 16
nan (*cmath* 모듈), 363
nan (*math* 모듈), 359
nan (*sys.hash_info*의 속성), 1994
nanj (*cmath* 모듈), 363
NannyNag, 2215
napms() (*curses* 모듈), 961
nargs (*optparse.Option*의 속성), 940
native_id (*threading.Thread*의 속성), 1003
nb_workers (*compression.zstd.CompressionParameter* 의 속성), 591
nbytes (*memoryview*의 속성), 89
ncurses_version (*curses* 모듈), 971
ND (*inspect.BufferFlags*의 속성), 2099
ndiff() (*difflib* 모듈), 167
ndim (*memoryview*의 속성), 90
ne() (*operator* 모듈), 451
needs_input (*bz2.BZ2Decompressor*의 속성), 604
needs_input (*compression.zstd.ZstdDecompressor* 의 속성), 587
needs_input (*lzma.LZMADecompressor*의 속성), 609
neg() (*operator* 모듈), 453
nested scope (중첩된 스코프), **2312**
nested_scopes (*__future__* 모듈), 2076
netmask (*ipaddress.IPv4Network*의 속성), 1596
netmask (*ipaddress.IPv6Network*의 속성), 1599

NetmaskValueError, 1603
netrc
    module, 669
netrc (*netrc* 클래스), 669
NetrcParseError, 669
netscape (*http.cookiejar.CookiePolicy*의 속성), 1571
network (*ipaddress.IPv4Interface*의 속성), 1601
network (*ipaddress.IPv6Interface*의 속성), 1602
network_address (*ipaddress.IPv4Network*의 속성), 1596
network_address (*ipaddress.IPv6Network*의 속성), 1599
Never (*typing* 모듈), 1730
NEVER_EQ (*test.support* 모듈), 1898
new() (*hashlib* 모듈), 674
new() (*hmac* 모듈), 685
new-style class (뉴스타일 클래스), **2312**
new_child() (*collections.ChainMap* 메서드), 270
new_class() (*types* 모듈), 311
new_event_loop() (*asyncio* 모듈), 1157
new_event_loop() (*asyncio.AbstractEventLoopPolicy* 메서드), 1198
new_panel() (*curses.panel* 모듈), 987
NEWLINE (*token* 모듈), 2205
newlines (*io.TextIOBase*의 속성), 767
newpad() (*curses* 모듈), 961
--new-tab
    webbrowser 명령줄 옵션, 1466
NewType (*typing* 클래스), 1751
newwin() (*curses* 모듈), 961
--new-window
    webbrowser 명령줄 옵션, 1465
next (*pdb command*), 1934
next()
    built-in function, 22
next() (*tarfile.TarFile* 메서드), 628
next() (*tkinter.ttk.Treeview* 메서드), 1660
next_minus() (*decimal.Context* 메서드), 381
next_minus() (*decimal.Decimal* 메서드), 373
next_plus() (*decimal.Context* 메서드), 381
next_plus() (*decimal.Decimal* 메서드), 373
next_toward() (*decimal.Context* 메서드), 381
next_toward() (*decimal.Decimal* 메서드), 373
nextafter() (*math* 모듈), 354
nextfile() (*fileinput* 모듈), 956
nextkey() (*dbm.gnu.gdbm* 메서드), 548
nextSibling (*xml.dom.Node*의 속성), 1429
ngettext() (*gettext* 모듈), 1610
ngettext() (*gettext.GNUTranslations* 메서드), 1613
ngettext() (*gettext.NullTranslations* 메서드), 1612
nice() (*os* 모듈), 743
NIL (*uuid* 모듈), 1545
nis
    module, 2295
NL (*curses.ascii* 모듈), 984
NL (*token* 모듈), 2205
nl() (*curses* 모듈), 961
nl_langinfo() (*locale* 모듈), 1618
nlargest() (*heapq* 모듈), 295
nlst() (*ftplib.FTP* 메서드), 1521
nntplib
    module, 2295
NO (*tkinter.messagebox* 모듈), 1646
no_animation() (*turtle* 모듈), 1704
no_cache() (*zoneinfo.ZoneInfo*의 클래스 메서드), 258
NO_EVENTS (*monitoring event*), 2010
no_proxy, 1482
no_site (*sys.flags*의 속성), 1988
no_tracing() (*test.support* 모듈), 1902
no_type_check() (*typing* 모듈), 1762
no_type_check_decorator() (*typing* 모듈), 1762
no_user_site (*sys.flags*의 속성), 1988
NoBoundaryInMultipartDefect, 1325
nocbreak() (*curses* 모듈), 961
NoDataAllowedErr, 1435
node (*uuid.UUID*의 속성), 1543
node() (*platform* 모듈), 829
NoDefault (*typing* 모듈), 1766
nodelay() (*curses.window* 메서드), 969
nodeName (*xml.dom.Node*의 속성), 1429
NodeTransformer (*ast* 클래스), 2198
nodeType (*xml.dom.Node*의 속성), 1429
nodeValue (*xml.dom.Node*의 속성), 1429
NodeVisitor (*ast* 클래스), 2197
noecho() (*curses* 모듈), 961
--no-ensure-ascii
    json 명령줄 옵션, 1370
NOEXPR (*locale* 모듈), 1620
NOFLAG (*re* 모듈), 150
--no-indent
    json 명령줄 옵션, 1370
NoModificationAllowedErr, 1435
--nonaliased
    platform 명령줄 옵션, 832
NonCallableMagicMock (*unittest.mock* 클래스), 1862
NonCallableMock (*unittest.mock* 클래스), 1842
None (*Built-in object*), 37
None (내장 변수), 35
NoneType (*types* 모듈), 313
nonl() (*curses* 모듈), 961
Nonlocal (*ast* 클래스), 2193
nonmember() (*enum* 모듈), 342
noop() (*imaplib.IMAP4* 메서드), 1531
noop() (*poplib.POP3* 메서드), 1526
NoOptionError, 666
NOP (*opcode*), 2228
noqiflush() (*curses* 모듈), 962
noraw() (*curses* 모듈), 962
--no-report
    trace 명령줄 옵션, 1953
NoReturn (*typing* 모듈), 1730
NORMAL (*tkinter.font* 모듈), 1640
NORMAL_PRIORITY_CLASS (*subprocess* 모듈), 1090
NormalDist (*statistics* 클래스), 417

normalize() (*decimal.Context* 메서드), 381
normalize() (*decimal.Decimal* 메서드), 373
normalize() (*locale* 모듈), 1622
normalize() (*unicodedata* 모듈), 182
normalize() (*xml.dom.Node* 메서드), 1430
normalize_encoding() (*encodings* 모듈), 214
NORMALIZE_WHITESPACE (*doctest* 모듈), 1784
normalvariate() (*random* 모듈), 401
normcase() (*os.path* 모듈), 490
normpath() (*os.path* 모듈), 490
NoSectionError, 666
NoSuchMailboxError, 1388
not
    operator, 38
Not (*ast* 클래스), 2171
not in
    operator, 38, 47
not_() (*operator* 모듈), 452
NOT_TAKEN (*opcode*), 2228
NotADirectoryError, 119
notationDecl() (*xml.sax.handler.DTDHandler* 메서드), 1448
NotationDeclHandler() (*xml.parsers.expat.xmlparser* 메서드), 1459
notations (*xml.dom.DocumentType*의 속성), 1431
NotConnected, 1511
Notebook (*tkinter.ttk* 클래스), 1654
NotEmptyError, 1388
NotEq (*ast* 클래스), 2172
NOTEQUAL (*token* 모듈), 2208
NotFoundErr, 1435
notify() (*asyncio.Condition* 메서드), 1143
notify() (*threading.Condition* 메서드), 1008
notify_all() (*asyncio.Condition* 메서드), 1143
notify_all() (*threading.Condition* 메서드), 1008
notimeout() (*curses.window* 메서드), 969
NotImplemented (내장 변수), 35
NotImplementedError, 114
NotImplementedType (*types* 모듈), 314
NotIn (*ast* 클래스), 2172
NotRequired (*typing* 모듈), 1736
NOTSET (*logging* 모듈), 790
NotShareableError, 1076
NotStandaloneHandler() (*xml.parsers.expat.xmlparser* 메서드), 1459
NotSupportedErr, 1435
NotSupportedError, 572
--no-type-comments
    ast 명령줄 옵션, 2200
noutrefresh() (*curses.window* 메서드), 969
NOVEMBER (*calendar* 모듈), 266
now() (*datetime.datetime*의 클래스 메서드), 229
npgettext() (*gettext* 모듈), 1610
npgettext() (*gettext.GNUTranslations* 메서드), 1613
npgettext() (*gettext.NullTranslations* 메서드), 1612
NSIG (*signal* 모듈), 1291
nsmallest() (*heapq* 모듈), 295
NTEventLogHandler (*logging.handlers* 클래스), 823
ntohl() (*socket* 모듈), 1227
ntohs() (*socket* 모듈), 1227
ntransfercmd() (*ftplib.FTP* 메서드), 1521
NUL (*curses.ascii* 모듈), 984
nullcontext() (*contextlib* 모듈), 2047
NullHandler (*logging* 클래스), 815
NullTranslations (*gettext* 클래스), 1611
num_addresses (*ipaddress.IPv4Network* 의 속성), 1597
num_addresses (*ipaddress.IPv6Network* 의 속성), 1599
num_tickets (*ssl.SSLContext*의 속성), 1265
--number
    timeit 명령줄 옵션, 1949
Number (*numbers* 클래스), 347
NUMBER (*token* 모듈), 2205
number_class() (*decimal.Context* 메서드), 381
number_class() (*decimal.Decimal* 메서드), 373
numbers
    module, 347
numerator (*fractions.Fraction*의 속성), 394
numerator (*numbers.Rational*의 속성), 348
numeric
    conversions, 39
    literals, 39
    object, 38
    types, operations on, 39
numeric() (*unicodedata* 모듈), 181
numinput() (*turtle* 모듈), 1706

O

-O
    ast 명령줄 옵션, 2200
    dis 명령줄 옵션, 2223
-o
    compileall 명령줄 옵션, 2220
    cProfile 명령줄 옵션, 1940
    doctest 명령줄 옵션, 1780
    pickletools 명령줄 옵션, 2246
    zipapp 명령줄 옵션, 1977
O_APPEND (*os* 모듈), 704
O_ASYNC (*os* 모듈), 705
O_BINARY (*os* 모듈), 704
O_CLOEXEC (*os* 모듈), 704
O_CREAT (*os* 모듈), 704
O_DIRECT (*os* 모듈), 705
O_DIRECTORY (*os* 모듈), 705
O_DSYNC (*os* 모듈), 704
O_EVTONLY (*os* 모듈), 704
O_EXCL (*os* 모듈), 704
O_EXLOCK (*os* 모듈), 705
O_FSYNC (*os* 모듈), 704
O_NDELAY (*os* 모듈), 704
O_NOATIME (*os* 모듈), 705
O_NOCTTY (*os* 모듈), 704
O_NOFOLLOW (*os* 모듈), 705
O_NOFOLLOW_ANY (*os* 모듈), 704
O_NOINHERIT (*os* 모듈), 704

O_NONBLOCK (*os* 모듈), 704
O_PATH (*os* 모듈), 705
O_RANDOM (*os* 모듈), 704
O_RDONLY (*os* 모듈), 704
O_RDWR (*os* 모듈), 704
O_RSYNC (*os* 모듈), 704
O_SEQUENTIAL (*os* 모듈), 704
O_SHLOCK (*os* 모듈), 705
O_SHORT_LIVED (*os* 모듈), 704
O_SYMLINK (*os* 모듈), 704
O_SYNC (*os* 모듈), 704
O_TEMPORARY (*os* 모듈), 704
O_TEXT (*os* 모듈), 704
O_TMPFILE (*os* 모듈), 705
O_TRUNC (*os* 모듈), 704
O_WRONLY (*os* 모듈), 704
obj (*AttributeError*의 속성), 113
obj (*memoryview*의 속성), 89
object
    Boolean, 38
    bytearray, 49, 67, 69
    bytes, 67, 68
    code, 106
    complex number, 38
    dictionary, 93
    floating-point, 38
    GenericAlias, 99
    integer, 38
    io.StringIO, 54
    list, 49, 50
    mapping, 93
    memoryview, 67
    method, 106
    numeric, 38
    range, 51
    sequence, 47
    set, 91
    socket, 1212
    string, 53
    tuple, 49, 51
    type, 31
    Union, 103
object (*UnicodeError*의 속성), 118
object (객체), **2312**
object (내장 클래스), 22
objects
    comparing, 38
oct()
    built-in function, 22
octal
    literals, 39
octdigits (*string* 모듈), 125
OCTOBER (*calendar* 모듈), 266
offset (*ctypes.CField*의 속성), 877
offset (*SyntaxError*의 속성), 116
offset (*tarfile.TarInfo*의 속성), 632
offset (*traceback.TracebackException*의 속성), 2069
offset (*xml.parsers.expat.ExpatError*의 속성), 1460
offset_data (*tarfile.TarInfo*의 속성), 632
OK (*curses* 모듈), 971
OK (*tkinter.messagebox* 모듈), 1646
ok_command() (*tkinter.filedialog.LoadFileDialog* 메서드), 1644
ok_command() (*tkinter.filedialog.SaveFileDialog* 메서드), 1644
ok_event() (*tkinter.filedialog.FileDialog* 메서드), 1644
OKCANCEL (*tkinter.messagebox* 모듈), 1646
old_value (*contextvars.Token*의 속성), 1102
OleDLL (*ctypes* 클래스), 861
on_motion() (*tkinter.dnd.DndHandler* 메서드), 1648
on_release() (*tkinter.dnd.DndHandler* 메서드), 1648
onclick() (*turtle* 모듈), 1705
ondrag() (*turtle* 모듈), 1700
onecmd() (*cmd.Cmd* 메서드), 989
onkey() (*turtle* 모듈), 1704
onkeypress() (*turtle* 모듈), 1705
onkeyrelease() (*turtle* 모듈), 1704
onrelease() (*turtle* 모듈), 1699
onscreenclick() (*turtle* 모듈), 1705
ontimer() (*turtle* 모듈), 1705
OP (*token* 모듈), 2205
OP_ALL (*ssl* 모듈), 1249
OP_CIPHER_SERVER_PREFERENCE (*ssl* 모듈), 1250
OP_ENABLE_KTLS (*ssl* 모듈), 1251
OP_ENABLE_MIDDLEBOX_COMPAT (*ssl* 모듈), 1251
OP_IGNORE_UNEXPECTED_EOF (*ssl* 모듈), 1251
OP_LEGACY_SERVER_CONNECT (*ssl* 모듈), 1251
OP_NO_COMPRESSION (*ssl* 모듈), 1251
OP_NO_RENEGOTIATION (*ssl* 모듈), 1250
OP_NO_SSLv2 (*ssl* 모듈), 1249
OP_NO_SSLv3 (*ssl* 모듈), 1250
OP_NO_TICKET (*ssl* 모듈), 1251
OP_NO_TLSv1 (*ssl* 모듈), 1250
OP_NO_TLSv1_1 (*ssl* 모듈), 1250
OP_NO_TLSv1_2 (*ssl* 모듈), 1250
OP_NO_TLSv1_3 (*ssl* 모듈), 1250
OP_SINGLE_DH_USE (*ssl* 모듈), 1250
OP_SINGLE_ECDH_USE (*ssl* 모듈), 1250
Open (*tkinter.filedialog* 클래스), 1643
open()
    built-in function, 23
open() (*bz2* 모듈), 602
open() (*codecs* 모듈), 200
open() (*compression.zstd* 모듈), 584
open() (*dbm* 모듈), 545
open() (*dbm.dumb* 모듈), 550
open() (*dbm.gnu* 모듈), 547
open() (*dbm.ndbm* 모듈), 549
open() (*dbm.sqlite3* 모듈), 547
open() (*gzip* 모듈), 598
open() (*imaplib.IMAP4* 메서드), 1531
open() (*importlib.abc.Traversable* 메서드), 2134
open() (*importlib.resources.abc.Traversable* 메서드), 2150

open() (*io* 모듈), 758
open() (*lzma* 모듈), 606
open() (*os* 모듈), 703
open() (*pathlib.Path* 메서드), 476
open() (*shelve* 모듈), 540
open() (*tarfile* 모듈), 623
open() (*tarfile.TarFile*의 클래스 메서드), 628
open() (*tokenize* 모듈), 2211
open() (*urllib.request.OpenerDirector* 메서드), 1486
open() (*wave* 모듈), 1605
open() (*webbrowser* 모듈), 1466
open() (*webbrowser.controller* 메서드), 1468
open() (*zipfile.Path* 메서드), 618
open() (*zipfile.ZipFile* 메서드), 615
open_binary() (*importlib.resources* 모듈), 2147
open_code() (*io* 모듈), 759
open_connection() (*asyncio* 모듈), 1133
open_flags (*dbm.gnu* 모듈), 548
open_new() (*webbrowser* 모듈), 1466
open_new() (*webbrowser.controller* 메서드), 1468
open_new_tab() (*webbrowser* 모듈), 1466
open_new_tab() (*webbrowser.controller* 메서드), 1468
open_osfhandle() (*msvcrt* 모듈), 2248
open_resource() (*importlib.abc.ResourceReader* 메서드), 2133
open_resource() (*importlib.resources.abc.ResourceReader* 메서드), 2149
open_text() (*importlib.resources* 모듈), 2147
open_unix_connection() (*asyncio* 모듈), 1134
open_urlresource() (*test.support* 모듈), 1903
OpenerDirector (*urllib.request* 클래스), 1482
OpenKey() (*winreg* 모듈), 2253
OpenKeyEx() (*winreg* 모듈), 2253
openlog() (*syslog* 모듈), 2283
openpty() (*os* 모듈), 705
openpty() (*pty* 모듈), 2274
OpenSSL
    (use in module ssl), 1241
    (모듈 hashlib 에서의 사용), 673
OPENSSL_VERSION (*ssl* 모듈), 1252
OPENSSL_VERSION_INFO (*ssl* 모듈), 1253
OPENSSL_VERSION_NUMBER (*ssl* 모듈), 1253
operation
    concatenation, 47
    repetition, 47
    slice, 47
    subscript, 47
OperationalError, 572
operations
    bitwise, 40
    Boolean, 37
    masking, 40
    shifting, 40
operations on
    dictionary type, 93
    integer types, 40
    list type, 49
    mapping types, 93
    numeric types, 39
    sequence types, 47, 49
operator
    - (*minus*), 39
    % (*percent*), 39
    & (*ampersand*), 40
    * (*asterisk*), 39
    **, 39
    + (*plus*), 39
    / (*slash*), 39
    //, 39
    < (*less*), 38
    <<, 40
    <=, 38
    !=, 38
    ==, 38
    > (*greater*), 38
    >=, 38
    >>, 40
    ^ (*caret*), 40
    | (*vertical bar*), 40
    ~ (*tilde*), 40
    and, 37, 38
    comparison, 38
    in, 38, 47
    is, 38
    is not, 38
    module, 451
    not, 38
    not in, 38, 47
    or, 37, 38
opmap (*dis* 모듈), 2244
opname (*dis* 모듈), 2244
optim_args_from_interpreter_flags() (*test.support* 모듈), 1900
--optimize
    ast 명령줄 옵션, 2200
optimize (*sys.flags*의 속성), 1988
optimize() (*pickletools* 모듈), 2246
optimized scope (최적화된 스코프), **2312**
OPTIMIZED_BYTECODE_SUFFIXES (*importlib.machinery* 모듈), 2135
--option
    doctest 명령줄 옵션, 1780
Option (*optparse* 클래스), 940
Optional (*typing* 모듈), 1733
OptionConflictError, 953
OptionError, 953
OptionGroup (*optparse* 클래스), 934
OptionParser (*optparse* 클래스), 937
options (*doctest.Example*의 속성), 1793
Options (*ssl* 클래스), 1251
options (*ssl.SSLContext*의 속성), 1265
options() (*configparser.ConfigParser* 메서드), 663
OptionValueError, 953

optionxform() (*configparser.ConfigParser* 메서드), 665
optparse
    module, 926
or
    operator, 37, 38
Or (*ast* 클래스), 2171
or_() (*operator* 모듈), 453
ord()
    built-in function, 25
ordered_attributes (*xml.parsers.expat.xmlparser*의 속성), 1457
OrderedDict (*collections* 클래스), 283
OrderedDict (*typing* 클래스), 1767
orig_argv (*sys* 모듈), 1999
origin (*importlib.machinery.ModuleSpec* 의 속성), 2139
origin_req_host (*urllib.request.Request* 의 속성), 1484
origin_server (*wsgiref.handlers.BaseHandler* 의 속성), 1476
os
    module, 689
os_environ (*wsgiref.handlers.BaseHandler* 의 속성), 1475
OSError, 115
os.path
    module, 486
ossaudiodev
    module, 2296
OUT_TO_DEFAULT (*msvcrt* 모듈), 2249
OUT_TO_MSGBOX (*msvcrt* 모듈), 2249
OUT_TO_STDERR (*msvcrt* 모듈), 2249
outfile
    json 명령줄 옵션, 1370
--output
    pickletools 명령줄 옵션, 2246
    zipapp 명령줄 옵션, 1977
output (*subprocess.CalledProcessError*의 속성), 1079
output (*subprocess.TimeoutExpired*의 속성), 1079
output (*unittest.TestCase*의 속성), 1814
output() (*http.cookies.BaseCookie* 메서드), 1564
output() (*http.cookies.Morsel* 메서드), 1565
output_charset (*email.charset.Charset* 의 속성), 1355
output_codec (*email.charset.Charset*의 속성), 1355
output_difference() (*doctest.OutputChecker* 메서드), 1796
OutputChecker (*doctest* 클래스), 1796
OutputString() (*http.cookies.Morsel* 메서드), 1566
OutsideDestinationError, 625
Overflow (*decimal* 클래스), 384
OverflowError, 115
overlap() (*statistics.NormalDist* 메서드), 418
overlap_log (*compression.zstd.CompressionParameter* 의 속성), 591
overlaps() (*ipaddress.IPv4Network* 메서드), 1597
overlaps() (*ipaddress.IPv6Network* 메서드), 1600
overlay() (*curses.window* 메서드), 969
overload() (*typing* 모듈), 1761
override() (*typing* 모듈), 1762
overwrite() (*curses.window* 메서드), 969
owner() (*pathlib.Path* 메서드), 482

P

-P
    dis 명령줄 옵션, 2223
-p
    compileall 명령줄 옵션, 2219
    http.server 명령줄 옵션, 1562
    pdb 명령줄 옵션, 1929
    pickletools 명령줄 옵션, 2246
    timeit 명령줄 옵션, 1949
    unittest-discover 명령줄 옵션, 1804
    zipapp 명령줄 옵션, 1977
p (*pdb command*), 1935
P_ALL (*os* 모듈), 749
P_DETACH (*os* 모듈), 746
P_NOWAIT (*os* 모듈), 746
P_NOWAITO (*os* 모듈), 746
P_OVERLAY (*os* 모듈), 746
P_PGID (*os* 모듈), 749
P_PID (*os* 모듈), 749
P_PIDFD (*os* 모듈), 749
P_WAIT (*os* 모듈), 746
pack() (*mailbox.MH* 메서드), 1378
pack() (*struct* 모듈), 192
pack() (*struct.Struct* 메서드), 198
pack_into() (*struct* 모듈), 192
pack_into() (*struct.Struct* 메서드), 198
package (패키지), **2313**
PackageMetadata (*importlib.metadata* 클래스), 2153
PackageNotFoundError, 2151
PackagePath (*importlib.metadata* 클래스), 2154
packages_distributions() (*importlib.metadata* 모듈), 2155
packed (*ipaddress.IPv4Address*의 속성), 1591
packed (*ipaddress.IPv6Address*의 속성), 1593
PAGER, 1773
pair_content() (*curses* 모듈), 962
pair_number() (*curses* 모듈), 962
pairwise() (*itertools* 모듈), 430
Parameter (*inspect* 클래스), 2089
parameter (매개변수), **2313**
ParameterizedMIMEHeader (*email.headerregistry* 클래스), 1329
parameters (*inspect.Signature*의 속성), 2088
params (*email.headerregistry.ParameterizedMIMEHeader* 의 속성), 1329
ParamSpec (*ast* 클래스), 2190
ParamSpec (*typing* 클래스), 1747
ParamSpecArgs (*typing* 모듈), 1748
ParamSpecKwargs (*typing* 모듈), 1748
paramstyle (*sqlite3* 모듈), 556
pardir (*os* 모듈), 755

parent (*importlib.machinery.ModuleSpec* 의 속성), 2139
parent (*logging.Logger* 의 속성), 785
parent (*pathlib.PurePath* 의 속성), 465
parent (*pyclbr.Class* 의 속성), 2216
parent (*pyclbr.Function* 의 속성), 2216
parent (*urllib.request.BaseHandler* 의 속성), 1486
parent() (*tkinter.ttk.Treeview* 메서드), 1660
parent_process() (*multiprocessing* 모듈), 1026
parentNode (*xml.dom.Node* 의 속성), 1429
parents (*collections.ChainMap* 의 속성), 270
parents (*pathlib.PurePath* 의 속성), 465
paretovariate() (*random* 모듈), 401
parse() (*ast* 모듈), 2195
parse() (*doctest.DocTestParser* 메서드), 1794
parse() (*email.parser.BytesParser* 메서드), 1313
parse() (*email.parser.Parser* 메서드), 1314
parse() (*string.Formatter* 메서드), 126
parse() (*urllib.robotparser.RobotFileParser* 메서드), 1505
parse() (*xml.dom.minidom* 모듈), 1437
parse() (*xml.dom.pulldom* 모듈), 1442
parse() (*xml.etree.ElementTree* 모듈), 1416
parse() (*xml.etree.ElementTree.ElementTree* 메서드), 1422
Parse() (*xml.parsers.expat.xmlparser* 메서드), 1456
parse() (*xml.sax* 모듈), 1443
parse() (*xml.sax.xmlreader.XMLReader* 메서드), 1451
parse_and_bind() (*readline* 모듈), 185
parse_args() (*argparse.ArgumentParser* 메서드), 900
parse_args() (*optparse.OptionParser* 메서드), 944
PARSE_COLNAMES (*sqlite3* 모듈), 555
parse_config_h() (*sysconfig* 모듈), 2018
PARSE_DECLTYPES (*sqlite3* 모듈), 555
parse_headers() (*http.client* 모듈), 1511
parse_intermixed_args() (*argparse.ArgumentParser* 메서드), 911
parse_known_args() (*argparse.ArgumentParser* 메서드), 910
parse_known_intermixed_args() (*argparse.ArgumentParser* 메서드), 911
parse_qs() (*urllib.parse* 모듈), 1498
parse_qsl() (*urllib.parse* 모듈), 1498
parseaddr() (*email.utils* 모듈), 1357
parsebytes() (*email.parser.BytesParser* 메서드), 1314
parsedate() (*email.utils* 모듈), 1358
parsedate_to_datetime() (*email.utils* 모듈), 1358
parsedate_tz() (*email.utils* 모듈), 1358
ParseError (*xml.etree.ElementTree* 클래스), 1426
ParseFile() (*xml.parsers.expat.xmlparser* 메서드), 1456
ParseFlags() (*imaplib* 모듈), 1528
Parser (*email.parser* 클래스), 1314
parser (*pathlib.PurePath* 의 속성), 464
ParserCreate() (*xml.parsers.expat* 모듈), 1455
ParseResult (*urllib.parse* 클래스), 1502
ParseResultBytes (*urllib.parse* 클래스), 1502
parsestr() (*email.parser.Parser* 메서드), 1314
parseString() (*xml.dom.minidom* 모듈), 1437
parseString() (*xml.dom.pulldom* 모듈), 1442
parseString() (*xml.sax* 모듈), 1443
ParsingError, 667
partial (*asyncio.IncompleteReadError* 의 속성), 1154
partial() (*functools* 모듈), 444
partial() (*imaplib.IMAP4* 메서드), 1532
partialmethod (*functools* 클래스), 445
parties (*asyncio.Barrier* 의 속성), 1146
parties (*threading.Barrier* 의 속성), 1011
partition() (*bytearray* 메서드), 73
partition() (*bytes* 메서드), 73
partition() (*str* 메서드), 60
partitioned (*http.cookies.Morsel* 의 속성), 1565
parts (*pathlib.PurePath* 의 속성), 464
Pass (*ast* 클래스), 2178
pass_() (*poplib.POP3* 메서드), 1525
Paste, 1670
patch() (*test.support* 모듈), 1904
patch() (*unittest.mock* 모듈), 1851
patch.dict() (*unittest.mock* 모듈), 1855
patch.multiple() (*unittest.mock* 모듈), 1856
patch.object() (*unittest.mock* 모듈), 1854
patch.stopall() (*unittest.mock* 모듈), 1858
PATH, 484, 516, 739, 745, 746, 755, 1081, 1465, 1969, 1970, 2109
path
  operations, 459
path (*http.cookiejar.Cookie* 의 속성), 1574
path (*http.cookies.Morsel* 의 속성), 1565
path (*http.server.BaseHTTPRequestHandler* 의 속성), 1557
path (*ImportError* 의 속성), 113
path (*importlib.abc.FileLoader* 의 속성), 2131
path (*importlib.machinery.AppleFrameworkLoader* 의 속성), 2140
path (*importlib.machinery.ExtensionFileLoader* 의 속성), 2138
path (*importlib.machinery.FileFinder* 의 속성), 2136
path (*importlib.machinery.SourceFileLoader* 의 속성), 2137
path (*importlib.machinery.SourcelessFileLoader* 의 속성), 2137
path (*os.DirEntry* 의 속성), 723
Path (*pathlib* 클래스), 471
path (*sys* 모듈), 1999
Path (*zipfile* 클래스), 618
path based finder (경로 기반 파인더), **2313**
path entry (경로 엔트리), **2313**
path entry finder (경로 엔트리 파인더), **2313**
path entry hook (경로 엔트리 훅), **2313**
path() (*importlib.resources* 모듈), 2148
path-like object (경로류 객체), **2313**
path_hook() (*importlib.machinery.FileFinder* 의 클래스 메서드), 2137

path_hooks (*sys* 모듈), 1999
path_importer_cache (*sys* 모듈), 1999
path_mtime() (*importlib.abc.SourceLoader* 메서드), 2132
path_return_ok() (*http.cookiejar.CookiePolicy* 메서드), 1571
path_stats() (*importlib.abc.SourceLoader* 메서드), 2132
path_stats() (*importlib.machinery.SourceFileLoader* 메서드), 2137
pathconf() (*os* 모듈), 719
pathconf_names (*os* 모듈), 719
PathEntryFinder (*importlib.abc* 클래스), 2129
PATHEXT, 516
PathFinder (*importlib.machinery* 클래스), 2136
PathInfo (*pathlib.types* 클래스), 485
pathlib
    module, 459
pathlib.types
    module, 485
PathLike (*os* 클래스), 693
pathname2url() (*urllib.request* 모듈), 1480
pathsep (*os* 모듈), 755
Path.stem (*zipfile* 모듈), 619
Path.suffix (*zipfile* 모듈), 618
Path.suffixes (*zipfile* 모듈), 619
--pattern
    unittest-discover 명령줄 옵션, 1804
Pattern (*re* 클래스), 155
pattern (*re.PatternError*의 속성), 154
pattern (*re.Pattern*의 속성), 156
Pattern (*typing* 클래스), 1768
PatternError, 154
pause() (*signal* 모듈), 1292
pause_reading() (*asyncio.ReadTransport* 메서드), 1187
pause_writing() (*asyncio.BaseProtocol* 메서드), 1190
PAX_FORMAT (*tarfile* 모듈), 627
pax_headers (*tarfile.TarFile*의 속성), 630
pax_headers (*tarfile.TarInfo*의 속성), 632
pbkdf2_hmac() (*hashlib* 모듈), 677
pd() (*turtle* 모듈), 1692
pdb
    module, 1928
Pdb (*pdb* 의 클래스), 1928
Pdb (*pdb* 클래스), 1931
pdb 명령줄 옵션
    -c, 1929
    --command, 1929
    -m, 1929
    -p, 1929
    --pid, 1929
.pdbrc
    파일, 1933
pdf() (*statistics.NormalDist* 메서드), 418
peek() (*bz2.BZ2File* 메서드), 602
peek() (*compression.zstd.ZstdFile* 메서드), 585
peek() (*gzip.GzipFile* 메서드), 599
peek() (*io.BufferedReader* 메서드), 765
peek() (*lzma.LZMAFile* 메서드), 607
peek() (*weakref.finalize* 메서드), 307
PEM_cert_to_DER_cert() (*ssl* 모듈), 1246
pen() (*turtle* 모듈), 1692
pencolor() (*turtle* 모듈), 1693
pending (*ssl.MemoryBIO*의 속성), 1274
pending() (*ssl.SSLSocket* 메서드), 1258
PendingDeprecationWarning, 120
pendown() (*turtle* 모듈), 1692
pensize() (*turtle* 모듈), 1692
penup() (*turtle* 모듈), 1692
PEP, **2314**
PERCENT (*token* 모듈), 2208
PERCENTEQUAL (*token* 모듈), 2208
perf_counter() (*time* 모듈), 775
perf_counter_ns() (*time* 모듈), 775
perm() (*math* 모듈), 352
PermissionError, 119
permutations() (*itertools* 모듈), 431
persistent_id (피클 프로토콜), 532
persistent_id() (*pickle.Pickler* 메서드), 527
persistent_load (피클 프로토콜), 532
persistent_load() (*pickle.Unpickler* 메서드), 528
PF_CAN (*socket* 모듈), 1218
PF_DIVERT (*socket* 모듈), 1219
PF_PACKET (*socket* 모듈), 1219
PF_RDS (*socket* 모듈), 1219
pformat() (*pprint* 모듈), 320
pformat() (*pprint.PrettyPrinter* 메서드), 321
pgettext() (*gettext* 모듈), 1610
pgettext() (*gettext.GNUTranslations* 메서드), 1613
pgettext() (*gettext.NullTranslations* 메서드), 1612
PGO (*test.support* 모듈), 1897
phase() (*cmath* 모듈), 361
pi (*cmath* 모듈), 363
pi (*math* 모듈), 359
pi() (*xml.etree.ElementTree.TreeBuilder* 메서드), 1423
pickle
    module, 523
pickle 명령줄 옵션
    pickle_file, 539
pickle() (*copyreg* 모듈), 540
pickle_file
    pickle 명령줄 옵션, 539
    pickletools 명령줄 옵션, 2246
PickleBuffer (*pickle* 클래스), 528
PickleError, 526
Pickler (*pickle* 클래스), 526
pickletools
    module, 2245
pickletools 명령줄 옵션
    -a, 2246
    --annotate, 2246
    --indentlevel, 2246
    -l, 2246
    -m, 2246

--memo, 2246
-o, 2246
--output, 2246
-p, 2246
pickle_file, 2246
--preamble, 2246
PicklingError, 526
--pid
pdb 명령줄 옵션, 1929
pid (*asyncio.subprocess.Process*의 속성), 1149
pid (*multiprocessing.Process*의 속성), 1021
pid (*subprocess.Popen*의 속성), 1088
PIDFD_NONBLOCK (*os* 모듈), 743
pidfd_open() (*os* 모듈), 743
pidfd_send_signal() (*signal* 모듈), 1292
PIPE (*subprocess* 모듈), 1079
Pipe() (*multiprocessing* 모듈), 1024
pipe() (*os* 모듈), 705
pipe2() (*os* 모듈), 705
PIPE_BUF (*select* 모듈), 1279
pipe_connection_lost() (*asyncio.SubprocessProtocol* 메서드), 1192
pipe_data_received() (*asyncio.SubprocessProtocol* 메서드), 1192
PIPE_MAX_SIZE (*test.support* 모듈), 1897
pipes
module, 2296
pkgutil
module, 2119
Placeholder (*functools* 모듈), 444
placeholder (*textwrap.TextWrapper*의 속성), 180
platform
module, 828
platform (*sys* 모듈), 1999
platform 명령줄 옵션
--nonaliased, 832
--terse, 832
platform() (*platform* 모듈), 829
platlibdir (*sys* 모듈), 2000
PlaySound() (*winsound* 모듈), 2258
plist
파일, 670
plistlib
module, 670
plock() (*os* 모듈), 743
PLUS (*token* 모듈), 2207
plus() (*decimal.Context* 메서드), 381
PLUSEQUAL (*token* 모듈), 2208
pm() (*pdb* 모듈), 1931
POINTER() (*ctypes* 모듈), 869
pointer() (*ctypes* 모듈), 870
polar() (*cmath* 모듈), 361
Policy (*email.policy* 클래스), 1319
poll() (*multiprocessing.connection.Connection* 메서드), 1028
poll() (*select* 모듈), 1278
poll() (*select.devpoll* 메서드), 1280
poll() (*select.epoll* 메서드), 1281
poll() (*select.poll* 메서드), 1282
poll() (*subprocess.Popen* 메서드), 1086
PollSelector (*selectors* 클래스), 1287
poly() (*turtle* 모듈), 1700
Pool (*multiprocessing.pool* 클래스), 1041
pop() (*array.array* 메서드), 303
pop() (*collections.deque* 메서드), 276
pop() (*dict* 메서드), 96
pop() (*frozenset* 메서드), 93
pop() (*mailbox.Mailbox* 메서드), 1373
pop() (*sequence* 메서드), 50
POP3
프로토콜, 1524
POP3 (*poplib* 클래스), 1524
POP3_SSL (*poplib* 클래스), 1524
pop_all() (*contextlib.ExitStack* 메서드), 2052
POP_BLOCK (*opcode*), 2244
POP_EXCEPT (*opcode*), 2232
POP_ITER (*opcode*), 2228
POP_JUMP_IF_FALSE (*opcode*), 2237
POP_JUMP_IF_NONE (*opcode*), 2238
POP_JUMP_IF_NOT_NONE (*opcode*), 2237
POP_JUMP_IF_TRUE (*opcode*), 2237
pop_source() (*shlex.shlex* 메서드), 2265
POP_TOP (*opcode*), 2229
Popen (*subprocess* 클래스), 1081
popen() (*os* 모듈), 743, 1278
popitem() (*collections.OrderedDict* 메서드), 284
popitem() (*dict* 메서드), 96
popitem() (*mailbox.Mailbox* 메서드), 1373
popleft() (*collections.deque* 메서드), 276
poplib
module, 1524
port
http.server 명령줄 옵션, 1562
port (*http.cookiejar.Cookie*의 속성), 1574
port_specified (*http.cookiejar.Cookie*의 속성), 1574
portion (포션), **2314**
pos (*json.JSONDecodeError*의 속성), 1368
pos (*re.Match*의 속성), 158
pos (*re.PatternError*의 속성), 154
pos (*tomllib.TOMLDecodeError*의 속성), 668
pos() (*operator* 모듈), 453
pos() (*turtle* 모듈), 1690
position (*xml.etree.ElementTree.ParseError*의 속성), 1426
position() (*turtle* 모듈), 1690
positional argument (위치 인자), **2314**
Positions (*dis* 클래스), 2228
positions (*inspect.FrameInfo*의 속성), 2094
positions (*inspect.Traceback*의 속성), 2095
Positions.col_offset (*dis* 모듈), 2228
Positions.end_col_offset (*dis* 모듈), 2228
Positions.end_lineno (*dis* 모듈), 2228
Positions.lineno (*dis* 모듈), 2228
POSIX
I/O 제어, 2271
스레드, 1105

posix
    module, 2268
POSIX 공유 메모리, 1057
POSIX_FADV_DONTNEED (os 모듈), 705
POSIX_FADV_NOREUSE (os 모듈), 705
POSIX_FADV_NORMAL (os 모듈), 705
POSIX_FADV_RANDOM (os 모듈), 705
POSIX_FADV_SEQUENTIAL (os 모듈), 705
POSIX_FADV_WILLNEED (os 모듈), 705
posix_fadvise() (os 모듈), 705
posix_fallocate() (os 모듈), 705
posix_openpt() (os 모듈), 706
posix_spawn() (os 모듈), 744
POSIX_SPAWN_CLOSE (os 모듈), 744
POSIX_SPAWN_CLOSEFROM (os 모듈), 744
POSIX_SPAWN_DUP2 (os 모듈), 744
POSIX_SPAWN_OPEN (os 모듈), 744
posix_spawnp() (os 모듈), 745
PosixPath (pathlib 클래스), 471
post_handshake_auth (ssl.SSLContext 의 속성), 1265
post_mortem() (pdb 모듈), 1931
post_setup() (venv.EnvBuilder 메서드), 1972
postcmd() (cmd.Cmd 메서드), 990
postloop() (cmd.Cmd 메서드), 990
Pow (ast 클래스), 2171
pow()
    built-in function, 25
pow() (math 모듈), 356
pow() (operator 모듈), 453
power() (decimal.Context 메서드), 381
pp (pdb command), 1935
pp() (pprint 모듈), 319
pprint
    module, 319
pprint() (pprint 모듈), 320
pprint() (pprint.PrettyPrinter 메서드), 321
prcal() (calendar 모듈), 265
pread() (os 모듈), 706
preadv() (os 모듈), 706
--preamble
    pickletools 명령줄 옵션, 2246
preamble (email.message.EmailMessage의 속성), 1311
preamble (email.message.Message의 속성), 1348
prec (decimal.Context의 속성), 377
precmd() (cmd.Cmd 메서드), 989
prefix (sys 모듈), 2000
prefix (xml.dom.Attr의 속성), 1433
prefix (xml.dom.Node의 속성), 1429
prefix (zipimport.zipimporter의 속성), 2119
PREFIXES (site 모듈), 2110
prefixlen (ipaddress.IPv4Network의 속성), 1597
prefixlen (ipaddress.IPv6Network의 속성), 1599
preloop() (cmd.Cmd 메서드), 990
prepare() (graphlib.TopologicalSorter 메서드), 344
prepare() (logging.handlers.QueueHandler 메서드), 826
prepare() (logging.handlers.QueueListener 메서드), 827
prepare_class() (types 모듈), 312
prepare_input_source() (xml.sax.saxutils 모듈), 1450
prepare_main() (concurrent.interpreters.Interpreter 메서드), 1075
PrepareProtocol (sqlite3 클래스), 571
PrettyPrinter (pprint 클래스), 320
prev() (tkinter.ttk.Treeview 메서드), 1660
previousSibling (xml.dom.Node의 속성), 1429
print()
    built-in function, 26
print() (traceback.TracebackException 메서드), 2069
print_call_graph() (asyncio 모듈), 1154
print_callees() (pstats.Stats 메서드), 1945
print_callers() (pstats.Stats 메서드), 1944
print_exc() (timeit.Timer 메서드), 1949
print_exc() (traceback 모듈), 2066
print_exception() (traceback 모듈), 2066
print_function (__future__ 모듈), 2076
print_help() (argparse.ArgumentParser 메서드), 910
print_last() (traceback 모듈), 2066
print_list() (traceback 모듈), 2066
print_stack() (asyncio.Task 메서드), 1130
print_stack() (traceback 모듈), 2066
print_stats() (profile.Profile 메서드), 1942
print_stats() (pstats.Stats 메서드), 1944
print_tb() (traceback 모듈), 2065
print_usage() (argparse.ArgumentParser 메서드), 910
print_usage() (optparse.OptionParser 메서드), 946
print_version() (optparse.OptionParser 메서드), 936
print_warning() (test.support 모듈), 1901
printable (string 모듈), 126
printdir() (zipfile.ZipFile 메서드), 616
printf-style formatting, 66, 81
PRIO_DARWIN_BG (os 모듈), 695
PRIO_DARWIN_NONUI (os 모듈), 695
PRIO_DARWIN_PROCESS (os 모듈), 695
PRIO_DARWIN_THREAD (os 모듈), 695
PRIO_PGRP (os 모듈), 694
PRIO_PROCESS (os 모듈), 694
PRIO_USER (os 모듈), 694
PriorityQueue (asyncio 클래스), 1152
PriorityQueue (queue 클래스), 1098
prlimit() (resource 모듈), 2279
prmonth() (calendar 모듈), 265
prmonth() (calendar.TextCalendar 메서드), 263
ProactorEventLoop (asyncio 클래스), 1178
process
    group, 693, 694
    id, 694
    id of parent, 694
    killing, 742, 743
    scheduling priority, 694, 697

signalling, 742, 743
--process
timeit 명령줄 옵션, 1949
Process (*multiprocessing* 클래스), 1019
process() (*logging.LoggerAdapter* 메서드), 796
process_cpu_count() (*os* 모듈), 754
process_exited() (*asyncio.SubprocessProtocol* 메서드), 1192
process_request() (*socketserver.BaseServer* 메서드), 1551
process_time() (*time* 모듈), 775
process_time_ns() (*time* 모듈), 775
process_tokens() (*tabnanny* 모듈), 2215
ProcessError, 1023
ProcessingInstruction() (*xml.etree.ElementTree* 모듈), 1416
processingInstruction() (*xml.sax.handler.ContentHandler* 메서드), 1448
ProcessingInstructionHandler() (*xml.parsers.expat.xmlparser* 메서드), 1458
ProcessLookupError, 119
processor() (*platform* 모듈), 829
ProcessPoolExecutor (*concurrent.futures* 클래스), 1068
prod() (*math* 모듈), 357
product() (*itertools* 모듈), 431
profile
module, 1941
Profile (*profile* 클래스), 1941
ProgrammingError, 572
Progressbar (*tkinter.ttk* 클래스), 1656
--prompt
venv 명령줄 옵션, 1968
prompt (*cmd.Cmd*의 속성), 990
propagate (*logging.Logger*의 속성), 785
property (내장 클래스), 26
property() (*enum* 모듈), 342
property_declaration_handler (*xml.sax.handler* 모듈), 1446
property_dom_node (*xml.sax.handler* 모듈), 1446
property_lexical_handler (*xml.sax.handler* 모듈), 1445
property_xml_string (*xml.sax.handler* 모듈), 1446
property.deleter()
built-in function, 27
property.getter()
built-in function, 27
PropertyMock (*unittest.mock* 클래스), 1843
property.setter()
built-in function, 27
prot_c() (*ftplib.FTP_TLS* 메서드), 1523
prot_p() (*ftplib.FTP_TLS* 메서드), 1523
proto (*socket.socket*의 속성), 1237
protocol
context management, 98
iterator, 46
--protocol
http.server 명령줄 옵션, 1562
Protocol (*asyncio* 클래스), 1189
protocol (*ssl.SSLContext*의 속성), 1265
Protocol (*typing* 클래스), 1752
PROTOCOL_SSLv3 (*ssl* 모듈), 1249
PROTOCOL_SSLv23 (*ssl* 모듈), 1249
PROTOCOL_TLS (*ssl* 모듈), 1248
PROTOCOL_TLS_CLIENT (*ssl* 모듈), 1249
PROTOCOL_TLS_SERVER (*ssl* 모듈), 1249
PROTOCOL_TLSv1 (*ssl* 모듈), 1249
PROTOCOL_TLSv1_1 (*ssl* 모듈), 1249
PROTOCOL_TLSv1_2 (*ssl* 모듈), 1249
protocol_version (*http.server.BaseHTTPRequestHandler*의 속성), 1558
PROTOCOL_VERSION (*imaplib.IMAP4*의 속성), 1534
ProtocolError (*xmlrpc.client* 클래스), 1581
provisional API (잠정 *API*), **2314**
provisional package (잠정 패키지), **2314**
proxy() (*weakref* 모듈), 305
proxyauth() (*imaplib.IMAP4* 메서드), 1532
ProxyBasicAuthHandler (*urllib.request* 클래스), 1483
ProxyDigestAuthHandler (*urllib.request* 클래스), 1483
ProxyHandler (*urllib.request* 클래스), 1482
ProxyType (*weakref* 모듈), 307
ProxyTypes (*weakref* 모듈), 308
pryear() (*calendar.TextCalendar* 메서드), 263
ps1 (*sys* 모듈), 2001
ps2 (*sys* 모듈), 2001
pstats
module, 1943
pstdev() (*statistics* 모듈), 413
pthread_getcpuclockid() (*time* 모듈), 772
pthread_kill() (*signal* 모듈), 1292
pthread_sigmask() (*signal* 모듈), 1293
pthreads, 1105
pthreads (*sys._emscripten_info*의 속성), 1985
ptsname() (*os* 모듈), 707
pty
module, 705, 2273
pu() (*turtle* 모듈), 1692
publicId (*xml.dom.DocumentType*의 속성), 1431
PullDom (*xml.dom.pulldom* 클래스), 1441
punctuation (*string* 모듈), 126
punctuation_chars (*shlex.shlex*의 속성), 2267
PurePath (*pathlib* 클래스), 461
PurePosixPath (*pathlib* 클래스), 462
PureWindowsPath (*pathlib* 클래스), 462
purge() (*re* 모듈), 154
Purpose.CLIENT_AUTH (*ssl* 모듈), 1253
Purpose.SERVER_AUTH (*ssl* 모듈), 1253
push() (*code.InteractiveConsole* 메서드), 2115
push() (*contextlib.ExitStack* 메서드), 2052
push_async_callback() (*contextlib.AsyncExitStack* 메서드), 2053
push_async_exit() (*contextlib.AsyncExitStack* 메서드), 2052

PUSH_EXC_INFO (*opcode*), 2232
PUSH_NULL (*opcode*), 2240
push_source() (*shlex.shlex* 메서드), 2265
push_token() (*shlex.shlex* 메서드), 2265
put() (*asyncio.Queue* 메서드), 1151
put() (*multiprocessing.Queue* 메서드), 1024
put() (*multiprocessing.SimpleQueue* 메서드), 1026
put() (*queue.Queue* 메서드), 1099
put() (*queue.SimpleQueue* 메서드), 1100
put_nowait() (*asyncio.Queue* 메서드), 1151
put_nowait() (*multiprocessing.Queue* 메서드), 1025
put_nowait() (*queue.Queue* 메서드), 1099
put_nowait() (*queue.SimpleQueue* 메서드), 1101
putch() (*msvcrt* 모듈), 2248
putenv() (*os* 모듈), 695
putheader() (*http.client.HTTPConnection* 메서드), 1514
putp() (*curses* 모듈), 962
putrequest() (*http.client.HTTPConnection* 메서드), 1514
putwch() (*msvcrt* 모듈), 2248
putwin() (*curses.window* 메서드), 969
pvariance() (*statistics* 모듈), 413
pwd
    module, 2269
    모듈, 488
pwd() (*ftplib.FTP* 메서드), 1522
pwrite() (*os* 모듈), 707
pwritev() (*os* 모듈), 707
py_compile
    module, 2217
Py_DEBUG (*test.support* 모듈), 1897
py_object (*ctypes* 클래스), 874
PY_RESUME (*monitoring event*), 2010
PY_RETURN (*monitoring event*), 2010
PY_START (*monitoring event*), 2010
PY_THROW (*monitoring event*), 2010
PY_UNWIND (*monitoring event*), 2010
PY_YIELD (*monitoring event*), 2010
pycache_prefix (*sys* 모듈), 1985
PyCF_ALLOW_TOP_LEVEL_AWAIT (*ast* 모듈), 2199
PyCF_ONLY_AST (*ast* 모듈), 2199
PyCF_OPTIMIZED_AST (*ast* 모듈), 2199
PyCF_TYPE_COMMENTS (*ast* 모듈), 2199
PycInvalidationMode (*py_compile* 클래스), 2217
pyclbr
    module, 2215
PyCompileError, 2217
PyDLL (*ctypes* 클래스), 862
pydoc
    module, 1773
pyexpat
    모듈, 1454
PYFUNCTYPE() (*ctypes* 모듈), 865
--python
    zipapp 명령줄 옵션, 1977
Python 3000 (파이썬 *3000*), **2314**
Python 향상 제안
PEP 1, 2314
PEP 8, 29
PEP 205, 308
PEP 227, 2076
PEP 235, 2126
PEP 236, 2077
PEP 237, 67, 83
PEP 238, 2076, 2307
PEP 246, 571
PEP 249, 551, 554, 566, 567, 572, 575, 581
PEP 0249#threadsafety, 556
PEP 255, 2076
PEP 263, 2126, 2211
PEP 273, 2117
PEP 278, 2317
PEP 282, 518, 802
PEP 292, 134
PEP 302, 33, 510, 1999, 2117, 2120, 2121, 2123, 2126, 21292131, 2311
PEP 305, 641
PEP 307, 525
PEP 324, 1077
PEP 328, 33, 2076, 2126
PEP 338, 2125
PEP 342, 290
PEP 343, 2056, 2076, 2304
PEP 362, 2092, 2302, 2313
PEP 366, 2125, 2126
PEP 370, 2111
PEP 378, 130
PEP 380#use-of-stopiteration-to-return-values, 2011
PEP 383, 202, 1213
PEP 387, 120
PEP 393, 208, 1998
PEP 405, 1967
PEP 411, 1994, 2003, 2004, 2314
PEP 412, 441
PEP 420, 2126, 2312, 2314
PEP 421, 1996
PEP 428, 460
PEP 434, 1677
PEP 442, 2080
PEP 443, 2308
PEP 451, 1999, 21242126
PEP 453, 1965
PEP 461, 83
PEP 468, 284
PEP 475, 25, 119, 704, 708, 711, 749, 763, 776, 1230, 1231, 12331236, 12791282, 1286, 1295
PEP 479, 116, 2076
PEP 483, 2308
PEP 484, 103, 1719, 1729, 1743, 1761, 2166, 2195, 2199, 2301, 2307, 2308, 2317, 2318
PEP 485, 354, 363
PEP 488, 1911, 2126, 2140, 2141, 2217
PEP 489, 2126, 2135, 2138, 2142

PEP 492, 292, 2099, 2302, 2303, 2305
PEP 495, 256
PEP 498, 2306
PEP 506, 687
PEP 515, 130, 394
PEP 519, 2314
PEP 524, 756
PEP 525, 292, 1994, 2003, 2099, 2302
PEP 526, 1735, 1751, 2033, 2040, 2101, 2195, 2199, 2301, 2318
PEP 529, 715, 1992, 2005
PEP 538, 1623
PEP 540, 690, 1623
PEP 544, 1729, 1752
PEP 552, 2126, 2217
PEP 554, 1073
PEP 557, 2033
PEP 560, 312, 313
PEP 563, 2076, 2101
PEP 565, 120
PEP 566, 2153
PEP 567, 1101, 1159, 1160, 1183
PEP 574, 525, 537
PEP 578, 1915, 1982
PEP 584, 270, 279, 284, 306, 316, 692
PEP 585, 103, 288, 315, 17651772, 2308
PEP 586, 1735
PEP 591, 1736, 1762
PEP 593, 1739, 1764
PEP 594, 2289, 22932297
PEP 597, 758
PEP 604, 105
PEP 610, 2156
PEP 612, 1721, 1727, 1734, 1748, 1771
PEP 613, 1732
PEP 615, 256
PEP 626, 2227
PEP 632, 2294
PEP 644, 1242
PEP 646, 1746
PEP 647, 1740
PEP 649, 1766, 2076, 2101, 2102, 2301
PEP 649#the-stringizer-and-the-fake-globals, 2103
PEP 655, 1736, 1756
PEP 667, 21, 1930
PEP 673, 1732
PEP 675, 1730
PEP 681, 1761
PEP 682, 129
PEP 683, 2309
PEP 684, 1073, 1996
PEP 686, 691, 758
PEP 688, 292
PEP 688#current-options, 291, 1769
PEP 692, 1741
PEP 695, 1730, 1743, 1744, 1746, 1748, 1772
PEP 698, 1763
PEP 702, 2032
PEP 703, 996, 2307, 2308
PEP 705, 1736
PEP 706, 633
PEP 709, 21
PEP 734, 1067, 1073, 1996
PEP 742, 1740
PEP 749, 2076, 2101
PEP 750, 137
PEP 3101, 126
PEP 3105, 2076
PEP 3107, 2101
PEP 3112, 2076
PEP 3115, 312, 2194
PEP 3116, 2317
PEP 3118, 85, 525
PEP 3119, 293, 2058
PEP 3120, 2126
PEP 3134, 112
PEP 3141, 347, 2058
PEP 3147, 1911, 2124, 2126, 2140, 2141, 2217, 22192221
PEP 3148, 1071
PEP 3149, 1981
PEP 3151, 120, 1216, 1277, 2278
PEP 3154, 525
PEP 3155, 2314
PEP 3333, 14681472, 1475, 1476
PEP 3333#input-and-error-streams, 1477
PEP 3333#optional-platform-specific-file-handling, 1477
PEP 3333#the-start-response-callable, 1476
python_branch() (*platform* 모듈), 829
python_build() (*platform* 모듈), 829
python_compiler() (*platform* 모듈), 829
PYTHON_CONTEXT_AWARE_WARNINGS, 1988, 2032
PYTHON_CPU_COUNT, 754, 1026
PYTHON_DOM, 1427
PYTHON_GIL, 1988, 2308
python_implementation() (*platform* 모듈), 829
python_is_optimized() (*test.support* 모듈), 1899
PYTHON_JIT, 1997
python_revision() (*platform* 모듈), 829
PYTHON_THREAD_INHERIT_CONTEXT, 1988
python_version() (*platform* 모듈), 829
python_version_tuple() (*platform* 모듈), 829
PYTHONASYNCIODEBUG, 1173, 1209, 1775
PYTHONBREAKPOINT, 10, 1984
PYTHONCASEOK, 34
PYTHONCOERCECLOCALE, 691
PYTHONDEVMODE, 1774
PYTHONDONTWRITEBYTECODE, 1985
PYTHONFAULTHANDLER, 1774, 1925
PythonFinalizationError, 115
PYTHONHOME, 1906, 2158, 2159
Pythonic (파이썬다운), **2314**
PYTHONINTMAXSTRDIGITS, 109, 1996

PYTHONIOENCODING, 690, 2005
PYTHONLEGACYWINDOWSFSENCODING, 2005
PYTHONLEGACYWINDOWSSTDIO, 2005
PYTHONMALLOC, 1774
python--m-py_compile 명령줄 옵션
    -, 2218
    <file>, 2218
    -q, 2218
    --quiet, 2218
python--m-sqlite3-[-h]-[-v]-[filename]-[sql] 명령줄 옵션
    -h, 574
    --help, 574
    -v, 574
    --version, 574
PYTHONNOUSERSITE, 2110
PYTHONPATH, 1906, 1999, 2158
PYTHONPLATLIBDIR, 2158
PYTHONPYCACHEPREFIX, 1985
PYTHONSAFEPATH, 1999, 2299
PYTHONSTARTUP, 187, 1110, 1673, 1674, 1996, 2110
PYTHONTRACEMALLOC, 1954, 1960
PYTHONTZPATH, 261
PYTHONUNBUFFERED, 2005
PYTHONUSERBASE, 2111
PYTHONUSERSITE, 1906
PYTHONUTF8, 691, 2005
PYTHONWARNDEFAULTENCODING, 758
PYTHONWARNINGS, 1774, 2026, 2027
PyZipFile (*zipfile* 클래스), 619

Q

-q
    compileall 명령줄 옵션, 2219
    python--m-py_compile 명령줄 옵션, 2218
qiflush() (*curses* 모듈), 962
QName (*xml.etree.ElementTree* 클래스), 1423
qsize() (*asyncio.Queue* 메서드), 1151
qsize() (*multiprocessing.Queue* 메서드), 1024
qsize() (*queue.Queue* 메서드), 1099
qsize() (*queue.SimpleQueue* 메서드), 1100
qualified name (정규화된 이름), **2314**
quantiles() (*statistics* 모듈), 415
quantiles() (*statistics.NormalDist* 메서드), 418
quantize() (*decimal.Context* 메서드), 382
quantize() (*decimal.Decimal* 메서드), 374
QueryInfoKey() (*winreg* 모듈), 2253
QueryReflectionKey() (*winreg* 모듈), 2255
QueryValue() (*winreg* 모듈), 2253
QueryValueEx() (*winreg* 모듈), 2254
QUESTION (*tkinter.messagebox* 모듈), 1646
queue
    module, 1098
Queue (*asyncio* 클래스), 1151
Queue (*concurrent.interpreters* 클래스), 1076
Queue (*multiprocessing* 클래스), 1024
Queue (*queue* 클래스), 1098
queue (*sched.scheduler*의 속성), 1097
Queue() (*multiprocessing.managers.SyncManager* 메서드), 1037
QueueEmpty, 1152
QueueEmptyError, 1076
QueueFull, 1152
QueueFullError, 1076
QueueHandler (*logging.handlers* 클래스), 826
QueueListener (*logging.handlers* 클래스), 827
QueueShutDown, 1152
quick_ratio() (*difflib.SequenceMatcher* 메서드), 171
--quiet
    python--m-py_compile 명령줄 옵션, 2218
quiet (*sys.flags*의 속성), 1988
quit (*pdb command*), 1937
quit (내장 변수), 36
quit() (*ftplib.FTP* 메서드), 1522
quit() (*poplib.POP3* 메서드), 1526
quit() (*smtplib.SMTP* 메서드), 1540
quit() (*tkinter.filedialog.FileDialog* 메서드), 1644
quitting (*bdb.Bdb* 어트리뷰트), 1923
quopri
    module, 1400
quote() (*email.utils* 모듈), 1357
quote() (*shlex* 모듈), 2263
quote() (*urllib.parse* 모듈), 1502
QUOTE_ALL (*csv* 모듈), 645
quote_from_bytes() (*urllib.parse* 모듈), 1503
QUOTE_MINIMAL (*csv* 모듈), 645
QUOTE_NONE (*csv* 모듈), 645
QUOTE_NONNUMERIC (*csv* 모듈), 645
QUOTE_NOTNULL (*csv* 모듈), 645
quote_plus() (*urllib.parse* 모듈), 1503
QUOTE_STRINGS (*csv* 모듈), 645
quoteattr() (*xml.sax.saxutils* 모듈), 1450
quotechar (*csv.Dialect*의 속성), 646
quoted-printable
    인코딩, 1400
quotes (*shlex.shlex*의 속성), 2266
quoting (*csv.Dialect*의 속성), 646

R

-R
    trace 명령줄 옵션, 1953
-r
    compileall 명령줄 옵션, 2219
    idle 명령줄 옵션, 1674
    timeit 명령줄 옵션, 1949
    trace 명령줄 옵션, 1952
R_OK (*os* 모듈), 713
radians() (*math* 모듈), 357
radians() (*turtle* 모듈), 1691
radix (*sys.float_info*의 속성), 1990
radix() (*decimal.Context* 메서드), 382
radix() (*decimal.Decimal* 메서드), 374
RADIXCHAR (*locale* 모듈), 1620
raise
    문장, 111

Raise (*ast* 클래스), 2177
RAISE (*monitoring event*), 2010
raise_on_defect (*email.policy.Policy*의 속성), 1320
raise_signal() (*signal* 모듈), 1292
RAISE_VARARGS (*opcode*), 2239
raiseExceptions (*logging* 모듈), 801
RAND_add() (*ssl* 모듈), 1245
RAND_bytes() (*ssl* 모듈), 1245
RAND_status() (*ssl* 모듈), 1245
randbelow() (*secrets* 모듈), 687
randbits() (*secrets* 모듈), 687
randbytes() (*random* 모듈), 398
randbytes() (*random.Random* 메서드), 401
randint() (*random* 모듈), 398
random
    module, 397
Random (*random* 클래스), 401
random 명령줄 옵션
    -c, 406
    --choice, 406
    -f, 406
    --float, 406
    -h, 405
    --help, 405
    -i, 406
    --integer, 406
random() (*random* 모듈), 400
random() (*random.Random* 메서드), 401
randrange() (*random* 모듈), 398
range
    object, 51
range (내장 클래스), 51
RARROW (*token* 모듈), 2209
ratio() (*difflib.SequenceMatcher* 메서드), 171
Rational (*numbers* 클래스), 348
raw (*io.BufferedIOBase*의 속성), 763
raw() (*curses* 모듈), 962
raw() (*pickle.PickleBuffer* 메서드), 528
raw_data_manager (*email.contentmanager* 모듈), 1333
raw_decode() (*json.JSONDecoder* 메서드), 1366
raw_input() (*code.InteractiveConsole* 메서드), 2115
RawArray() (*multiprocessing.sharedctypes* 모듈), 1033
RawConfigParser (*configparser* 클래스), 665
RawDescriptionHelpFormatter (*argparse* 클래스), 885
RawIOBase (*io* 클래스), 762
RawPen (*turtle* 클래스), 1709
RawTextHelpFormatter (*argparse* 클래스), 885
RawTurtle (*turtle* 클래스), 1709
RawValue() (*multiprocessing.sharedctypes* 모듈), 1033
RBRACE (*token* 모듈), 2208
re
    module, 55, 142
    모듈, 509
re (*re.Match*의 속성), 159
READ (*inspect.BufferFlags*의 속성), 2100
read() (*asyncio.StreamReader* 메서드), 1135
read() (*codecs.StreamReader* 메서드), 206
read() (*configparser.ConfigParser* 메서드), 663
read() (*http.client.HTTPResponse* 메서드), 1515
read() (*imaplib.IMAP4* 메서드), 1532
read() (*io.BufferedIOBase* 메서드), 763
read() (*io.BufferedReader* 메서드), 765
read() (*io.RawIOBase* 메서드), 762
read() (*io.Reader* 메서드), 770
read() (*io.TextIOBase* 메서드), 767
read() (*mimetypes.MimeTypes* 메서드), 1392
read() (*mmap.mmap* 메서드), 1300
read() (*os* 모듈), 708
read() (*sqlite3.Blob* 메서드), 571
read() (*ssl.MemoryBIO* 메서드), 1274
read() (*ssl.SSLSocket* 메서드), 1255
read() (*urllib.robotparser.RobotFileParser* 메서드), 1505
read() (*zipfile.ZipFile* 메서드), 616
read1() (*bz2.BZ2File* 메서드), 603
read1() (*io.BufferedIOBase* 메서드), 763
read1() (*io.BufferedReader* 메서드), 765
read1() (*io.BytesIO* 메서드), 765
read_binary() (*importlib.resources* 모듈), 2147
read_byte() (*mmap.mmap* 메서드), 1300
read_bytes() (*importlib.abc.Traversable* 메서드), 2134
read_bytes() (*importlib.resources.abc.Traversable* 메서드), 2150
read_bytes() (*pathlib.Path* 메서드), 476
read_bytes() (*zipfile.Path* 메서드), 619
read_dict() (*configparser.ConfigParser* 메서드), 664
read_environ() (*wsgiref.handlers* 모듈), 1476
read_events() (*xml.etree.ElementTree.XMLPullParser* 메서드), 1426
read_file() (*configparser.ConfigParser* 메서드), 663
read_history_file() (*readline* 모듈), 185
read_init_file() (*readline* 모듈), 185
read_mime_types() (*mimetypes* 모듈), 1391
read_string() (*configparser.ConfigParser* 메서드), 663
read_text() (*importlib.abc.Traversable* 메서드), 2134
read_text() (*importlib.resources* 모듈), 2147
read_text() (*importlib.resources.abc.Traversable* 메서드), 2150
read_text() (*pathlib.Path* 메서드), 476
read_text() (*zipfile.Path* 메서드), 619
read_token() (*shlex.shlex* 메서드), 2265
read_windows_registry() (*mimetypes.MimeTypes* 메서드), 1392
READABLE (*_tkinter* 모듈), 1640
readable() (*bz2.BZ2File* 메서드), 602
readable() (*io.IOBase* 메서드), 761
readall() (*io.RawIOBase* 메서드), 762
readbuffer_encode() (*codecs* 모듈), 201
Reader (*io* 클래스), 770
reader() (*csv* 모듈), 641
ReadError, 625

readexactly() (*asyncio.StreamReader* 메서드), 1135
readfp() (*mimetypes.MimeTypes* 메서드), 1392
readframes() (*wave.Wave_read* 메서드), 1606
readinto() (*bz2.BZ2File* 메서드), 603
readinto() (*http.client.HTTPResponse* 메서드), 1515
readinto() (*io.BufferedIOBase* 메서드), 763
readinto() (*io.RawIOBase* 메서드), 762
readinto() (*os* 모듈), 708
readinto1() (*io.BufferedIOBase* 메서드), 763
readinto1() (*io.BytesIO* 메서드), 765
readline
module, 184
readline() (*asyncio.StreamReader* 메서드), 1135
readline() (*codecs.StreamReader* 메서드), 207
readline() (*imaplib.IMAP4* 메서드), 1532
readline() (*io.IOBase* 메서드), 761
readline() (*io.TextIOBase* 메서드), 767
readline() (*mmap.mmap* 메서드), 1300
readlines() (*codecs.StreamReader* 메서드), 207
readlines() (*io.IOBase* 메서드), 761
readlink() (*os* 모듈), 719
readlink() (*pathlib.Path* 메서드), 473
readmodule() (*pyclbr* 모듈), 2215
readmodule_ex() (*pyclbr* 모듈), 2215
readonly (*memoryview*의 속성), 90
ReadOnly (*typing* 모듈), 1736
ReadTransport (*asyncio* 클래스), 1185
readuntil() (*asyncio.StreamReader* 메서드), 1135
readv() (*os* 모듈), 710
ready() (*multiprocessing.pool.AsyncResult* 메서드), 1043
Real (*numbers* 클래스), 347
real (*numbers.Complex*의 속성), 347
real_max_memuse (*test.support* 모듈), 1897
real_quick_ratio() (*difflib.SequenceMatcher* 메서드), 171
realpath() (*os.path* 모듈), 490
REALTIME_PRIORITY_CLASS (*subprocess* 모듈), 1090
reap_children() (*test.support* 모듈), 1903
reap_threads() (*test.support.threading_helper* 모듈), 1907
reason (*http.client.HTTPResponse*의 속성), 1515
reason (*ssl.SSLError*의 속성), 1244
reason (*UnicodeError*의 속성), 118
reason (*urllib.error.HTTPError*의 속성), 1505
reason (*urllib.error.URLError*의 속성), 1504
reattach() (*tkinter.ttk.Treeview* 메서드), 1660
recent() (*imaplib.IMAP4* 메서드), 1532
reconfigure() (*io.TextIOWrapper* 메서드), 768
record_original_stdout() (*test.support* 모듈), 1899
RECORDS (*inspect.BufferFlags*의 속성), 2100
records (*unittest.TestCase*의 속성), 1814
RECORDS_RO (*inspect.BufferFlags*의 속성), 2100
rect() (*cmath* 모듈), 361
rectangle() (*curses.textpad* 모듈), 982
RecursionError, 116
recursive_repr() (*reprlib* 모듈), 325
recv() (*multiprocessing.connection.Connection* 메서드), 1028
recv() (*socket.socket* 메서드), 1233
recv_bytes() (*multiprocessing.connection.Connection* 메서드), 1029
recv_bytes_into() (*multiprocessing.connection.Connection* 메서드), 1029
recv_fds() (*socket* 모듈), 1230
recv_into() (*socket.socket* 메서드), 1234
recvfrom() (*socket.socket* 메서드), 1233
recvfrom_into() (*socket.socket* 메서드), 1234
recvmsg() (*socket.socket* 메서드), 1233
recvmsg_into() (*socket.socket* 메서드), 1234
redirect_request() (*urllib.request.HTTPRedirectHandler* 메서드), 1488
redirect_stderr() (*contextlib* 모듈), 2049
redirect_stdout() (*contextlib* 모듈), 2048
redisplay() (*readline* 모듈), 185
redrawln() (*curses.window* 메서드), 969
redrawwin() (*curses.window* 메서드), 969
reduce() (*functools* 모듈), 446
reducer_override() (*pickle.Pickler* 메서드), 527
ref (*weakref* 클래스), 305
refcount_test() (*test.support* 모듈), 1902
reference count (참조 횟수), **2315**
ReferenceError, 116
ReferenceType (*weakref* 모듈), 307
refold_source (*email.policy.EmailPolicy*의 속성), 1322
refresh() (*curses.window* 메서드), 969
REG_BINARY (*winreg* 모듈), 2257
REG_DWORD (*winreg* 모듈), 2257
REG_DWORD_BIG_ENDIAN (*winreg* 모듈), 2257
REG_DWORD_LITTLE_ENDIAN (*winreg* 모듈), 2257
REG_EXPAND_SZ (*winreg* 모듈), 2257
REG_FULL_RESOURCE_DESCRIPTOR (*winreg* 모듈), 2257
REG_LINK (*winreg* 모듈), 2257
REG_MULTI_SZ (*winreg* 모듈), 2257
REG_NONE (*winreg* 모듈), 2257
REG_QWORD (*winreg* 모듈), 2257
REG_QWORD_LITTLE_ENDIAN (*winreg* 모듈), 2257
REG_RESOURCE_LIST (*winreg* 모듈), 2257
REG_RESOURCE_REQUIREMENTS_LIST (*winreg* 모듈), 2257
REG_SZ (*winreg* 모듈), 2257
RegexFlag (*re* 클래스), 149
register() (*abc.ABCMeta* 메서드), 2059
register() (*argparse.ArgumentParser* 메서드), 912
register() (*atexit* 모듈), 2063
register() (*codecs* 모듈), 200
register() (*faulthandler* 모듈), 1927
register() (*functools.singledispatch* 메서드), 446
register() (*multiprocessing.managers.BaseManager* 메서드), 1036
register() (*select.devpoll* 메서드), 1279
register() (*select.epoll* 메서드), 1281

register() (*selectors.BaseSelector* 메서드), 1285
register() (*select.poll* 메서드), 1281
register() (*webbrowser* 모듈), 1466
register_adapter() (*sqlite3* 모듈), 555
register_archive_format() (*shutil* 모듈), 519
register_at_fork() (*os* 모듈), 745
register_callback() (*sys.monitoring* 모듈), 2012
register_converter() (*sqlite3* 모듈), 555
register_defect() (*email.policy.Policy* 메서드), 1321
register_dialect() (*csv* 모듈), 642
register_error() (*codecs* 모듈), 203
register_function() (*xmlrpc.server.CGIXMLRPCRequestHandler* 메서드), 1587
register_function() (*xmlrpc.server.SimpleXMLRPCServer* 메서드), 1584
register_instance() (*xmlrpc.server.CGIXMLRPCRequestHandler* 메서드), 1587
register_instance() (*xmlrpc.server.SimpleXMLRPCServer* 메서드), 1584
register_introspection_functions() (*xmlrpc.server.CGIXMLRPCRequestHandler* 메서드), 1588
register_introspection_functions() (*xmlrpc.server.SimpleXMLRPCServer* 메서드), 1585
register_multicall_functions() (*xmlrpc.server.CGIXMLRPCRequestHandler* 메서드), 1588
register_multicall_functions() (*xmlrpc.server.SimpleXMLRPCServer* 메서드), 1585
register_namespace() (*xml.etree.ElementTree* 모듈), 1416
register_optionflag() (*doctest* 모듈), 1786
register_shape() (*turtle* 모듈), 1707
register_unpack_format() (*shutil* 모듈), 519
registerDOMImplementation() (*xml.dom* 모듈), 1427
registerResult() (*unittest* 모듈), 1831
REGTYPE (*tarfile* 모듈), 626
regular package (정규 패키지), **2315**
relative_to() (*pathlib.PurePath* 메서드), 469
release() (*_thread.lock* 메서드), 1107
release() (*asyncio.Condition* 메서드), 1143
release() (*asyncio.Lock* 메서드), 1141
release() (*asyncio.Semaphore* 메서드), 1144
release() (*logging.Handler* 메서드), 790
release() (*memoryview* 메서드), 87
release() (*multiprocessing.Lock* 메서드), 1031
release() (*multiprocessing.RLock* 메서드), 1031
release() (*pickle.PickleBuffer* 메서드), 528
release() (*platform* 모듈), 830
release() (*threading.Condition* 메서드), 1007
release() (*threading.Lock* 메서드), 1004
release() (*threading.RLock* 메서드), 1006
release() (*threading.Semaphore* 메서드), 1009
reload() (*importlib* 모듈), 2127
reload_environ() (*os* 모듈), 692
relpath() (*os.path* 모듈), 491
remainder() (*decimal.Context* 메서드), 382
remainder() (*math* 모듈), 353
remainder_near() (*decimal.Context* 메서드), 382
remainder_near() (*decimal.Decimal* 메서드), 374
remote_exec() (*sys* 모듈), 2004
RemoteDisconnected, 1512
remove() (*array.array* 메서드), 303
remove() (*collections.deque* 메서드), 276
remove() (*frozenset* 메서드), 93
remove() (*mailbox.Mailbox* 메서드), 1371
remove() (*mailbox.MH* 메서드), 1378
remove() (*os* 모듈), 720
remove() (*sequence* 메서드), 50
remove() (*xml.etree.ElementTree.Element* 메서드), 1420
remove_done_callback() (*asyncio.Future* 메서드), 1183
remove_done_callback() (*asyncio.Task* 메서드), 1130
remove_flag() (*mailbox.Maildir* 메서드), 1375
remove_flag() (*mailbox.MaildirMessage* 메서드), 1382
remove_flag() (*mailbox.mboxMessage* 메서드), 1383
remove_flag() (*mailbox.MMDFMessage* 메서드), 1387
remove_folder() (*mailbox.Maildir* 메서드), 1374
remove_folder() (*mailbox.MH* 메서드), 1378
remove_header() (*urllib.request.Request* 메서드), 1485
remove_history_item() (*readline* 모듈), 186
remove_label() (*mailbox.BabylMessage* 메서드), 1385
remove_option() (*configparser.ConfigParser* 메서드), 665
remove_option() (*optparse.OptionParser* 메서드), 945
remove_reader() (*asyncio.loop* 메서드), 1167
remove_section() (*configparser.ConfigParser* 메서드), 665
remove_sequence() (*mailbox.MHMessage* 메서드), 1384
remove_signal_handler() (*asyncio.loop* 메서드), 1171
remove_writer() (*asyncio.loop* 메서드), 1168
removeAttribute() (*xml.dom.Element* 메서드), 1432
removeAttributeNode() (*xml.dom.Element* 메서드), 1432
removeAttributeNS() (*xml.dom.Element* 메서드), 1432
removeChild() (*xml.dom.Node* 메서드), 1430
removedirs() (*os* 모듈), 720

removeFilter() (*logging.Handler* 메서드), 791
removeFilter() (*logging.Logger* 메서드), 789
removeHandler() (*logging.Logger* 메서드), 789
removeHandler() (*unittest* 모듈), 1831
removeprefix() (*bytearray* 메서드), 71
removeprefix() (*bytes* 메서드), 71
removeprefix() (*str* 메서드), 60
removeResult() (*unittest* 모듈), 1831
removesuffix() (*bytearray* 메서드), 71
removesuffix() (*bytes* 메서드), 71
removesuffix() (*str* 메서드), 60
removexattr() (*os* 모듈), 738
rename() (*ftplib.FTP* 메서드), 1521
rename() (*imaplib.IMAP4* 메서드), 1532
rename() (*os* 모듈), 720
rename() (*pathlib.Path* 메서드), 481
renames() (*os* 모듈), 721
reopenIfNeeded() (*logging.handlers.WatchedFileHandler* 메서드), 816
reorganize() (*dbm.gnu.gdbm* 메서드), 548
--repeat
timeit 명령줄 옵션, 1949
repeat() (*itertools* 모듈), 432
repeat() (*timeit* 모듈), 1948
repeat() (*timeit.Timer* 메서드), 1949
repetition
operation, 47
REPL, **2315**
replace
에러 처리기의 이름, 202
replace() (*bytearray* 메서드), 73
replace() (*bytes* 메서드), 73
replace() (*copy* 모듈), 318
replace() (*curses.panel.Panel* 메서드), 988
replace() (*dataclasses* 모듈), 2039
replace() (*datetime.date* 메서드), 226
replace() (*datetime.datetime* 메서드), 234
replace() (*datetime.time* 메서드), 242
replace() (*inspect.Parameter* 메서드), 2090
replace() (*inspect.Signature* 메서드), 2088
replace() (*os* 모듈), 721
replace() (*pathlib.Path* 메서드), 482
replace() (*str* 메서드), 60
replace() (*tarfile.TarInfo* 메서드), 632
replace_errors() (*codecs* 모듈), 203
replace_header() (*email.message.EmailMessage* 메서드), 1307
replace_header() (*email.message.Message* 메서드), 1345
replace_history_item() (*readline* 모듈), 186
replace_whitespace (*textwrap.TextWrapper* 의 속성), 179
replaceChild() (*xml.dom.Node* 메서드), 1430
ReplacePackage() (*modulefinder* 모듈), 2122
--report
trace 명령줄 옵션, 1952
report() (*filecmp.dircmp* 메서드), 499
report() (*modulefinder.ModuleFinder* 메서드), 2122
REPORT_CDIFF (*doctest* 모듈), 1785
REPORT_ERRMODE (*msvcrt* 모듈), 2249
report_failure() (*doctest.DocTestRunner* 메서드), 1795
report_full_closure() (*filecmp.dircmp* 메서드), 500
REPORT_NDIFF (*doctest* 모듈), 1785
REPORT_ONLY_FIRST_FAILURE (*doctest* 모듈), 1785
report_partial_closure() (*filecmp.dircmp* 메서드), 499
report_start() (*doctest.DocTestRunner* 메서드), 1795
report_success() (*doctest.DocTestRunner* 메서드), 1795
REPORT_UDIFF (*doctest* 모듈), 1785
report_unexpected_exception() (*doctest.DocTestRunner* 메서드), 1795
REPORTING_FLAGS (*doctest* 모듈), 1786
Repr (*reprlib* 클래스), 324
repr()
built-in function, 28
repr() (*reprlib* 모듈), 325
repr() (*reprlib.Repr* 메서드), 327
repr1() (*reprlib.Repr* 메서드), 327
ReprEnum (*enum* 클래스), 338
reprlib
module, 324
request (*socketserver.BaseRequestHandler* 의 속성), 1552
Request (*urllib.request* 클래스), 1481
request() (*http.client.HTTPConnection* 메서드), 1512
request_queue_size (*socketserver.BaseServer* 의 속성), 1550
request_rate() (*urllib.robotparser.RobotFileParser* 메서드), 1506
request_uri() (*wsgiref.util* 모듈), 1468
request_version (*http.server.BaseHTTPRequestHandler* 의 속성), 1557
RequestHandlerClass (*socketserver.BaseServer* 의 속성), 1550
requestline (*http.server.BaseHTTPRequestHandler* 의 속성), 1557
Required (*typing* 모듈), 1736
requires() (*importlib.metadata* 모듈), 2155
requires() (*test.support* 모듈), 1899
requires_bz2() (*test.support* 모듈), 1902
requires_docstrings() (*test.support* 모듈), 1902
requires_freebsd_version() (*test.support* 모듈), 1901
requires_gil_enabled() (*test.support* 모듈), 1902
requires_gzip() (*test.support* 모듈), 1902
requires_IEEE_754() (*test.support* 모듈), 1902
requires_limited_api() (*test.support* 모듈), 1902
requires_linux_version() (*test.support* 모듈), 1901
requires_lzma() (*test.support* 모듈), 1902
requires_mac_version() (*test.support* 모듈), 1901

requires_resource() (*test.support* 모듈), 1902
requires_zlib() (*test.support* 모듈), 1902
RERAISE (*monitoring event*), 2010
RERAISE (*opcode*), 2232
reschedule() (*asyncio.Timeout* 메서드), 1123
reserved (*zipfile.ZipInfo*의 속성), 621
RESERVED_FUTURE (*uuid* 모듈), 1545
RESERVED_MICROSOFT (*uuid* 모듈), 1545
RESERVED_NCS (*uuid* 모듈), 1545
reset() (*asyncio.Barrier* 메서드), 1146
reset() (*bdb.Bdb* 메서드), 1921
reset() (*codecs.IncrementalDecoder* 메서드), 205
reset() (*codecs.IncrementalEncoder* 메서드), 205
reset() (*codecs.StreamReader* 메서드), 207
reset() (*codecs.StreamWriter* 메서드), 206
reset() (*contextvars.ContextVar* 메서드), 1102
reset() (*html.parser.HTMLParser* 메서드), 1403
reset() (*threading.Barrier* 메서드), 1011
reset() (*turtle* 모듈), 1696
reset() (*xml.dom.pulldom.DOMEventStream* 메서드), 1442
reset() (*xml.sax.xmlreader.IncrementalParser* 메서드), 1452
reset_mock() (*unittest.mock.AsyncMock* 메서드), 1846
reset_mock() (*unittest.mock.Mock* 메서드), 1836
reset_peak() (*tracemalloc* 모듈), 1959
reset_prog_mode() (*curses* 모듈), 962
reset_shell_mode() (*curses* 모듈), 962
reset_tzpath() (*zoneinfo* 모듈), 260
resetbuffer() (*code.InteractiveConsole* 메서드), 2115
resetscreen() (*turtle* 모듈), 1703
resetty() (*curses* 모듈), 962
resetwarnings() (*warnings* 모듈), 2031
resize() (*bytearray* 메서드), 70
resize() (*ctypes* 모듈), 870
resize() (*curses.window* 메서드), 969
resize() (*mmap.mmap* 메서드), 1300
resize_term() (*curses* 모듈), 962
resizemode() (*turtle* 모듈), 1697
resizeterm() (*curses* 모듈), 962
resolution (*datetime.datetime*의 속성), 232
resolution (*datetime.date*의 속성), 225
resolution (*datetime.timedelta*의 속성), 221
resolution (*datetime.time*의 속성), 241
resolve() (*pathlib.Path* 메서드), 473
resolve_bases() (*types* 모듈), 312
resolve_name() (*importlib.util* 모듈), 2141
resolve_name() (*pkgutil* 모듈), 2121
resolveEntity() (*xml.sax.handler.EntityResolver* 메서드), 1448
resource
module, 2278
resource_path() (*importlib.abc.ResourceReader* 메서드), 2133
resource_path() (*importlib.resources.abc.ResourceReader* 메서드), 2149
ResourceDenied, 1896
ResourceLoader (*importlib.abc* 클래스), 2130
ResourceReader (*importlib.abc* 클래스), 2133
ResourceReader (*importlib.resources.abc* 클래스), 2148
ResourceWarning, 121
response() (*imaplib.IMAP4* 메서드), 1532
ResponseNotReady, 1512
responses (*http.client* 모듈), 1512
responses (*http.server.BaseHTTPRequestHandler* 의 속성), 1558
restart (*pdb command*), 1937
restart_events() (*bdb.Bdb* 메서드), 1923
restart_events() (*sys.monitoring* 모듈), 2012
restore() (*difflib* 모듈), 167
restore() (*test.support.SaveSignals* 메서드), 1905
restype (*ctypes._CFuncPtr*의 속성), 864
result() (*asyncio.Future* 메서드), 1182
result() (*asyncio.Task* 메서드), 1129
result() (*concurrent.futures.Future* 메서드), 1070
results() (*trace.Trace* 메서드), 1953
RESUME (*opcode*), 2242
resume_reading() (*asyncio.ReadTransport* 메서드), 1187
resume_writing() (*asyncio.BaseProtocol* 메서드), 1190
retr() (*poplib.POP3* 메서드), 1526
retrbinary() (*ftplib.FTP* 메서드), 1520
retrlines() (*ftplib.FTP* 메서드), 1520
RETRY (*tkinter.messagebox* 모듈), 1646
RETRYCANCEL (*tkinter.messagebox* 모듈), 1646
Return (*ast* 클래스), 2193
return (*pdb command*), 1935
return_annotation (*inspect.Signature*의 속성), 2088
RETURN_GENERATOR (*opcode*), 2242
return_ok() (*http.cookiejar.CookiePolicy* 메서드), 1571
RETURN_VALUE (*opcode*), 2232
return_value (*unittest.mock.Mock*의 속성), 1838
returncode (*asyncio.subprocess.Process* 의 속성), 1150
returncode (*subprocess.CalledProcessError* 의 속성), 1079
returncode (*subprocess.CompletedProcess* 의 속성), 1078
returncode (*subprocess.Popen*의 속성), 1088
retval (*pdb command*), 1938
reveal_type() (*typing* 모듈), 1759
reverse() (*array.array* 메서드), 303
reverse() (*collections.deque* 메서드), 276
reverse() (*sequence* 메서드), 50
reverse_order() (*pstats.Stats* 메서드), 1944
reverse_pointer (*ipaddress.IPv4Address* 의 속성), 1591
reverse_pointer (*ipaddress.IPv6Address* 의 속성), 1593
reversed()

built-in function, 28
Reversible (*collections.abc* 클래스), 290
Reversible (*typing* 클래스), 1771
revert() (*http.cookiejar.FileCookieJar* 메서드), 1570
rewind() (*wave.Wave_read* 메서드), 1606
RFC
RFC 821, 1534, 1536
RFC 822, 778, 1335, 1352, 1514, 1537, 1539, 1540, 1612
RFC 959, 1517
RFC 1123, 778
RFC 1321, 673
RFC 1422, 1268, 1276
RFC 1521, 1398, 1400
RFC 1522, 1399, 1400
RFC 1730, 1527
RFC 1738, 1504
RFC 1750, 1245
RFC 1808, 1496, 1497, 1504
RFC 1869, 1534, 1536
RFC 1939, 1524
RFC 2045, 1303, 1307, 1329, 1345, 1346, 1352, 1394, 1397
RFC 2045 Section 6.8, 1579
RFC 2046, 1303, 1334, 1352
RFC 2047, 1303, 1322, 1327, 1328, 1352, 1353, 1358
RFC 2060, 1527, 1533
RFC 2068, 1563
RFC 2104, 685
RFC 2109, 15631568, 1573, 1574
RFC 2177, 1530
RFC 2183, 1303, 1309, 1348
RFC 2231, 1303, 1307, 1308, 13451347, 1352, 1359
RFC 2295, 1508
RFC 2324, 1508
RFC 2342, 1531
RFC 2368, 1504
RFC 2373, 1591, 1592
RFC 2396, 1499, 1502, 1504
RFC 2397, 1490
RFC 2449, 1525
RFC 2518, 1507
RFC 2595, 1524, 1526
RFC 2616, 1469, 1472, 1488, 1505
RFC 2616 Section 5.1.2, 1512
RFC 2616 Section 14.23, 1512
RFC 2640, 1517, 1519, 1523
RFC 2732, 1504
RFC 2774, 1508
RFC 2821, 1303
RFC 2822, 778, 1344, 1352, 1353, 13571359, 1380, 1511, 1558
RFC 2964, 1569
RFC 2965, 1481, 1484, 15671569, 15711575
RFC 3056, 1594
RFC 3171, 1591
RFC 3229, 1507
RFC 3280, 1256
RFC 3330, 1592
RFC 3454, 183
RFC 3490, 213, 215, 216
RFC 3490 Section 3.1, 215
RFC 3492, 213, 215
RFC 3493, 1241
RFC 3501, 1533
RFC 3542, 1228
RFC 3548, 1398
RFC 3659, 1521
RFC 3879, 1592, 1594
RFC 3927, 1592
RFC 3986, 1496, 1498, 15001502, 1504, 1557
RFC 4007, 1593, 1594
RFC 4086, 1277
RFC 4122, 1541, 1543, 1545
RFC 4180, 641
RFC 4193, 1594
RFC 4217, 1522
RFC 4291, 1592, 1593
RFC 4380, 1594
RFC 4627, 1360, 1369
RFC 4648, 1394, 1395, 1397, 2299
RFC 4918, 1507, 1508
RFC 4954, 1538
RFC 5161, 1529
RFC 5246, 1253, 1277
RFC 5280, 1243, 1244, 1246, 1277
RFC 5321, 1331
RFC 5322, 1303, 1304, 1313, 1316, 1317, 1320, 1322, 13251328, 1331, 1332, 1341, 1540
RFC 5424, 822
RFC 5735, 1592
RFC 5789, 1509
RFC 5842, 1507, 1508
RFC 5891, 215
RFC 5895, 215
RFC 5929, 1257
RFC 6066, 1252, 1262, 1277
RFC 6531, 1305, 1322, 1535
RFC 6532, 1303, 1304, 1313, 1322
RFC 6585, 1508
RFC 6855, 1529
RFC 6856, 1526
RFC 7159, 1360, 1368, 1369
RFC 7230, 1481, 1514
RFC 7301, 1251, 1261
RFC 7525, 1277
RFC 7693, 678
RFC 7725, 1508
RFC 7914, 677
RFC 8089, 472
RFC 8089 Section 3, 1480
RFC 8297, 1507
RFC 8305, 1162
RFC 8470, 1508

RFC 9110, 15071509
RFC 9562, 15411543, 1545
RFC 9562 Section 5.1, 1544
RFC 9562 Section 5.3, 1544
RFC 9562 Section 5.4, 1544
RFC 9562 Section 5.5, 1544
RFC 9562 Section 5.6, 1544
RFC 9562 Section 5.7, 1544
RFC 9562 Section 5.8, 1544
RFC 9562 Section 5.9, 1545
RFC 9562 Section 5.10, 1545
rfc2109 (*http.cookiejar.Cookie*의 속성), 1574
rfc2109_as_netscape (*http.cookiejar.DefaultCookiePolicy* 의 속성), 1573
rfc2965 (*http.cookiejar.CookiePolicy*의 속성), 1572
RFC_4122 (*uuid* 모듈), 1545
rfile (*http.server.BaseHTTPRequestHandler*의 속성), 1558
rfile (*socketserver.DatagramRequestHandler*의 속성), 1552
rfind() (*bytearray* 메서드), 73
rfind() (*bytes* 메서드), 73
rfind() (*mmap.mmap* 메서드), 1300
rfind() (*str* 메서드), 60
rgb_to_hls() (*colorsys* 모듈), 1608
rgb_to_hsv() (*colorsys* 모듈), 1608
rgb_to_yiq() (*colorsys* 모듈), 1608
rglob() (*pathlib.Path* 메서드), 478
right (*filecmp.dircmp*의 속성), 500
right() (*turtle* 모듈), 1685
right_list (*filecmp.dircmp*의 속성), 500
right_only (*filecmp.dircmp*의 속성), 500
RIGHTSHIFT (*token* 모듈), 2208
RIGHTSHIFTEQUAL (*token* 모듈), 2209
rindex() (*bytearray* 메서드), 73
rindex() (*bytes* 메서드), 73
rindex() (*str* 메서드), 60
rjust() (*bytearray* 메서드), 75
rjust() (*bytes* 메서드), 75
rjust() (*str* 메서드), 60
rlcompleter
module, 189
RLIM_INFINITY (*resource* 모듈), 2278
RLIMIT_AS (*resource* 모듈), 2280
RLIMIT_CORE (*resource* 모듈), 2279
RLIMIT_CPU (*resource* 모듈), 2279
RLIMIT_DATA (*resource* 모듈), 2279
RLIMIT_FSIZE (*resource* 모듈), 2279
RLIMIT_KQUEUES (*resource* 모듈), 2281
RLIMIT_MEMLOCK (*resource* 모듈), 2280
RLIMIT_MSGQUEUE (*resource* 모듈), 2280
RLIMIT_NICE (*resource* 모듈), 2280
RLIMIT_NOFILE (*resource* 모듈), 2279
RLIMIT_NPROC (*resource* 모듈), 2279
RLIMIT_NPTS (*resource* 모듈), 2281
RLIMIT_OFILE (*resource* 모듈), 2280
RLIMIT_RSS (*resource* 모듈), 2279
RLIMIT_RTPRIO (*resource* 모듈), 2280
RLIMIT_RTTIME (*resource* 모듈), 2280
RLIMIT_SBSIZE (*resource* 모듈), 2280
RLIMIT_SIGPENDING (*resource* 모듈), 2280
RLIMIT_STACK (*resource* 모듈), 2279
RLIMIT_SWAP (*resource* 모듈), 2280
RLIMIT_VMEM (*resource* 모듈), 2280
RLock (*multiprocessing* 클래스), 1031
RLock (*threading* 클래스), 1005
RLock() (*multiprocessing.managers.SyncManager* 메서드), 1037
rmd() (*ftplib.FTP* 메서드), 1522
rmdir() (*os* 모듈), 721
rmdir() (*pathlib.Path* 메서드), 482
rmdir() (*test.support.os_helper* 모듈), 1910
rmtree() (*shutil* 모듈), 514
rmtree() (*test.support.os_helper* 모듈), 1910
RobotFileParser (*urllib.robotparser* 클래스), 1505
robots.txt, 1505
rollback() (*sqlite3.Connection* 메서드), 558
rollover() (*tempfile.SpooledTemporaryFile* 메서드), 502
ROMAN (*tkinter.font* 모듈), 1640
--root
ensurepip 명령줄 옵션, 1966
root (*pathlib.PurePath*의 속성), 464
rotate() (*collections.deque* 메서드), 276
rotate() (*decimal.Context* 메서드), 382
rotate() (*decimal.Decimal* 메서드), 374
rotate() (*logging.handlers.BaseRotatingHandler* 메서드), 817
RotatingFileHandler (*logging.handlers* 클래스), 817
rotation_filename() (*logging.handlers.BaseRotatingHandler* 메서드), 817
rotator (*logging.handlers.BaseRotatingHandler*의 속성), 817
round()
built-in function, 28
ROUND_05UP (*decimal* 모듈), 383
ROUND_CEILING (*decimal* 모듈), 383
ROUND_DOWN (*decimal* 모듈), 383
ROUND_FLOOR (*decimal* 모듈), 383
ROUND_HALF_DOWN (*decimal* 모듈), 383
ROUND_HALF_EVEN (*decimal* 모듈), 383
ROUND_HALF_UP (*decimal* 모듈), 383
ROUND_UP (*decimal* 모듈), 383
Rounded (*decimal* 클래스), 384
rounding (*decimal.Context*의 속성), 377
rounds (*sys.float_info*의 속성), 1990
Row (*sqlite3* 클래스), 570
row_factory (*sqlite3.Connection*의 속성), 567
row_factory (*sqlite3.Cursor*의 속성), 570
rowcount (*sqlite3.Cursor*의 속성), 570
RPAR (*token* 모듈), 2207
rpartition() (*bytearray* 메서드), 73
rpartition() (*bytes* 메서드), 73

rpartition() (*str* 메서드), 60
rpc_paths (*xmlrpc.server.SimpleXMLRPCRequestHandler*의 속성), 1585
rpop() (*poplib.POP3* 메서드), 1525
RS (*curses.ascii* 모듈), 985
rset() (*poplib.POP3* 메서드), 1526
RShift (*ast* 클래스), 2171
rshift() (*operator* 모듈), 453
rsplit() (*bytearray* 메서드), 75
rsplit() (*bytes* 메서드), 75
rsplit() (*str* 메서드), 60
RSQB (*token* 모듈), 2207
rstrip() (*bytearray* 메서드), 75
rstrip() (*bytes* 메서드), 75
rstrip() (*str* 메서드), 61
rt() (*turtle* 모듈), 1685
RTLD_DEEPBIND (*os* 모듈), 755
RTLD_GLOBAL (*os* 모듈), 755
RTLD_LAZY (*os* 모듈), 755
RTLD_LOCAL (*os* 모듈), 755
RTLD_NODELETE (*os* 모듈), 755
RTLD_NOLOAD (*os* 모듈), 755
RTLD_NOW (*os* 모듈), 755
ruler (*cmd.Cmd*의 속성), 990
run (*pdb command*), 1937
run() (*asyncio* 모듈), 1110
run() (*asyncio.Runner* 메서드), 1111
run() (*bdb.Bdb* 메서드), 1924
run() (*contextvars.Context* 메서드), 1103
run() (*doctest.DocTestRunner* 메서드), 1795
run() (*multiprocessing.Process* 메서드), 1020
run() (*pdb* 모듈), 1930
run() (*pdb.Pdb* 메서드), 1932
run() (*profile* 모듈), 1941
run() (*profile.Profile* 메서드), 1942
run() (*sched.scheduler* 메서드), 1097
run() (*subprocess* 모듈), 1077
run() (*threading.Thread* 메서드), 1002
run() (*trace.Trace* 메서드), 1953
run() (*unittest.IsolatedAsyncioTestCase* 메서드), 1820
run() (*unittest.TestCase* 메서드), 1810
run() (*unittest.TestSuite* 메서드), 1821
run() (*unittest.TextTestRunner* 메서드), 1827
run() (*wsgiref.handlers.BaseHandler* 메서드), 1474
run_concurrently() (*test.support.threading_helper* 모듈), 1908
run_coroutine_threadsafe() (*asyncio* 모듈), 1127
run_docstring_examples() (*doctest* 모듈), 1789
run_forever() (*asyncio.loop* 메서드), 1158
run_in_executor() (*asyncio.loop* 메서드), 1171
run_in_subinterp() (*test.support* 모듈), 1904
run_module() (*runpy* 모듈), 2123
run_path() (*runpy* 모듈), 2124
run_python_until_end() (*test.support.script_helper* 모듈), 1906
run_script() (*modulefinder.ModuleFinder* 메서드), 2122
run_until_complete() (*asyncio.loop* 메서드), 1157
run_with_locale() (*test.support* 모듈), 1901
run_with_tz() (*test.support* 모듈), 1901
runcall() (*bdb.Bdb* 메서드), 1924
runcall() (*pdb* 모듈), 1930
runcall() (*pdb.Pdb* 메서드), 1932
runcall() (*profile.Profile* 메서드), 1942
runcode() (*code.InteractiveInterpreter* 메서드), 2114
runctx() (*bdb.Bdb* 메서드), 1924
runctx() (*profile* 모듈), 1941
runctx() (*profile.Profile* 메서드), 1942
runctx() (*trace.Trace* 메서드), 1953
runeval() (*bdb.Bdb* 메서드), 1924
runeval() (*pdb* 모듈), 1930
runeval() (*pdb.Pdb* 메서드), 1932
runfunc() (*trace.Trace* 메서드), 1953
Runner (*asyncio* 클래스), 1111
running() (*concurrent.futures.Future* 메서드), 1070
runpy
    module, 2123
runsource() (*code.InteractiveInterpreter* 메서드), 2114
runtime (*sys._emscripten_info*의 속성), 1985
runtime_checkable() (*typing* 모듈), 1752
RuntimeError, 116
RuntimeWarning, 120
RUSAGE_BOTH (*resource* 모듈), 2282
RUSAGE_CHILDREN (*resource* 모듈), 2282
RUSAGE_SELF (*resource* 모듈), 2282
RUSAGE_THREAD (*resource* 모듈), 2282
RWF_APPEND (*os* 모듈), 707
RWF_DSYNC (*os* 모듈), 707
RWF_HIPRI (*os* 모듈), 706
RWF_NOWAIT (*os* 모듈), 706
RWF_SYNC (*os* 모듈), 707

S

-S
    dis 명령줄 옵션, 2223
-s
    calendar 명령줄 옵션, 269
    compileall 명령줄 옵션, 2219
    cProfile 명령줄 옵션, 1940
    idle 명령줄 옵션, 1674
    timeit 명령줄 옵션, 1949
    trace 명령줄 옵션, 1953
    unittest-discover 명령줄 옵션, 1804
S (*re* 모듈), 150
S_ENFMT (*stat* 모듈), 496
S_IEXEC (*stat* 모듈), 497
S_IFBLK (*stat* 모듈), 495
S_IFCHR (*stat* 모듈), 495
S_IFDIR (*stat* 모듈), 495
S_IFDOOR (*stat* 모듈), 495
S_IFIFO (*stat* 모듈), 495
S_IFLNK (*stat* 모듈), 495
S_IFMT() (*stat* 모듈), 493
S_IFPORT (*stat* 모듈), 495

S_IFREG (*stat* 모듈), 495
S_IFSOCK (*stat* 모듈), 495
S_IFWHT (*stat* 모듈), 495
S_IMODE() (*stat* 모듈), 493
S_IREAD (*stat* 모듈), 496
S_IRGRP (*stat* 모듈), 496
S_IROTH (*stat* 모듈), 496
S_IRUSR (*stat* 모듈), 496
S_IRWXG (*stat* 모듈), 496
S_IRWXO (*stat* 모듈), 496
S_IRWXU (*stat* 모듈), 496
S_ISBLK() (*stat* 모듈), 493
S_ISCHR() (*stat* 모듈), 493
S_ISDIR() (*stat* 모듈), 493
S_ISDOOR() (*stat* 모듈), 493
S_ISFIFO() (*stat* 모듈), 493
S_ISGID (*stat* 모듈), 496
S_ISLNK() (*stat* 모듈), 493
S_ISPORT() (*stat* 모듈), 493
S_ISREG() (*stat* 모듈), 493
S_ISSOCK() (*stat* 모듈), 493
S_ISUID (*stat* 모듈), 496
S_ISVTX (*stat* 모듈), 496
S_ISWHT() (*stat* 모듈), 493
S_IWGRP (*stat* 모듈), 496
S_IWOTH (*stat* 모듈), 496
S_IWRITE (*stat* 모듈), 497
S_IWUSR (*stat* 모듈), 496
S_IXGRP (*stat* 모듈), 496
S_IXOTH (*stat* 모듈), 496
S_IXUSR (*stat* 모듈), 496
safe (*uuid.SafeUUID*의 속성), 1541
safe_path (*sys.flags*의 속성), 1988
safe_substitute() (*string.Template* 메서드), 135
saferepr() (*pprint* 모듈), 320
SafeUUID (*uuid* 클래스), 1541
same_files (*filecmp.dircmp*의 속성), 500
same_quantum() (*decimal.Context* 메서드), 382
same_quantum() (*decimal.Decimal* 메서드), 375
samefile() (*os.path* 모듈), 491
samefile() (*pathlib.Path* 메서드), 475
SameFileError, 512
sameopenfile() (*os.path* 모듈), 491
samesite (*http.cookies.Morsel*의 속성), 1565
samestat() (*os.path* 모듈), 491
sample() (*random* 모듈), 399
samples() (*statistics.NormalDist* 메서드), 418
SATURDAY (*calendar* 모듈), 266
save() (*http.cookiejar.FileCookieJar* 메서드), 1570
save() (*test.support.SaveSignals* 메서드), 1905
save() (*turtle* 모듈), 1709
SaveAs (*tkinter.filedialog* 클래스), 1643
SAVEDCWD (*test.support.os_helper* 모듈), 1908
SaveFileDialog (*tkinter.filedialog* 클래스), 1644
SaveKey() (*winreg* 모듈), 2254
SaveSignals (*test.support* 클래스), 1905
savetty() (*curses* 모듈), 962
SAX2DOM (*xml.dom.pulldom* 클래스), 1441
SAXException, 1443
SAXNotRecognizedException, 1444
SAXNotSupportedException, 1444
SAXParseException, 1443
scaleb() (*decimal.Context* 메서드), 382
scaleb() (*decimal.Decimal* 메서드), 375
scandir() (*os* 모듈), 721
scanf (*C function*), 160
sched
    module, 1096
SCHED_BATCH (*os* 모듈), 752
SCHED_DEADLINE (*os* 모듈), 752
SCHED_FIFO (*os* 모듈), 752
sched_get_priority_max() (*os* 모듈), 753
sched_get_priority_min() (*os* 모듈), 753
sched_getaffinity() (*os* 모듈), 753
sched_getparam() (*os* 모듈), 753
sched_getscheduler() (*os* 모듈), 753
SCHED_IDLE (*os* 모듈), 752
SCHED_NORMAL (*os* 모듈), 752
SCHED_OTHER (*os* 모듈), 752
sched_param (*os* 클래스), 753
sched_priority (*os.sched_param*의 속성), 753
SCHED_RESET_ON_FORK (*os* 모듈), 753
SCHED_RR (*os* 모듈), 752
sched_rr_get_interval() (*os* 모듈), 753
sched_setaffinity() (*os* 모듈), 753
sched_setparam() (*os* 모듈), 753
sched_setscheduler() (*os* 모듈), 753
SCHED_SPORADIC (*os* 모듈), 752
sched_yield() (*os* 모듈), 753
scheduler (*sched* 클래스), 1096
SCM_CREDS2 (*socket* 모듈), 1221
scope_id (*ipaddress.IPv6Address*의 속성), 1594
Screen (*turtle* 클래스), 1710
screensize() (*turtle* 모듈), 1703
script_from_examples() (*doctest* 모듈), 1797
scroll() (*curses.window* 메서드), 970
ScrolledCanvas (*turtle* 클래스), 1710
ScrolledText (*tkinter.scrolledtext* 클래스), 1647
scrollok() (*curses.window* 메서드), 970
scrypt() (*hashlib* 모듈), 677
seal() (*unittest.mock* 모듈), 1871
search() (*imaplib.IMAP4* 메서드), 1532
search() (*re* 모듈), 151
search() (*re.Pattern* 메서드), 155
search_function() (*encodings* 모듈), 215
search_log (*compression.zstd.CompressionParameter*의 속성), 590
second (*datetime.datetime*의 속성), 233
second (*datetime.time*의 속성), 241
seconds (*datetime.timedelta*의 속성), 221
secrets
    module, 686
SECTCRE (*configparser.ConfigParser*의 속성), 659
sections() (*configparser.ConfigParser* 메서드), 663
secure (*http.cookiejar.Cookie*의 속성), 1574
secure (*http.cookies.Morsel*의 속성), 1565

secure hash algorithm, SHA1, SHA2, SHA224, SHA256, SHA384, SHA512, SHA3, Shake, Blake2, 673
Secure Sockets Layer, 1241
security considerations, 2297
security_level (*ssl.SSLContext*의 속성), 1266
see() (*tkinter.ttk.Treeview* 메서드), 1660
seed() (*random* 모듈), 397
seed() (*random.Random* 메서드), 401
seed_bits (*sys.hash_info*의 속성), 1995
seek() (*io.IOBase* 메서드), 761
seek() (*io.TextIOBase* 메서드), 767
seek() (*io.TextIOWrapper* 메서드), 769
seek() (*mmap.mmap* 메서드), 1301
seek() (*sqlite3.Blob* 메서드), 571
SEEK_CUR (*os* 모듈), 703
SEEK_DATA (*os* 모듈), 703
SEEK_END (*os* 모듈), 703
SEEK_HOLE (*os* 모듈), 703
SEEK_SET (*os* 모듈), 703
seekable() (*bz2.BZ2File* 메서드), 603
seekable() (*io.IOBase* 메서드), 761
seekable() (*mmap.mmap* 메서드), 1301
select
    module, 1277
select() (*imaplib.IMAP4* 메서드), 1532
select() (*select* 모듈), 1278
select() (*selectors.BaseSelector* 메서드), 1286
select() (*tkinter.ttk.Notebook* 메서드), 1655
selected_alpn_protocol() (*ssl.SSLSocket* 메서드), 1257
selected_npn_protocol() (*ssl.SSLSocket* 메서드), 1257
selection() (*tkinter.ttk.Treeview* 메서드), 1660
selection_add() (*tkinter.ttk.Treeview* 메서드), 1661
selection_remove() (*tkinter.ttk.Treeview* 메서드), 1661
selection_set() (*tkinter.ttk.Treeview* 메서드), 1661
selection_toggle() (*tkinter.ttk.Treeview* 메서드), 1661
selector (*urllib.request.Request*의 속성), 1484
SelectorEventLoop (*asyncio* 클래스), 1177
SelectorKey (*selectors* 클래스), 1285
selectors
    module, 1284
SelectSelector (*selectors* 클래스), 1287
Self (*typing* 모듈), 1731
Semaphore (*asyncio* 클래스), 1144
Semaphore (*multiprocessing* 클래스), 1032
Semaphore (*threading* 클래스), 1008
Semaphore() (*multiprocessing.managers.SyncManager* 메서드), 1037
SEMI (*token* 모듈), 2207
SEND (*opcode*), 2242
send() (*http.client.HTTPConnection* 메서드), 1515
send() (*imaplib.IMAP4* 메서드), 1532
send() (*logging.handlers.DatagramHandler* 메서드), 821
send() (*logging.handlers.SocketHandler* 메서드), 820
send() (*multiprocessing.connection.Connection* 메서드), 1028
send() (*socket.socket* 메서드), 1235
send_bytes() (*multiprocessing.connection.Connection* 메서드), 1029
send_error() (*http.server.BaseHTTPRequestHandler* 메서드), 1559
send_fds() (*socket* 모듈), 1230
send_header() (*http.server.BaseHTTPRequestHandler* 메서드), 1559
send_message() (*smtplib.SMTP* 메서드), 1540
send_response() (*http.server.BaseHTTPRequestHandler* 메서드), 1559
send_response_only() (*http.server.BaseHTTPRequestHandler* 메서드), 1559
send_signal() (*asyncio.subprocess.Process* 메서드), 1149
send_signal() (*asyncio.SubprocessTransport* 메서드), 1188
send_signal() (*subprocess.Popen* 메서드), 1087
sendall() (*socket.socket* 메서드), 1235
sendcmd() (*ftplib.FTP* 메서드), 1519
sendfile() (*asyncio.loop* 메서드), 1167
sendfile() (*os* 모듈), 708
sendfile() (*socket.socket* 메서드), 1236
sendfile() (*wsgiref.handlers.BaseHandler* 메서드), 1476
SendfileNotAvailableError, 1154
sendmail() (*smtplib.SMTP* 메서드), 1539
sendmsg() (*socket.socket* 메서드), 1235
sendmsg_afalg() (*socket.socket* 메서드), 1236
sendto() (*asyncio.DatagramTransport* 메서드), 1188
sendto() (*socket.socket* 메서드), 1235
sentinel (*multiprocessing.Process*의 속성), 1021
sentinel (*unittest.mock* 모듈), 1864
sep (*os* 모듈), 755
SEPTEMBER (*calendar* 모듈), 266
sequence
    iteration, 46
    object, 47
    types, immutable, 49
    types, mutable, 49
    types, operations on, 47, 49
Sequence (*collections.abc* 클래스), 290
Sequence (*typing* 클래스), 1770
sequence (시퀀스), **2315**
SequenceMatcher (*difflib* 클래스), 169
serialize() (*sqlite3.Connection* 메서드), 566
serve_forever() (*asyncio.Server* 메서드), 1177
serve_forever() (*socketserver.BaseServer* 메서드), 1550
Server (*asyncio* 클래스), 1176
server (*http.server.BaseHTTPRequestHandler*의 속성), 1557
server (*socketserver.BaseRequestHandler*의 속성), 1552

server_activate() (*socketserver.BaseServer* 메서드), 1551
server_address (*socketserver.BaseServer*의 속성), 1550
server_bind() (*socketserver.BaseServer* 메서드), 1551
server_close() (*socketserver.BaseServer* 메서드), 1550
server_hostname (*ssl.SSLSocket*의 속성), 1258
server_side (*ssl.SSLSocket*의 속성), 1258
server_software (*wsgiref.handlers.BaseHandler* 의 속성), 1475
server_version (*http.server.BaseHTTPRequestHandler* 의 속성), 1558
server_version (*http.server.SimpleHTTPRequestHandler* 의 속성), 1560
ServerProxy (*xmlrpc.client* 클래스), 1576
service_actions() (*socketserver.BaseServer* 메서드), 1550
session (*ssl.SSLSocket*의 속성), 1258
session_reused (*ssl.SSLSocket*의 속성), 1258
session_stats() (*ssl.SSLContext* 메서드), 1264
set
object, 91
Set (*ast* 클래스), 2169
Set (*collections.abc* 클래스), 291
Set (*typing* 클래스), 1767
set (내장 클래스), 91
Set Breakpoint, 1670
set comprehension (집합 컴프리헨션), **2315**
set() (*asyncio.Event* 메서드), 1142
set() (*configparser.ConfigParser* 메서드), 664
set() (*configparser.RawConfigParser* 메서드), 666
set() (*contextvars.ContextVar* 메서드), 1102
set() (*http.cookies.Morsel* 메서드), 1565
set() (*multiprocessing.managers.SyncManager* 메서드), 1037
set() (*test.support.os_helper.EnvironmentVarGuard* 메서드), 1909
set() (*threading.Event* 메서드), 1010
set() (*tkinter.ttk.Combobox* 메서드), 1652
set() (*tkinter.ttk.Spinbox* 메서드), 1653
set() (*tkinter.ttk.Treeview* 메서드), 1661
set() (*xml.etree.ElementTree.Element* 메서드), 1419
SET_ADD (*opcode*), 2231
set_allowed_domains() (*http.cookiejar.DefaultCookiePolicy* 메서드), 1573
set_alpn_protocols() (*ssl.SSLContext* 메서드), 1261
set_app() (*wsgiref.simple_server.WSGIServer* 메서드), 1471
set_asyncgen_hooks() (*sys* 모듈), 2003
set_authorizer() (*sqlite3.Connection* 메서드), 561
set_auto_history() (*readline* 모듈), 186
set_blocked_domains() (*http.cookiejar.DefaultCookiePolicy* 메서드), 1572
set_blocking() (*os* 모듈), 709
set_boundary() (*email.message.EmailMessage* 메서드), 1308
set_boundary() (*email.message.Message* 메서드), 1347
set_break() (*bdb.Bdb* 메서드), 1923
set_charset() (*email.message.Message* 메서드), 1343
set_children() (*tkinter.ttk.Treeview* 메서드), 1658
set_ciphers() (*ssl.SSLContext* 메서드), 1261
set_completer() (*readline* 모듈), 187
set_completer_delims() (*readline* 모듈), 187
set_completion_display_matches_hook() (*readline* 모듈), 187
set_content() (*email.contentmanager* 모듈), 1333
set_content() (*email.contentmanager.ContentManager* 메서드), 1332
set_content() (*email.message.EmailMessage* 메서드), 1310
set_continue() (*bdb.Bdb* 메서드), 1923
set_cookie() (*http.cookiejar.CookieJar* 메서드), 1569
set_cookie_if_ok() (*http.cookiejar.CookieJar* 메서드), 1569
set_coroutine_origin_tracking_depth() (*sys* 모듈), 2003
set_data() (*importlib.abc.SourceLoader* 메서드), 2132
set_data() (*importlib.machinery.SourceFileLoader* 메서드), 2137
set_date() (*mailbox.MaildirMessage* 메서드), 1382
set_debug() (*asyncio.loop* 메서드), 1173
set_debug() (*gc* 모듈), 2078
set_debuglevel() (*ftplib.FTP* 메서드), 1519
set_debuglevel() (*http.client.HTTPConnection* 메서드), 1513
set_debuglevel() (*poplib.POP3* 메서드), 1525
set_debuglevel() (*smtplib.SMTP* 메서드), 1537
set_default_backend() (*pdb* 모듈), 1931
set_default_executor() (*asyncio.loop* 메서드), 1172
set_default_type() (*email.message.EmailMessage* 메서드), 1308
set_default_type() (*email.message.Message* 메서드), 1346
set_default_verify_paths() (*ssl.SSLContext* 메서드), 1261
set_defaults() (*argparse.ArgumentParser* 메서드), 909
set_defaults() (*optparse.OptionParser* 메서드), 946
set_ecdh_curve() (*ssl.SSLContext* 메서드), 1263
set_errno() (*ctypes* 모듈), 870
set_error_mode() (*msvcrt* 모듈), 2248
set_escdelay() (*curses* 모듈), 962
set_event_loop() (*asyncio* 모듈), 1156
set_event_loop() (*asyncio.AbstractEventLoopPolicy* 메서드), 1198
set_event_loop_policy() (*asyncio* 모듈), 1198

set_events() (*sys.monitoring* 모듈), 2011
set_exception() (*asyncio.Future* 메서드), 1182
set_exception() (*concurrent.futures.Future* 메서드), 1070
set_exception_handler() (*asyncio.loop* 메서드), 1172
set_executable() (*multiprocessing* 모듈), 1027
set_filter() (*tkinter.filedialog.FileDialog* 메서드), 1644
set_flags() (*mailbox.Maildir* 메서드), 1375
set_flags() (*mailbox.MaildirMessage* 메서드), 1381
set_flags() (*mailbox.mboxMessage* 메서드), 1383
set_flags() (*mailbox.MMDFMessage* 메서드), 1387
set_forkserver_preload() (*multiprocessing* 모듈), 1027
set_from() (*mailbox.mboxMessage* 메서드), 1383
set_from() (*mailbox.MMDFMessage* 메서드), 1387
SET_FUNCTION_ATTRIBUTE (*opcode*), 2240
set_handle_inheritable() (*os* 모듈), 712
set_history_length() (*readline* 모듈), 186
set_info() (*mailbox.Maildir* 메서드), 1375
set_info() (*mailbox.MaildirMessage* 메서드), 1382
set_inheritable() (*os* 모듈), 712
set_inheritable() (*socket.socket* 메서드), 1236
set_int_max_str_digits() (*sys* 모듈), 2001
set_labels() (*mailbox.BabylMessage* 메서드), 1385
set_last_error() (*ctypes* 모듈), 870
set_local_events() (*sys.monitoring* 모듈), 2012
set_memlimit() (*test.support* 모듈), 1899
set_name() (*asyncio.Task* 메서드), 1130
set_next() (*bdb.Bdb* 메서드), 1922
set_nonstandard_attr() (*http.cookiejar.Cookie* 메서드), 1575
set_npn_protocols() (*ssl.SSLContext* 메서드), 1262
set_ok() (*http.cookiejar.CookiePolicy* 메서드), 1571
set_param() (*email.message.EmailMessage* 메서드), 1308
set_param() (*email.message.Message* 메서드), 1346
set_pasv() (*ftplib.FTP* 메서드), 1520
set_payload() (*email.message.Message* 메서드), 1343
set_pledged_input_size() (*compression.zstd.ZstdCompressor* 메서드), 586
set_policy() (*http.cookiejar.CookieJar* 메서드), 1569
set_pre_input_hook() (*readline* 모듈), 186
set_progress_handler() (*sqlite3.Connection* 메서드), 562
set_protocol() (*asyncio.BaseTransport* 메서드), 1186
set_proxy() (*urllib.request.Request* 메서드), 1485
set_psk_client_callback() (*ssl.SSLContext* 메서드), 1266
set_psk_server_callback() (*ssl.SSLContext* 메서드), 1267
set_quit() (*bdb.Bdb* 메서드), 1923
set_result() (*asyncio.Future* 메서드), 1182
set_result() (*concurrent.futures.Future* 메서드), 1070
set_return() (*bdb.Bdb* 메서드), 1922
set_running_or_notify_cancel() (*concurrent.futures.Future* 메서드), 1070
set_selection() (*tkinter.filedialog.FileDialog* 메서드), 1644
set_seq1() (*difflib.SequenceMatcher* 메서드), 169
set_seq2() (*difflib.SequenceMatcher* 메서드), 169
set_seqs() (*difflib.SequenceMatcher* 메서드), 169
set_sequences() (*mailbox.MH* 메서드), 1378
set_sequences() (*mailbox.MHMessage* 메서드), 1384
set_server_documentation() (*xmlrpc.server.DocCGIXMLRPCRequestHandler* 메서드), 1589
set_server_documentation() (*xmlrpc.server.DocXMLRPCServer* 메서드), 1589
set_server_name() (*xmlrpc.server.DocCGIXMLRPCRequestHandler* 메서드), 1589
set_server_name() (*xmlrpc.server.DocXMLRPCServer* 메서드), 1589
set_server_title() (*xmlrpc.server.DocCGIXMLRPCRequestHandler* 메서드), 1589
set_server_title() (*xmlrpc.server.DocXMLRPCServer* 메서드), 1589
set_servername_callback() (*ssl.SSLContext* 메서드), 1262
set_start_method() (*multiprocessing* 모듈), 1028
set_startup_hook() (*readline* 모듈), 186
set_step() (*bdb.Bdb* 메서드), 1922
set_subdir() (*mailbox.MaildirMessage* 메서드), 1381
set_tabsize() (*curses* 모듈), 963
set_task_factory() (*asyncio.loop* 메서드), 1161
set_threshold() (*gc* 모듈), 2078
set_trace() (*bdb* 모듈), 1924
set_trace() (*bdb.Bdb* 메서드), 1922
set_trace() (*pdb* 모듈), 1930
set_trace() (*pdb.Pdb* 메서드), 1932
set_trace_async() (*pdb* 모듈), 1931
set_trace_callback() (*sqlite3.Connection* 메서드), 562
set_tunnel() (*http.client.HTTPConnection* 메서드), 1513
set_type() (*email.message.Message* 메서드), 1347
set_unittest_reportflags() (*doctest* 모듈), 1791
set_unixfrom() (*email.message.EmailMessage* 메서드), 1305
set_unixfrom() (*email.message.Message* 메서드), 1342
set_until() (*bdb.Bdb* 메서드), 1922
SET_UPDATE (*opcode*), 2236

set_url() (*urllib.robotparser.RobotFileParser* 메서드), 1505
set_usage() (*optparse.OptionParser* 메서드), 946
set_userptr() (*curses.panel.Panel* 메서드), 988
set_visible() (*mailbox.BabylMessage* 메서드), 1386
set_wakeup_fd() (*signal* 모듈), 1293
set_write_buffer_limits() (*asyncio.WriteTransport* 메서드), 1187
setacl() (*imaplib.IMAP4* 메서드), 1532
setannotation() (*imaplib.IMAP4* 메서드), 1532
setattr()
built-in function, 28
setAttribute() (*xml.dom.Element* 메서드), 1433
setAttributeNode() (*xml.dom.Element* 메서드), 1433
setAttributeNodeNS() (*xml.dom.Element* 메서드), 1433
setAttributeNS() (*xml.dom.Element* 메서드), 1433
SetBase() (*xml.parsers.expat.xmlparser* 메서드), 1456
setblocking() (*socket.socket* 메서드), 1236
setByteStream() (*xml.sax.xmlreader.InputSource* 메서드), 1453
setcbreak() (*tty* 모듈), 2273
setCharacterStream() (*xml.sax.xmlreader.InputSource* 메서드), 1453
SetComp (*ast* 클래스), 2174
setcomptype() (*wave.Wave_write* 메서드), 1607
setconfig() (*sqlite3.Connection* 메서드), 565
setContentHandler() (*xml.sax.xmlreader.XMLReader* 메서드), 1451
setcontext() (*decimal* 모듈), 376
setDaemon() (*threading.Thread* 메서드), 1004
setdefault() (*dict* 메서드), 96
setdefault() (*http.cookies.Morsel* 메서드), 1566
setdefaulttimeout() (*socket* 모듈), 1229
setdlopenflags() (*sys* 모듈), 2001
setDocumentLocator() (*xml.sax.handler.ContentHandler* 메서드), 1446
setDTDHandler() (*xml.sax.xmlreader.XMLReader* 메서드), 1452
setegid() (*os* 모듈), 696
setEncoding() (*xml.sax.xmlreader.InputSource* 메서드), 1453
setEntityResolver() (*xml.sax.xmlreader.XMLReader* 메서드), 1452
setErrorHandler() (*xml.sax.xmlreader.XMLReader* 메서드), 1452
seteuid() (*os* 모듈), 696
setFeature() (*xml.sax.xmlreader.XMLReader* 메서드), 1452
setfirstweekday() (*calendar* 모듈), 265
setfirstweekday() (*calendar.Calendar* 메서드), 261
setFormatter() (*logging.Handler* 메서드), 790
setframerate() (*wave.Wave_write* 메서드), 1607
setgid() (*os* 모듈), 696
setgroups() (*os* 모듈), 696
seth() (*turtle* 모듈), 1687
setheading() (*turtle* 모듈), 1687
sethostname() (*socket* 모듈), 1229
setinputsizes() (*sqlite3.Cursor* 메서드), 569
setitem() (*operator* 모듈), 454
setitimer() (*signal* 모듈), 1293
setLevel() (*logging.Handler* 메서드), 790
setLevel() (*logging.Logger* 메서드), 786
setlimit() (*sqlite3.Connection* 메서드), 565
setlocale() (*locale* 모듈), 1617
setLocale() (*xml.sax.xmlreader.XMLReader* 메서드), 1452
setLoggerClass() (*logging* 모듈), 800
setlogmask() (*syslog* 모듈), 2283
setLogRecordFactory() (*logging* 모듈), 800
setMaxConns() (*urllib.request.CacheFTPHandler* 메서드), 1491
setmode() (*msvcrt* 모듈), 2248
setName() (*threading.Thread* 메서드), 1003
setnchannels() (*wave.Wave_write* 메서드), 1607
setnframes() (*wave.Wave_write* 메서드), 1607
setns() (*os* 모듈), 696
setoutputsize() (*sqlite3.Cursor* 메서드), 569
SetParamEntityParsing() (*xml.parsers.expat.xmlparser* 메서드), 1456
setparams() (*wave.Wave_write* 메서드), 1607
setpassword() (*zipfile.ZipFile* 메서드), 616
setpgid() (*os* 모듈), 697
setpgrp() (*os* 모듈), 696
setpos() (*turtle* 모듈), 1686
setpos() (*wave.Wave_read* 메서드), 1606
setposition() (*turtle* 모듈), 1686
setpriority() (*os* 모듈), 697
setprofile() (*sys* 모듈), 2001
setprofile() (*threading* 모듈), 998
setprofile_all_threads() (*threading* 모듈), 998
setProperty() (*xml.sax.xmlreader.XMLReader* 메서드), 1452
setPublicId() (*xml.sax.xmlreader.InputSource* 메서드), 1453
setquota() (*imaplib.IMAP4* 메서드), 1532
setraw() (*tty* 모듈), 2273
setrecursionlimit() (*sys* 모듈), 2002
setregid() (*os* 모듈), 697
SetReparseDeferralEnabled() (*xml.parsers.expat.xmlparser* 메서드), 1456
setresgid() (*os* 모듈), 697
setresuid() (*os* 모듈), 697
setreuid() (*os* 모듈), 697
setrlimit() (*resource* 모듈), 2278
setsampwidth() (*wave.Wave_write* 메서드), 1607
setscrreg() (*curses.window* 메서드), 970
setsid() (*os* 모듈), 697
setsockopt() (*socket.socket* 메서드), 1236

setstate() (*codecs.IncrementalDecoder* 메서드), 205
setstate() (*codecs.IncrementalEncoder* 메서드), 205
setstate() (*random* 모듈), 398
setstate() (*random.Random* 메서드), 401
setStream() (*logging.StreamHandler* 메서드), 814
setswitchinterval() (*sys* 모듈), 2002
setswitchinterval() (*test.support* 모듈), 1899
setSystemId() (*xml.sax.xmlreader.InputSource* 메서드), 1453
setsyx() (*curses* 모듈), 963
setTarget() (*logging.handlers.MemoryHandler* 메서드), 825
settimeout() (*socket.socket* 메서드), 1236
setTimeout() (*urllib.request.CacheFTPHandler* 메서드), 1491
settrace() (*sys* 모듈), 2002
settrace() (*threading* 모듈), 998
settrace_all_threads() (*threading* 모듈), 998
setuid() (*os* 모듈), 697
setundobuffer() (*turtle* 모듈), 1701
--setup
    timeit 명령줄 옵션, 1949
setup() (*socketserver.BaseRequestHandler* 메서드), 1551
setup() (*turtle* 모듈), 1709
setUp() (*unittest.TestCase* 메서드), 1810
SETUP_ANNOTATIONS (*opcode*), 2232
SETUP_CLEANUP (*opcode*), 2243
setup_environ() (*wsgiref.handlers.BaseHandler* 메서드), 1475
SETUP_FINALLY (*opcode*), 2243
setup_python() (*venv.EnvBuilder* 메서드), 1972
setup_scripts() (*venv.EnvBuilder* 메서드), 1972
setup_testing_defaults() (*wsgiref.util* 모듈), 1469
SETUP_WITH (*opcode*), 2243
setUpClass() (*unittest.TestCase* 메서드), 1810
setUpModule() (*unittest* 모듈), 1830
setupterm() (*curses* 모듈), 963
SetValue() (*winreg* 모듈), 2254
SetValueEx() (*winreg* 모듈), 2254
setworldcoordinates() (*turtle* 모듈), 1703
setx() (*turtle* 모듈), 1687
setxattr() (*os* 모듈), 738
sety() (*turtle* 모듈), 1687
SF_APPEND (*stat* 모듈), 498
SF_ARCHIVED (*stat* 모듈), 497
SF_DATALESS (*stat* 모듈), 498
SF_FIRMLINK (*stat* 모듈), 498
SF_IMMUTABLE (*stat* 모듈), 498
SF_MNOWAIT (*os* 모듈), 709
SF_NOCACHE (*os* 모듈), 709
SF_NODISKIO (*os* 모듈), 709
SF_NOUNLINK (*stat* 모듈), 498
SF_RESTRICTED (*stat* 모듈), 498
SF_SETTABLE (*stat* 모듈), 497
SF_SNAPSHOT (*stat* 모듈), 498
SF_SUPPORTED (*stat* 모듈), 497
SF_SYNC (*os* 모듈), 709
SF_SYNTHETIC (*stat* 모듈), 497
sha1() (*hashlib* 모듈), 674
sha3_224() (*hashlib* 모듈), 675
sha3_256() (*hashlib* 모듈), 675
sha3_384() (*hashlib* 모듈), 675
sha3_512() (*hashlib* 모듈), 675
sha224() (*hashlib* 모듈), 674
sha256() (*hashlib* 모듈), 674
sha384() (*hashlib* 모듈), 675
sha512() (*hashlib* 모듈), 675
shake_128() (*hashlib* 모듈), 676
shake_256() (*hashlib* 모듈), 676
shape (*memoryview*의 속성), 90
Shape (*turtle* 클래스), 1710
shape() (*turtle* 모듈), 1697
shapesize() (*turtle* 모듈), 1697
shapetransform() (*turtle* 모듈), 1698
share() (*socket.socket* 메서드), 1237
ShareableList (*multiprocessing.shared_memory* 클래스), 1061
ShareableList() (*multiprocessing.managers.SharedMemoryManager* 메서드), 1060
shared_ciphers() (*ssl.SSLSocket* 메서드), 1257
shared_memory (*sys._emscripten_info*의 속성), 1985
SharedMemory (*multiprocessing.shared_memory* 클래스), 1058
SharedMemory() (*multiprocessing.managers.SharedMemoryManager* 메서드), 1060
SharedMemoryManager (*multiprocessing.managers* 클래스), 1060
shearfactor() (*turtle* 모듈), 1698
Shelf (*shelve* 클래스), 542
shelve
    module, 540
shield() (*asyncio* 모듈), 1121
shift() (*decimal.Context* 메서드), 382
shift() (*decimal.Decimal* 메서드), 375
shift_path_info() (*wsgiref.util* 모듈), 1468
shifting
    operations, 40
shlex
    module, 2263
shlex (*shlex* 클래스), 2264
shm (*multiprocessing.shared_memory.ShareableList*의 속성), 1062
SHORT_TIMEOUT (*test.support* 모듈), 1897
shortDescription() (*unittest.TestCase* 메서드), 1818
shorten() (*textwrap* 모듈), 177
shouldFlush() (*logging.handlers.BufferingHandler* 메서드), 824
shouldFlush() (*logging.handlers.MemoryHandler* 메서드), 825
shouldRollover() (*logging.handlers.RotatingFileHandler* 메서

드), 818
shouldRollover() (*logging.handlers.TimedRotatingFileHandler* 메서드), 819
shouldStop (*unittest.TestResult*의 속성), 1825
show() (*curses.panel.Panel* 메서드), 988
show() (*tkinter.commondialog.Dialog* 메서드), 1644
show() (*tkinter.messagebox.Message* 메서드), 1645
show_code() (*dis* 모듈), 2225
show_flag_values() (*enum* 모듈), 343
--show-caches
    dis 명령줄 옵션, 2223
--show-empty
    ast 명령줄 옵션, 2200
showerror() (*tkinter.messagebox* 모듈), 1645
showinfo() (*tkinter.messagebox* 모듈), 1645
--show-offsets
    dis 명령줄 옵션, 2223
--show-positions
    dis 명령줄 옵션, 2223
showsyntaxerror() (*code.InteractiveInterpreter* 메서드), 2114
showtraceback() (*code.InteractiveInterpreter* 메서드), 2114
showturtle() (*turtle* 모듈), 1696
showwarning() (*tkinter.messagebox* 모듈), 1645
showwarning() (*warnings* 모듈), 2031
shuffle() (*random* 모듈), 399
SHUT_RD (*socket* 모듈), 1222
SHUT_RDWR (*socket* 모듈), 1222
SHUT_WR (*socket* 모듈), 1222
ShutDown, 1098
shutdown() (*asyncio.Queue* 메서드), 1152
shutdown() (*concurrent.futures.Executor* 메서드), 1064
shutdown() (*imaplib.IMAP4* 메서드), 1532
shutdown() (*logging* 모듈), 800
shutdown() (*multiprocessing.managers.BaseManager* 메서드), 1036
shutdown() (*queue.Queue* 메서드), 1100
shutdown() (*socketserver.BaseServer* 메서드), 1550
shutdown() (*socket.socket* 메서드), 1237
shutdown_asyncgens() (*asyncio.loop* 메서드), 1158
shutdown_default_executor() (*asyncio.loop* 메서드), 1158
shutil
    module, 511
SI (*curses.ascii* 모듈), 985
side_effect (*unittest.mock.Mock*의 속성), 1839
SIG_BLOCK (*signal* 모듈), 1291
SIG_DFL (*signal* 모듈), 1289
SIG_IGN (*signal* 모듈), 1289
SIG_SETMASK (*signal* 모듈), 1291
SIG_UNBLOCK (*signal* 모듈), 1291
SIGABRT (*signal* 모듈), 1289
SIGALRM (*signal* 모듈), 1289
SIGBREAK (*signal* 모듈), 1289
SIGBUS (*signal* 모듈), 1289
SIGCHLD (*signal* 모듈), 1289
SIGCLD (*signal* 모듈), 1289
SIGCONT (*signal* 모듈), 1289
SIGFPE (*signal* 모듈), 1289
SIGHUP (*signal* 모듈), 1290
SIGILL (*signal* 모듈), 1290
SIGINT (*signal* 모듈), 1290
siginterrupt() (*signal* 모듈), 1294
SIGKILL (*signal* 모듈), 1290
Sigmasks (*signal* 클래스), 1289
signal
    module, 1288
signal() (*signal* 모듈), 1294
Signals (*signal* 클래스), 1289
Signature (*inspect* 클래스), 2088
signature (*inspect.BoundArguments*의 속성), 2091
signature() (*inspect* 모듈), 2087
sigpending() (*signal* 모듈), 1294
SIGPIPE (*signal* 모듈), 1290
SIGSEGV (*signal* 모듈), 1290
SIGSTKFLT (*signal* 모듈), 1290
SIGTERM (*signal* 모듈), 1290
sigtimedwait() (*signal* 모듈), 1295
SIGUSR1 (*signal* 모듈), 1290
SIGUSR2 (*signal* 모듈), 1290
sigwait() (*signal* 모듈), 1294
sigwaitinfo() (*signal* 모듈), 1295
SIGWINCH (*signal* 모듈), 1290
SIMPLE (*inspect.BufferFlags*의 속성), 2099
Simple Mail Transfer Protocol, 1534
SimpleCookie (*http.cookies* 클래스), 1564
simplefilter() (*warnings* 모듈), 2031
SimpleHandler (*wsgiref.handlers* 클래스), 1474
SimpleHTTPRequestHandler (*http.server* 클래스), 1560
SimpleNamespace (*types* 클래스), 316
SimpleQueue (*multiprocessing* 클래스), 1025
SimpleQueue (*queue* 클래스), 1098
SimpleXMLRPCRequestHandler (*xmlrpc.server* 클래스), 1584
SimpleXMLRPCServer (*xmlrpc.server* 클래스), 1584
sin() (*cmath* 모듈), 362
sin() (*math* 모듈), 358
single dispatch (싱글 디스패치), **2315**
SingleAddressHeader (*email.headerregistry* 클래스), 1328
singledispatch() (*functools* 모듈), 446
singledispatchmethod (*functools* 클래스), 449
sinh() (*cmath* 모듈), 362
sinh() (*math* 모듈), 358
SIO_KEEPALIVE_VALS (*socket* 모듈), 1219
SIO_LOOPBACK_FAST_PATH (*socket* 모듈), 1219
SIO_RCVALL (*socket* 모듈), 1219
site
    module, 2108
site 명령줄 옵션
    --user-base, 2111
    --user-site, 2111

site_maps() (*urllib.robotparser.RobotFileParser* 메서드), 1506
sitecustomize
    module, 2110
site-packages
    디렉터리, 2109
sixtofour (*ipaddress.IPv6Address*의 속성), 1594
size (*ctypes.CField*의 속성), 877
size (*multiprocessing.shared_memory.SharedMemory*의 속성), 1059
size (*struct.Struct*의 속성), 198
size (*tarfile.TarInfo*의 속성), 631
size (*tracemalloc.StatisticDiff*의 속성), 1963
size (*tracemalloc.Statistic*의 속성), 1963
size (*tracemalloc.Trace*의 속성), 1963
size() (*ftplib.FTP* 메서드), 1522
size() (*mmap.mmap* 메서드), 1301
size_diff (*tracemalloc.StatisticDiff*의 속성), 1963
Sized (*collections.abc* 클래스), 290
Sized (*typing* 클래스), 1771
sizeof() (*ctypes* 모듈), 870
sizeof_digit (*sys.int_info*의 속성), 1996
SKIP (*doctest* 모듈), 1785
skip() (*unittest* 모듈), 1808
skip_if_broken_multiprocessing_synchronize() (*test.support* 모듈), 1904
skip_unless_bind_unix_socket() (*test.support.socket_helper* 모듈), 1906
skip_unless_symlink() (*test.support.os_helper* 모듈), 1910
skip_unless_xattr() (*test.support.os_helper* 모듈), 1910
skipIf() (*unittest* 모듈), 1808
skipinitialspace (*csv.Dialect*의 속성), 646
skipped (*doctest.TestResults*의 속성), 1794
skipped (*unittest.TestResult*의 속성), 1825
skippedEntity() (*xml.sax.handler.ContentHandler* 메서드), 1448
skips (*doctest.DocTestRunner*의 속성), 1796
SkipTest, 1808
skipTest() (*unittest.TestCase* 메서드), 1810
skipUnless() (*unittest* 모듈), 1808
SLASH (*token* 모듈), 2207
SLASHEQUAL (*token* 모듈), 2208
sleep() (*asyncio* 모듈), 1119
sleep() (*time* 모듈), 775
sleeping_retry() (*test.support* 모듈), 1898
slice
    assignment, 49
    built-in function, 2241
    operation, 47
Slice (*ast* 클래스), 2173
slice (내장 클래스), 29
slice (슬라이스), **2316**
slow_callback_duration (*asyncio.loop*의 속성), 1173
SMALLEST (*test.support* 모듈), 1898
SMTP
    프로토콜, 1534
SMTP (*email.policy* 모듈), 1323
SMTP (*smtplib* 클래스), 1535
SMTP_SSL (*smtplib* 클래스), 1535
SMTPAuthenticationError, 1536
SMTPConnectError, 1536
smtpd
    module, 2296
SMTPDataError, 1536
SMTPException, 1536
SMTPHandler (*logging.handlers* 클래스), 824
SMTPHeloError, 1536
smtplib
    module, 1534
SMTPNotSupportedError, 1536
SMTPRecipientsRefused, 1536
SMTPResponseException, 1536
SMTPSenderRefused, 1536
SMTPServerDisconnected, 1536
SMTPUTF8 (*email.policy* 모듈), 1323
Snapshot (*tracemalloc* 클래스), 1961
SND_ALIAS (*winsound* 모듈), 2259
SND_APPLICATION (*winsound* 모듈), 2260
SND_ASYNC (*winsound* 모듈), 2260
SND_FILENAME (*winsound* 모듈), 2259
SND_LOOP (*winsound* 모듈), 2259
SND_MEMORY (*winsound* 모듈), 2259
SND_NODEFAULT (*winsound* 모듈), 2260
SND_NOSTOP (*winsound* 모듈), 2260
SND_NOWAIT (*winsound* 모듈), 2260
SND_PURGE (*winsound* 모듈), 2259
SND_SENTRY (*winsound* 모듈), 2260
SND_SYNC (*winsound* 모듈), 2260
SND_SYSTEM (*winsound* 모듈), 2260
sndhdr
    module, 2296
sni_callback (*ssl.SSLContext*의 속성), 1262
sniff() (*csv.Sniffer* 메서드), 644
Sniffer (*csv* 클래스), 644
SO (*curses.ascii* 모듈), 985
SO_HCI_EVT_FILTER (*socket* 모듈), 1221
SO_HCI_PKT_FILTER (*socket* 모듈), 1221
SO_INCOMING_CPU (*socket* 모듈), 1221
SO_REUSEPORT_LB (*socket* 모듈), 1221
sock_accept() (*asyncio.loop* 메서드), 1169
SOCK_CLOEXEC (*socket* 모듈), 1217
sock_connect() (*asyncio.loop* 메서드), 1169
SOCK_DGRAM (*socket* 모듈), 1216
SOCK_MAX_SIZE (*test.support* 모듈), 1897
SOCK_NONBLOCK (*socket* 모듈), 1217
SOCK_RAW (*socket* 모듈), 1216
SOCK_RDM (*socket* 모듈), 1216
sock_recv() (*asyncio.loop* 메서드), 1168
sock_recv_into() (*asyncio.loop* 메서드), 1168
sock_recvfrom() (*asyncio.loop* 메서드), 1168
sock_recvfrom_into() (*asyncio.loop* 메서드), 1168
sock_sendall() (*asyncio.loop* 메서드), 1168
sock_sendfile() (*asyncio.loop* 메서드), 1169

sock_sendto() (*asyncio.loop* 메서드), 1169
SOCK_SEQPACKET (*socket* 모듈), 1216
SOCK_STREAM (*socket* 모듈), 1216
socket
    module, 1212
    object, 1212
socket (*socket* 클래스), 1222
socket (*socketserver.BaseServer*의 속성), 1550
socket() (*imaplib.IMAP4* 메서드), 1532
socket() (*socket* 모듈), 1278
socket_type (*socketserver.BaseServer*의 속성), 1550
SocketHandler (*logging.handlers* 클래스), 819
socketpair() (*socket* 모듈), 1223
sockets (*asyncio.Server*의 속성), 1177
socketserver
    module, 1547
SocketType (*socket* 모듈), 1224
soft deprecated (약하게 폐지된), **2316**
SOFT_KEYWORD (*token* 모듈), 2206
softkwlist (*keyword* 모듈), 2210
SOH (*curses.ascii* 모듈), 984
SOL_ALG (*socket* 모듈), 1219
SOL_BLUETOOTH (*socket* 모듈), 1220
SOL_HCI (*socket* 모듈), 1220
SOL_L2CAP (*socket* 모듈), 1220
SOL_RDS (*socket* 모듈), 1219
SOL_RFCOMM (*socket* 모듈), 1220
SOL_SCO (*socket* 모듈), 1220
SOMAXCONN (*socket* 모듈), 1217
sort() (*imaplib.IMAP4* 메서드), 1533
sort() (*list* 메서드), 50
sort_stats() (*pstats.Stats* 메서드), 1943
sortdict() (*test.support* 모듈), 1899
sorted()
    built-in function, 29
--sort-keys
    json 명령줄 옵션, 1370
sortTestMethodsUsing (*unittest.TestLoader*의 속성), 1824
source (*doctest.Example*의 속성), 1792
source (*pdb command*), 1935
source (*shlex.shlex*의 속성), 2266
SOURCE_DATE_EPOCH, 2217, 2220
source_from_cache() (*importlib.util* 모듈), 2141
source_hash() (*importlib.util* 모듈), 2142
SOURCE_SUFFIXES (*importlib.machinery* 모듈), 2135
source_to_code() (*importlib.abc.InspectLoader*의 정적 메서드), 2131
SourceFileLoader (*importlib.machinery* 클래스), 2137
sourcehook() (*shlex.shlex* 메서드), 2265
SourcelessFileLoader (*importlib.machinery* 클래스), 2137
SourceLoader (*importlib.abc* 클래스), 2132
SP (*curses.ascii* 모듈), 986
space
    in printf-style formatting, 66, 82
--spacing
    calendar 명령줄 옵션, 269
span() (*re.Match* 메서드), 158
sparse (*tarfile.TarInfo*의 속성), 632
spawn() (*pty* 모듈), 2274
spawn_python() (*test.support.script_helper* 모듈), 1907
spawnl() (*os* 모듈), 745
spawnle() (*os* 모듈), 745
spawnlp() (*os* 모듈), 745
spawnlpe() (*os* 모듈), 745
spawnv() (*os* 모듈), 745
spawnve() (*os* 모듈), 745
spawnvp() (*os* 모듈), 745
spawnvpe() (*os* 모듈), 745
spec_from_file_location() (*importlib.util* 모듈), 2142
spec_from_loader() (*importlib.util* 모듈), 2141
special
    method (메서드), 2316
special method (특수 메서드), **2316**
SpecialFileError, 512, 626
--specialized
    dis 명령줄 옵션, 2223
specified_attributes (*xml.parsers.expat.xmlparser*의 속성), 1457
speed() (*turtle* 모듈), 1689
Spinbox (*tkinter.ttk* 클래스), 1653
splice() (*os* 모듈), 709
SPLICE_F_MORE (*os* 모듈), 710
SPLICE_F_MOVE (*os* 모듈), 710
SPLICE_F_NONBLOCK (*os* 모듈), 710
split() (*BaseExceptionGroup* 메서드), 121
split() (*bytearray* 메서드), 76
split() (*bytes* 메서드), 76
split() (*os.path* 모듈), 491
split() (*re* 모듈), 152
split() (*re.Pattern* 메서드), 156
split() (*shlex* 모듈), 2263
split() (*str* 메서드), 61
splitdrive() (*os.path* 모듈), 491
splitext() (*os.path* 모듈), 492
splitlines() (*bytearray* 메서드), 79
splitlines() (*bytes* 메서드), 79
splitlines() (*str* 메서드), 62
SplitResult (*urllib.parse* 클래스), 1502
SplitResultBytes (*urllib.parse* 클래스), 1502
splitroot() (*os.path* 모듈), 492
SpooledTemporaryFile (*tempfile* 클래스), 502
sprintf-style formatting, 66, 81
spwd
    module, 2296
sqlite3
    module, 551
SQLITE_DBCONFIG_DEFENSIVE (*sqlite3* 모듈), 556
SQLITE_DBCONFIG_DQS_DDL (*sqlite3* 모듈), 556
SQLITE_DBCONFIG_DQS_DML (*sqlite3* 모듈), 556
SQLITE_DBCONFIG_ENABLE_FKEY (*sqlite3* 모듈), 556

SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER (*sqlite3* 모듈), 556
SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION (*sqlite3* 모듈), 556
SQLITE_DBCONFIG_ENABLE_QPSG (*sqlite3* 모듈), 556
SQLITE_DBCONFIG_ENABLE_TRIGGER (*sqlite3* 모듈), 556
SQLITE_DBCONFIG_ENABLE_VIEW (*sqlite3* 모듈), 556
SQLITE_DBCONFIG_LEGACY_ALTER_TABLE (*sqlite3* 모듈), 556
SQLITE_DBCONFIG_LEGACY_FILE_FORMAT (*sqlite3* 모듈), 556
SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE (*sqlite3* 모듈), 556
SQLITE_DBCONFIG_RESET_DATABASE (*sqlite3* 모듈), 556
SQLITE_DBCONFIG_TRIGGER_EQP (*sqlite3* 모듈), 556
SQLITE_DBCONFIG_TRUSTED_SCHEMA (*sqlite3* 모듈), 556
SQLITE_DBCONFIG_WRITABLE_SCHEMA (*sqlite3* 모듈), 556
SQLITE_DENY (*sqlite3* 모듈), 556
sqlite_errorcode (*sqlite3.Error*의 속성), 572
sqlite_errorname (*sqlite3.Error*의 속성), 572
SQLITE_IGNORE (*sqlite3* 모듈), 556
SQLITE_OK (*sqlite3* 모듈), 556
sqlite_version (*sqlite3* 모듈), 556
sqlite_version_info (*sqlite3* 모듈), 556
sqrt() (*cmath* 모듈), 362
sqrt() (*decimal.Context* 메서드), 382
sqrt() (*decimal.Decimal* 메서드), 375
sqrt() (*math* 모듈), 356
SSL, 1241
ssl
    module, 1241
ssl_version (*ftplib.FTP_TLS*의 속성), 1523
SSLCertVerificationError, 1245
SSLContext (*ssl* 클래스), 1258
SSLEOFError, 1245
SSLError, 1244
SSLErrorNumber (*ssl* 클래스), 1253
SSLKEYLOGFILE, 1243, 1244
SSLObject (*ssl* 클래스), 1273
sslobject_class (*ssl.SSLContext*의 속성), 1264
SSLSession (*ssl* 클래스), 1275
SSLSocket (*ssl* 클래스), 1254
sslsocket_class (*ssl.SSLContext*의 속성), 1263
SSLSyscallError, 1245
SSLv3 (*ssl.TLSVersion*의 속성), 1254
SSLWantReadError, 1244
SSLWantWriteError, 1245
SSLZeroReturnError, 1244
st() (*turtle* 모듈), 1696
st_atime (*os.stat_result*의 속성), 725
ST_ATIME (*stat* 모듈), 495
st_atime_ns (*os.stat_result*의 속성), 726
st_birthtime (*os.stat_result*의 속성), 726
st_birthtime_ns (*os.stat_result*의 속성), 726
st_blksize (*os.stat_result*의 속성), 726
st_blocks (*os.stat_result*의 속성), 726
st_creator (*os.stat_result*의 속성), 727
st_ctime (*os.stat_result*의 속성), 725
ST_CTIME (*stat* 모듈), 495
st_ctime_ns (*os.stat_result*의 속성), 726
st_dev (*os.stat_result*의 속성), 725
ST_DEV (*stat* 모듈), 494
st_file_attributes (*os.stat_result*의 속성), 727
st_flags (*os.stat_result*의 속성), 726
st_fstype (*os.stat_result*의 속성), 727
st_gen (*os.stat_result*의 속성), 727
st_gid (*os.stat_result*의 속성), 725
ST_GID (*stat* 모듈), 494
st_ino (*os.stat_result*의 속성), 725
ST_INO (*stat* 모듈), 494
st_mode (*os.stat_result*의 속성), 725
ST_MODE (*stat* 모듈), 494
st_mtime (*os.stat_result*의 속성), 725
ST_MTIME (*stat* 모듈), 495
st_mtime_ns (*os.stat_result*의 속성), 726
st_nlink (*os.stat_result*의 속성), 725
ST_NLINK (*stat* 모듈), 494
st_rdev (*os.stat_result*의 속성), 726
st_reparse_tag (*os.stat_result*의 속성), 727
st_rsize (*os.stat_result*의 속성), 727
st_size (*os.stat_result*의 속성), 725
ST_SIZE (*stat* 모듈), 495
st_type (*os.stat_result*의 속성), 727
st_uid (*os.stat_result*의 속성), 725
ST_UID (*stat* 모듈), 494
stack (*traceback.TracebackException*의 속성), 2068
stack() (*inspect* 모듈), 2096
stack_effect() (*dis* 모듈), 2227
stack_size() (*_thread* 모듈), 1106
stack_size() (*threading* 모듈), 998
stackable
    스트림, 198
StackSummary (*traceback* 클래스), 2069
stamp() (*turtle* 모듈), 1688
standard library, **2316**
standard_b64decode() (*base64* 모듈), 1395
standard_b64encode() (*base64* 모듈), 1395
standend() (*curses.window* 메서드), 970
standout() (*curses.window* 메서드), 970
STAR (*token* 모듈), 2207
STAREQUAL (*token* 모듈), 2208
starmap() (*itertools* 모듈), 432
starmap() (*multiprocessing.pool.Pool* 메서드), 1043
starmap_async() (*multiprocessing.pool.Pool* 메서드), 1043
Starred (*ast* 클래스), 2170
start (*range*의 속성), 52
start (*slice*의 속성), 29
start (*UnicodeError*의 속성), 118
start() (*logging.handlers.QueueListener* 메서드), 828
start() (*multiprocessing.managers.BaseManager* 메서드), 1035

start() (*multiprocessing.Process* 메서드), 1020
start() (*re.Match* 메서드), 158
start() (*threading.Thread* 메서드), 1002
start() (*tkinter.ttk.Progressbar* 메서드), 1656
start() (*tracemalloc* 모듈), 1960
start() (*xml.etree.ElementTree.TreeBuilder* 메서드), 1423
start_color() (*curses* 모듈), 963
start_new_thread() (*_thread* 모듈), 1106
start_ns() (*xml.etree.ElementTree.TreeBuilder* 메서드), 1424
start_server() (*asyncio* 모듈), 1133
start_serving() (*asyncio.Server* 메서드), 1177
start_threads() (*test.support.threading_helper* 모듈), 1908
start_tls() (*asyncio.loop* 메서드), 1167
start_tls() (*asyncio.StreamWriter* 메서드), 1137
start_trace() (*bdb.Bdb* 메서드), 1921
start_unix_server() (*asyncio* 모듈), 1134
StartBoundaryNotFoundDefect, 1325
startCDATA() (*xml.sax.handler.LexicalHandler* 메서드), 1449
StartCdataSectionHandler() (*xml.parsers.expat.xmlparser* 메서드), 1459
--start-directory
    unittest-discover 명령줄 옵션, 1804
StartDoctypeDeclHandler() (*xml.parsers.expat.xmlparser* 메서드), 1458
startDocument() (*xml.sax.handler.ContentHandler* 메서드), 1446
startDTD() (*xml.sax.handler.LexicalHandler* 메서드), 1449
startElement() (*xml.sax.handler.ContentHandler* 메서드), 1447
StartElementHandler() (*xml.parsers.expat.xmlparser* 메서드), 1458
startElementNS() (*xml.sax.handler.ContentHandler* 메서드), 1447
STARTF_FORCEOFFFEEDBACK (*subprocess* 모듈), 1089
STARTF_FORCEONFEEDBACK (*subprocess* 모듈), 1089
STARTF_USESHOWWINDOW (*subprocess* 모듈), 1089
STARTF_USESTDHANDLES (*subprocess* 모듈), 1089
startfile() (*os* 모듈), 747
StartNamespaceDeclHandler() (*xml.parsers.expat.xmlparser* 메서드), 1459
startPrefixMapping() (*xml.sax.handler.ContentHandler* 메서드), 1446
StartResponse (*wsgiref.types* 클래스), 1476
startswith() (*bytearray* 메서드), 74
startswith() (*bytes* 메서드), 74
startswith() (*str* 메서드), 62
startTest() (*unittest.TestResult* 메서드), 1826
startTestRun() (*unittest.TestResult* 메서드), 1826
starttls() (*imaplib.IMAP4* 메서드), 1533
starttls() (*smtplib.SMTP* 메서드), 1538
STARTUPINFO (*subprocess* 클래스), 1088
stat
    module, 493, 724
stat() (*os* 모듈), 724
stat() (*os.DirEntry* 메서드), 724
stat() (*pathlib.Path* 메서드), 474
stat() (*poplib.POP3* 메서드), 1525
stat_result (*os* 클래스), 725
state() (*tkinter.ttk.Widget* 메서드), 1651
statement
    del, 49, 93
    if, 37
    import, 33
    while, 37
statement (문장), **2316**
static type checker (정적 형 검사기), **2316**
static_order() (*graphlib.TopologicalSorter* 메서드), 345
staticmethod()
    built-in function, 29
Statistic (*tracemalloc* 클래스), 1962
StatisticDiff (*tracemalloc* 클래스), 1963
statistics
    module, 406
statistics() (*tracemalloc.Snapshot* 메서드), 1962
StatisticsError, 417
Stats (*pstats* 클래스), 1943
status (*http.client.HTTPResponse*의 속성), 1515
status (*urllib.response.addinfourl*의 속성), 1495
status() (*imaplib.IMAP4* 메서드), 1533
statvfs() (*os* 모듈), 727
STD_ERROR_HANDLE (*subprocess* 모듈), 1089
STD_INPUT_HANDLE (*subprocess* 모듈), 1089
STD_OUTPUT_HANDLE (*subprocess* 모듈), 1089
stderr (*asyncio.subprocess.Process*의 속성), 1149
stderr (*subprocess.CalledProcessError*의 속성), 1080
stderr (*subprocess.CompletedProcess*의 속성), 1078
stderr (*subprocess.Popen*의 속성), 1087
stderr (*subprocess.TimeoutExpired*의 속성), 1079
stderr (*sys* 모듈), 2005
stdev (*statistics.NormalDist*의 속성), 417
stdev() (*statistics* 모듈), 414
stdin (*asyncio.subprocess.Process*의 속성), 1149
stdin (*subprocess.Popen*의 속성), 1087
stdin (*sys* 모듈), 2005
stdlib, **2316**
stdlib_module_names (*sys* 모듈), 2006
stdout (*asyncio.subprocess.Process*의 속성), 1149
STDOUT (*subprocess* 모듈), 1079
stdout (*subprocess.CalledProcessError*의 속성), 1079
stdout (*subprocess.CompletedProcess*의 속성), 1078
stdout (*subprocess.Popen*의 속성), 1087
stdout (*subprocess.TimeoutExpired*의 속성), 1079
stdout (*sys* 모듈), 2005
stem (*pathlib.PurePath*의 속성), 466
step (*pdb command*), 1934
step (*range*의 속성), 52
step (*slice*의 속성), 29
step() (*tkinter.ttk.Progressbar* 메서드), 1656
stls() (*poplib.POP3* 메서드), 1526

stop (*range*의 속성), 52
stop (*slice*의 속성), 29
stop() (*asyncio.loop* 메서드), 1158
stop() (*logging.handlers.QueueListener* 메서드), 828
stop() (*tkinter.ttk.Progressbar* 메서드), 1656
stop() (*tracemalloc* 모듈), 1960
stop() (*unittest.TestResult* 메서드), 1825
stop_here() (*bdb.Bdb* 메서드), 1922
STOP_ITERATION (*monitoring event*), 2010
stop_trace() (*bdb.Bdb* 메서드), 1921
StopAsyncIteration, 116
StopIteration, 116
stopListening() (*logging.config* 모듈), 804
stopTest() (*unittest.TestResult* 메서드), 1826
stopTestRun() (*unittest.TestResult* 메서드), 1826
storbinary() (*ftplib.FTP* 메서드), 1520
Store (*ast* 클래스), 2169
store() (*imaplib.IMAP4* 메서드), 1533
STORE_ACTIONS (*optparse.Option*의 속성), 952
STORE_ATTR (*opcode*), 2234
STORE_DEREF (*opcode*), 2239
STORE_FAST (*opcode*), 2238
STORE_FAST_LOAD_FAST (*opcode*), 2239
STORE_FAST_STORE_FAST (*opcode*), 2238
STORE_GLOBAL (*opcode*), 2234
STORE_NAME (*opcode*), 2233
STORE_SLICE (*opcode*), 2230
STORE_SUBSCR (*opcode*), 2230
storlines() (*ftplib.FTP* 메서드), 1520
str (*built-in class*)
    (see also string), 53
str (내장 클래스), 54
str() (*locale* 모듈), 1622
str_digits_check_threshold (*sys.int_info*의 속성), 1996
Strategy (*compression.zstd* 클래스), 592
strategy (*compression.zstd.CompressionParameter*의 속성), 590
strcoll() (*locale* 모듈), 1622
StreamError, 625
StreamHandler (*logging* 클래스), 814
StreamReader (*asyncio* 클래스), 1135
StreamReader (*codecs* 클래스), 206
streamreader (*codecs.CodecInfo*의 속성), 199
StreamReaderWriter (*codecs* 클래스), 207
StreamRecoder (*codecs* 클래스), 207
StreamRequestHandler (*socketserver* 클래스), 1552
StreamWriter (*asyncio* 클래스), 1136
StreamWriter (*codecs* 클래스), 206
streamwriter (*codecs.CodecInfo*의 속성), 199
StrEnum (*enum* 클래스), 335
strerror (*OSError*의 속성), 115
strerror() (*os* 모듈), 697
strftime() (*datetime.date* 메서드), 227
strftime() (*datetime.datetime* 메서드), 238
strftime() (*datetime.time* 메서드), 243
strftime() (*time* 모듈), 776
strict
    에러 처리기의 이름, 202
strict (*csv.Dialect*의 속성), 646
strict (*email.policy* 모듈), 1324
STRICT (*enum.FlagBoundary*의 속성), 339
strict_domain (*http.cookiejar.DefaultCookiePolicy*의 속성), 1573
strict_errors() (*codecs* 모듈), 203
strict_ns_domain (*http.cookiejar.DefaultCookiePolicy*의 속성), 1573
strict_ns_set_initial_dollar (*http.cookiejar.DefaultCookiePolicy*의 속성), 1573
strict_ns_set_path (*http.cookiejar.DefaultCookiePolicy*의 속성), 1573
strict_ns_unverifiable (*http.cookiejar.DefaultCookiePolicy*의 속성), 1573
strict_rfc2965_unverifiable (*http.cookiejar.DefaultCookiePolicy*의 속성), 1573
STRIDED (*inspect.BufferFlags*의 속성), 2100
STRIDED_RO (*inspect.BufferFlags*의 속성), 2100
STRIDES (*inspect.BufferFlags*의 속성), 2099
strides (*memoryview*의 속성), 90
string
    format() (*built-in function*), 17
    formatted literal, 64
    formatting, printf, 66
    interpolated literal, 64
    interpolation, printf, 66
    methods, 54
    module, 125
    object, 53
    str (*built-in class*), 54
    str() (*built-in function*), 30
    text sequence type, 53
STRING (*annotationlib.Format*의 속성), 2102
string (*re.Match*의 속성), 159
STRING (*token* 모듈), 2205
string_at() (*ctypes* 모듈), 870
StringIO (*io* 클래스), 769
stringprep
    module, 183
strings (*string.templatelib.Template*의 속성), 138
string.templatelib
    module, 136
strip() (*bytearray* 메서드), 76
strip() (*bytes* 메서드), 76
strip() (*str* 메서드), 62
strip_dirs() (*pstats.Stats* 메서드), 1943
stripspaces (*curses.textpad.Textbox*의 속성), 983
strong reference (강한 참조), **2316**
strptime() (*datetime.datetime*의 클래스 메서드), 232
strptime() (*datetime.date*의 클래스 메서드), 224
strptime() (*datetime.time*의 클래스 메서드), 242
strptime() (*time* 모듈), 778

strsignal() (*signal* 모듈), 1292
struct
    module, 191, 1236
Struct (*struct* 클래스), 198
struct_time (*time* 클래스), 778
Structure (*ctypes* 클래스), 875
strxfrm() (*locale* 모듈), 1622
STX (*curses.ascii* 모듈), 984
Style (*tkinter.ttk* 클래스), 1662
Sub (*ast* 클래스), 2171
SUB (*curses.ascii* 모듈), 985
sub() (*operator* 모듈), 453
sub() (*re* 모듈), 153
sub() (*re.Pattern* 메서드), 156
subdirs (*filecmp.dircmp*의 속성), 500
SubElement() (*xml.etree.ElementTree* 모듈), 1416
subgroup() (*BaseExceptionGroup* 메서드), 121
submit() (*concurrent.futures.Executor* 메서드), 1064
submodule_search_locations (*importlib.machinery.ModuleSpec* 의 속성), 2139
subn() (*re* 모듈), 154
subn() (*re.Pattern* 메서드), 156
subnet_of() (*ipaddress.IPv4Network* 메서드), 1598
subnet_of() (*ipaddress.IPv6Network* 메서드), 1600
subnets() (*ipaddress.IPv4Network* 메서드), 1597
subnets() (*ipaddress.IPv6Network* 메서드), 1600
Subnormal (*decimal* 클래스), 384
suboffsets (*memoryview*의 속성), 91
subpad() (*curses.window* 메서드), 970
subprocess
    module, 1077
subprocess_exec() (*asyncio.loop* 메서드), 1174
subprocess_shell() (*asyncio.loop* 메서드), 1175
SubprocessError, 1079
SubprocessProtocol (*asyncio* 클래스), 1189
SubprocessTransport (*asyncio* 클래스), 1185
subscribe() (*imaplib.IMAP4* 메서드), 1533
subscript
    assignment, 49
    operation, 47
Subscript (*ast* 클래스), 2173
subsequent_indent (*textwrap.TextWrapper*의 속성), 179
substitute() (*string.Template* 메서드), 135
subTest() (*unittest.TestCase* 메서드), 1810
subtract() (*collections.Counter* 메서드), 273
subtract() (*decimal.Context* 메서드), 382
subtype (*email.headerregistry.ContentTypeHeader* 의 속성), 1329
subwin() (*curses.window* 메서드), 970
successful() (*multiprocessing.pool.AsyncResult* 메서드), 1044
suffix (*pathlib.PurePath*의 속성), 466
suffix_map (*mimetypes* 모듈), 1391
suffix_map (*mimetypes.MimeTypes*의 속성), 1392
suffixes (*pathlib.PurePath*의 속성), 466
suiteClass (*unittest.TestLoader*의 속성), 1824
sum()
    built-in function, 30
summarize() (*doctest.DocTestRunner* 메서드), 1795
summarize_address_range() (*ipaddress* 모듈), 1602
--summary
    trace 명령줄 옵션, 1953
sumprod() (*math* 모듈), 357
sunau
    module, 2296
SUNDAY (*calendar* 모듈), 266
super (*pyclbr.Class*의 속성), 2216
super (내장 클래스), 30
supernet() (*ipaddress.IPv4Network* 메서드), 1598
supernet() (*ipaddress.IPv6Network* 메서드), 1600
supernet_of() (*ipaddress.IPv4Network* 메서드), 1598
supernet_of() (*ipaddress.IPv6Network* 메서드), 1600
supports_bytes_environ (*os* 모듈), 697
supports_dir_fd (*os* 모듈), 728
supports_effective_ids (*os* 모듈), 728
supports_fd (*os* 모듈), 728
supports_follow_symlinks (*os* 모듈), 729
supports_unicode_filenames (*os.path* 모듈), 493
SupportsAbs (*typing* 클래스), 1757
SupportsBytes (*typing* 클래스), 1757
SupportsComplex (*typing* 클래스), 1757
SupportsFloat (*typing* 클래스), 1757
SupportsIndex (*typing* 클래스), 1757
SupportsInt (*typing* 클래스), 1757
SupportsRound (*typing* 클래스), 1757
suppress() (*contextlib* 모듈), 2048
SuppressCrashReport (*test.support* 클래스), 1905
surrogateescape
    에러 처리기의 이름, 202
surrogatepass
    에러 처리기의 이름, 202
SW_HIDE (*subprocess* 모듈), 1089
SWAP (*opcode*), 2229
swap_attr() (*test.support* 모듈), 1900
swap_item() (*test.support* 모듈), 1900
swapcase() (*bytearray* 메서드), 80
swapcase() (*bytes* 메서드), 80
swapcase() (*str* 메서드), 63
Symbol (*symtable* 클래스), 2203
SymbolTable (*symtable* 클래스), 2201
SymbolTableType (*symtable* 클래스), 2201
symlink() (*os* 모듈), 729
symlink_to() (*pathlib.Path* 메서드), 480
--symlinks
    venv 명령줄 옵션, 1968
symmetric_difference() (*frozenset* 메서드), 92
symmetric_difference_update() (*frozenset* 메서드), 93
symtable
    module, 2200
symtable() (*symtable* 모듈), 2201

SYMTYPE (*tarfile* 모듈), 626
SYN (*curses.ascii* 모듈), 985
sync() (*dbm.dumb.dumbdbm* 메서드), 550
sync() (*dbm.gnu.gdbm* 메서드), 548
sync() (*os* 모듈), 730
sync() (*shelve.Shelf* 메서드), 541
syncdown() (*curses.window* 메서드), 970
synchronized() (*multiprocessing.sharedctypes* 모듈), 1034
SyncManager (*multiprocessing.managers* 클래스), 1036
syncok() (*curses.window* 메서드), 970
syncup() (*curses.window* 메서드), 970
SyntaxErr, 1435
SyntaxError, 116
SyntaxWarning, 120
sys
    module, 25, 1981
sys_version (*http.server.BaseHTTPRequestHandler*의 속성), 1558
sysconf() (*os* 모듈), 754
sysconf_names (*os* 모듈), 754
sysconfig
    module, 2013
syslog
    module, 2282
syslog() (*syslog* 모듈), 2282
SysLogHandler (*logging.handlers* 클래스), 821
sys.monitoring
    module, 2008
system() (*os* 모듈), 747
system() (*platform* 모듈), 830
system_alias() (*platform* 모듈), 830
system_must_validate_cert() (*test.support* 모듈), 1901
SystemError, 117
SystemExit, 117
systemId (*xml.dom.DocumentType*의 속성), 1431
SystemRandom (*random* 클래스), 401
SystemRandom (*secrets* 클래스), 687
SystemRoot, 1084
--system-site-packages
    venv 명령줄 옵션, 1968

T

-T
    trace 명령줄 옵션, 1952
-t
    calendar 명령줄 옵션, 268
    idle 명령줄 옵션, 1674
    tarfile 명령줄 옵션, 637
    trace 명령줄 옵션, 1952
    unittest-discover 명령줄 옵션, 1804
    webbrowser 명령줄 옵션, 1466
    zipfile 명령줄 옵션, 622
t-string, **2316**
t-strings, **2316**
T_FMT (*locale* 모듈), 1619
T_FMT_AMPM (*locale* 모듈), 1619
--tab
    json 명령줄 옵션, 1370
TAB (*curses.ascii* 모듈), 984
tab() (*tkinter.ttk.Notebook* 메서드), 1655
TabError, 117
tabnanny
    module, 2214
tabs() (*tkinter.ttk.Notebook* 메서드), 1655
tabsize (*textwrap.TextWrapper*의 속성), 179
tag (*xml.etree.ElementTree.Element*의 속성), 1419
tag_bind() (*tkinter.ttk.Treeview* 메서드), 1661
tag_configure() (*tkinter.ttk.Treeview* 메서드), 1661
tag_has() (*tkinter.ttk.Treeview* 메서드), 1661
tagName (*xml.dom.Element*의 속성), 1432
tail (*xml.etree.ElementTree.Element*의 속성), 1419
take_snapshot() (*tracemalloc* 모듈), 1960
takewhile() (*itertools* 모듈), 432
tan() (*cmath* 모듈), 362
tan() (*math* 모듈), 358
tanh() (*cmath* 모듈), 362
tanh() (*math* 모듈), 358
tar_filter() (*tarfile* 모듈), 634
TarError, 625
tarfile
    module, 623
TarFile (*tarfile* 클래스), 627
tarfile 명령줄 옵션
    -c, 637
    --create, 637
    -e, 637
    --extract, 637
    --filter, 637
    -l, 637
    --list, 637
    -t, 637
    --test, 637
    -v, 637
    --verbose, 637
target (*xml.dom.ProcessingInstruction*의 속성), 1434
target_length (*compression.zstd.CompressionParameter* 의 속성), 590
TarInfo (*tarfile* 클래스), 631
tarinfo (*tarfile.FilterError*의 속성), 625
Task (*asyncio* 클래스), 1129
task_done() (*asyncio.Queue* 메서드), 1152
task_done() (*multiprocessing.JoinableQueue* 메서드), 1026
task_done() (*queue.Queue* 메서드), 1099
TaskGroup (*asyncio* 클래스), 1117
tau (*cmath* 모듈), 363
tau (*math* 모듈), 359
tb_locals (*unittest.TestResult*의 속성), 1825
tbreak (*pdb command*), 1933
tcdrain() (*termios* 모듈), 2271
tcflow() (*termios* 모듈), 2272
tcflush() (*termios* 모듈), 2271

tcgetattr() (*termios* 모듈), 2271
tcgetpgrp() (*os* 모듈), 710
tcgetwinsize() (*termios* 모듈), 2272
Tcl() (*tkinter* 모듈), 1629
TCPServer (*socketserver* 클래스), 1547
TCSADRAIN (*termios* 모듈), 2271
TCSAFLUSH (*termios* 모듈), 2271
TCSANOW (*termios* 모듈), 2271
tcsendbreak() (*termios* 모듈), 2271
tcsetattr() (*termios* 모듈), 2271
tcsetpgrp() (*os* 모듈), 710
tcsetwinsize() (*termios* 모듈), 2272
tearDown() (*unittest.TestCase* 메서드), 1810
tearDownClass() (*unittest.TestCase* 메서드), 1810
tearDownModule() (*unittest* 모듈), 1830
tee() (*itertools* 모듈), 433
teleport() (*turtle* 모듈), 1686
tell() (*io.IOBase* 메서드), 761
tell() (*io.TextIOBase* 메서드), 767
tell() (*io.TextIOWrapper* 메서드), 769
tell() (*mmap.mmap* 메서드), 1301
tell() (*sqlite3.Blob* 메서드), 571
tell() (*wave.Wave_read* 메서드), 1606
tell() (*wave.Wave_write* 메서드), 1607
telnetlib
    module, 2297
TEMP, 504
temp_cwd() (*test.support.os_helper* 모듈), 1910
temp_dir() (*test.support.os_helper* 모듈), 1910
temp_umask() (*test.support.os_helper* 모듈), 1910
tempdir (*tempfile* 모듈), 505
tempfile
    module, 501
Template (*string* 클래스), 134
Template (*string.templatelib* 클래스), 137
template (*string.Template*의 속성), 135
TemplateStr (*ast* 클래스), 2168
temporary (*bdb.Breakpoint*의 속성), 1920
TemporaryDirectory (*tempfile* 클래스), 503
TemporaryFile() (*tempfile* 모듈), 501
teredo (*ipaddress.IPv6Address*의 속성), 1594
TERM, 963
termattrs() (*curses* 모듈), 963
terminal_size (*os* 클래스), 711
terminate() (*asyncio.subprocess.Process* 메서드), 1149
terminate() (*asyncio.SubprocessTransport* 메서드), 1188
terminate() (*multiprocessing.pool.Pool* 메서드), 1043
terminate() (*multiprocessing.Process* 메서드), 1022
terminate() (*subprocess.Popen* 메서드), 1087
terminate_workers() (*concurrent.futures.ProcessPoolExecutor* 메서드), 1068
terminator (*logging.StreamHandler*의 속성), 815
termios
    module, 2271
termname() (*curses* 모듈), 963
--terse
    platform 명령줄 옵션, 832
test
    module, 1893
--test
    tarfile 명령줄 옵션, 637
    zipfile 명령줄 옵션, 622
test (*doctest.DocTestFailure*의 속성), 1798
test (*doctest.UnexpectedException*의 속성), 1799
TEST_DATA_DIR (*test.support* 모듈), 1897
TEST_HOME_DIR (*test.support* 모듈), 1897
TEST_HTTP_URL (*test.support* 모듈), 1898
TEST_SUPPORT_DIR (*test.support* 모듈), 1897
TestCase (*unittest* 클래스), 1809
TestFailed, 1896
testfile() (*doctest* 모듈), 1788
TESTFN (*test.support.os_helper* 모듈), 1909
TESTFN_NONASCII (*test.support.os_helper* 모듈), 1909
TESTFN_UNDECODABLE (*test.support.os_helper* 모듈), 1909
TESTFN_UNENCODABLE (*test.support.os_helper* 모듈), 1909
TESTFN_UNICODE (*test.support.os_helper* 모듈), 1909
TestLoader (*unittest* 클래스), 1822
testMethodPrefix (*unittest.TestLoader*의 속성), 1824
testmod() (*doctest* 모듈), 1788
testNamePatterns (*unittest.TestLoader*의 속성), 1824
test.regrtest
    module, 1895
TestResult (*unittest* 클래스), 1824
TestResults (*doctest* 클래스), 1794
testsource() (*doctest* 모듈), 1797
testsRun (*unittest.TestResult*의 속성), 1825
TestSuite (*unittest* 클래스), 1821
test.support
    module, 1896
test.support.bytecode_helper
    module, 1907
test.support.import_helper
    module, 1910
test.support.os_helper
    module, 1908
test.support.script_helper
    module, 1906
test.support.socket_helper
    module, 1905
test.support.threading_helper
    module, 1907
test.support.warnings_helper
    module, 1912
testzip() (*zipfile.ZipFile* 메서드), 616
text (*ctypes.COMError*의 속성), 878
text (*SyntaxError*의 속성), 117
text (*traceback.TracebackException*의 속성), 2069
Text (*typing* 클래스), 1768

text (*xml.etree.ElementTree.Element*의 속성), 1419
text encoding (텍스트 인코딩), **2316**
text file (텍스트 파일), **2316**
text mode, 25
text_encoding() (*io* 모듈), 759
text_factory (*sqlite3.Connection*의 속성), 567
Textbox (*curses.textpad* 클래스), 982
TextCalendar (*calendar* 클래스), 262
textdomain() (*gettext* 모듈), 1609
textdomain() (*locale* 모듈), 1625
textinput() (*turtle* 모듈), 1706
TextIO (*typing* 클래스), 1757
TextIOBase (*io* 클래스), 767
TextIOWrapper (*io* 클래스), 768
TextTestResult (*unittest* 클래스), 1827
TextTestRunner (*unittest* 클래스), 1827
textwrap
    module, 177
TextWrapper (*textwrap* 클래스), 178
TFD_CLOEXEC (*os* 모듈), 737
TFD_NONBLOCK (*os* 모듈), 737
TFD_TIMER_ABSTIME (*os* 모듈), 737
TFD_TIMER_CANCEL_ON_SET (*os* 모듈), 737
theme_create() (*tkinter.ttk.Style* 메서드), 1665
theme_names() (*tkinter.ttk.Style* 메서드), 1665
theme_settings() (*tkinter.ttk.Style* 메서드), 1665
theme_use() (*tkinter.ttk.Style* 메서드), 1665
THOUSEP (*locale* 모듈), 1620
Thread (*threading* 클래스), 1001
thread state, **2317**
thread() (*imaplib.IMAP4* 메서드), 1533
thread_info (*sys* 모듈), 2006
thread_inherit_context (*sys.flags*의 속성), 1988
thread_time() (*time* 모듈), 780
thread_time_ns() (*time* 모듈), 780
thread_unsafe() (*test.support* 모듈), 1902
threading
    module, 995
threading_cleanup() (*test.support.threading_helper* 모듈), 1908
threading_setup() (*test.support.threading_helper* 모듈), 1908
ThreadingHTTPServer (*http.server* 클래스), 1556
ThreadingHTTPSServer (*http.server* 클래스), 1557
ThreadingMixIn (*socketserver* 클래스), 1548
ThreadingMock (*unittest.mock* 클래스), 1847
ThreadingTCPServer (*socketserver* 클래스), 1549
ThreadingUDPServer (*socketserver* 클래스), 1549
ThreadingUnixDatagramServer (*socketserver* 클래스), 1549
ThreadingUnixStreamServer (*socketserver* 클래스), 1549
ThreadPool (*multiprocessing.pool* 클래스), 1048
ThreadPoolExecutor (*concurrent.futures* 클래스), 1065
threadsafety (*sqlite3* 모듈), 556
THURSDAY (*calendar* 모듈), 266
ticket_lifetime_hint (*ssl.SSLSession*의 속성), 1275
tigetflag() (*curses* 모듈), 963
tigetnum() (*curses* 모듈), 963
tigetstr() (*curses* 모듈), 963
TILDE (*token* 모듈), 2208
tilt() (*turtle* 모듈), 1698
tiltangle() (*turtle* 모듈), 1698
time
    module, 771
time (*datetime* 클래스), 240
time (*ssl.SSLSession*의 속성), 1275
time (*uuid.UUID*의 속성), 1543
time() (*asyncio.loop* 메서드), 1160
time() (*datetime.datetime* 메서드), 234
time() (*time* 모듈), 779
Time2Internaldate() (*imaplib* 모듈), 1528
time_hi_version (*uuid.UUID*의 속성), 1543
time_low (*uuid.UUID*의 속성), 1543
time_mid (*uuid.UUID*의 속성), 1543
time_ns() (*time* 모듈), 780
timedelta (*datetime* 클래스), 219
TimedRotatingFileHandler (*logging.handlers* 클래스), 818
timegm() (*calendar* 모듈), 265
timeit
    module, 1947
timeit 명령줄 옵션
    -h, 1950
    --help, 1950
    -n, 1949
    --number, 1949
    -p, 1949
    --process, 1949
    -r, 1949
    --repeat, 1949
    -s, 1949
    --setup, 1949
    -u, 1949
    --unit, 1949
    -v, 1950
    --verbose, 1950
timeit() (*timeit* 모듈), 1947
timeit() (*timeit.Timer* 메서드), 1948
timeout, 1216
Timeout (*asyncio* 클래스), 1123
timeout (*socketserver.BaseServer*의 속성), 1550
timeout (*ssl.SSLSession*의 속성), 1275
timeout (*subprocess.TimeoutExpired*의 속성), 1079
timeout() (*asyncio* 모듈), 1122
timeout() (*curses.window* 메서드), 970
timeout_at() (*asyncio* 모듈), 1123
TIMEOUT_MAX (*_thread* 모듈), 1107
TIMEOUT_MAX (*threading* 모듈), 999
TimeoutError, 120, 1023, 1071, 1153
TimeoutExpired, 1079
Timer (*threading* 클래스), 1010
Timer (*timeit* 클래스), 1948

timerfd_create() (*os* 모듈), 735
timerfd_gettime() (*os* 모듈), 736
timerfd_gettime_ns() (*os* 모듈), 737
timerfd_settime() (*os* 모듈), 735
timerfd_settime_ns() (*os* 모듈), 736
TimerHandle (*asyncio* 클래스), 1175
times() (*os* 모듈), 748
TIMESTAMP (*py_compile.PycInvalidationMode*의 속성), 2218
timestamp() (*datetime.datetime* 메서드), 236
timetuple() (*datetime.date* 메서드), 226
timetuple() (*datetime.datetime* 메서드), 235
timetz() (*datetime.datetime* 메서드), 234
timezone (*datetime* 클래스), 251
timezone (*time* 모듈), 783
--timing
    trace 명령줄 옵션, 1953
title() (*bytearray* 메서드), 80
title() (*bytes* 메서드), 80
title() (*str* 메서드), 63
title() (*turtle* 모듈), 1709
Tk, 1627
Tk (*tkinter* 클래스), 1629
tk (*tkinter.Tk*의 속성), 1629
Tk 옵션 데이터형, 1637
Tkinter, 1627
tkinter
    module, 1627
tkinter.colorchooser
    module, 1640
tkinter.commondialog
    module, 1644
tkinter.dnd
    module, 1647
tkinter.filedialog
    module, 1642
tkinter.font
    module, 1640
tkinter.messagebox
    module, 1644
tkinter.scrolledtext
    module, 1647
tkinter.simpledialog
    module, 1641
tkinter.ttk
    module, 1648
TLS, 1241
--tls-cert
    http.server 명령줄 옵션, 1562
--tls-key
    http.server 명령줄 옵션, 1563
--tls-password-file
    http.server 명령줄 옵션, 1563
TLSv1 (*ssl.TLSVersion*의 속성), 1254
TLSv1_1 (*ssl.TLSVersion*의 속성), 1254
TLSv1_2 (*ssl.TLSVersion*의 속성), 1254
TLSv1_3 (*ssl.TLSVersion*의 속성), 1254
TLSVersion (*ssl* 클래스), 1253
tm_gmtoff (*time.struct_time*의 속성), 779
tm_hour (*time.struct_time*의 속성), 779
tm_isdst (*time.struct_time*의 속성), 779
tm_mday (*time.struct_time*의 속성), 779
tm_min (*time.struct_time*의 속성), 779
tm_mon (*time.struct_time*의 속성), 779
tm_sec (*time.struct_time*의 속성), 779
tm_wday (*time.struct_time*의 속성), 779
tm_yday (*time.struct_time*의 속성), 779
tm_year (*time.struct_time*의 속성), 779
tm_zone (*time.struct_time*의 속성), 779
TMP, 504
TMPDIR, 504
TO_BOOL (*opcode*), 2230
to_bytes() (*int* 메서드), 41
to_eng_string() (*decimal.Context* 메서드), 382
to_eng_string() (*decimal.Decimal* 메서드), 375
to_integral() (*decimal.Decimal* 메서드), 375
to_integral_exact() (*decimal.Context* 메서드), 382
to_integral_exact() (*decimal.Decimal* 메서드), 375
to_integral_value() (*decimal.Decimal* 메서드), 375
to_sci_string() (*decimal.Context* 메서드), 382
to_thread() (*asyncio* 모듈), 1126
ToASCII() (*encodings.idna* 모듈), 216
tobuf() (*tarfile.TarInfo* 메서드), 631
tobytes() (*array.array* 메서드), 303
tobytes() (*memoryview* 메서드), 86
today() (*datetime.datetime*의 클래스 메서드), 229
today() (*datetime.date*의 클래스 메서드), 223
tofile() (*array.array* 메서드), 303
tok_name (*token* 모듈), 2205
token
    module, 2205
Token (*contextvars* 클래스), 1102
token (*shlex.shlex*의 속성), 2266
token (토큰), **2317**
token_bytes() (*secrets* 모듈), 687
token_hex() (*secrets* 모듈), 687
token_urlsafe() (*secrets* 모듈), 687
TokenError, 2211
tokenize
    module, 2210
tokenize 명령줄 옵션
    -e, 2212
    --exact, 2212
    -h, 2212
    --help, 2212
tokenize() (*tokenize* 모듈), 2210
tolist() (*array.array* 메서드), 303
tolist() (*memoryview* 메서드), 86
TOMLDecodeError, 668
tomllib
    module, 667
toordinal() (*datetime.date* 메서드), 226
toordinal() (*datetime.datetime* 메서드), 236

top() (*curses.panel.Panel* 메서드), 988
top() (*poplib.POP3* 메서드), 1526
top_panel() (*curses.panel* 모듈), 987
--top-level-directory
    unittest-discover 명령줄 옵션, 1804
TopologicalSorter (*graphlib* 클래스), 343
toprettyxml() (*xml.dom.minidom.Node* 메서드), 1438
toreadonly() (*memoryview* 메서드), 87
tostring() (*xml.etree.ElementTree* 모듈), 1417
tostringlist() (*xml.etree.ElementTree* 모듈), 1417
total() (*collections.Counter* 메서드), 273
total_changes (*sqlite3.Connection*의 속성), 567
total_nframe (*tracemalloc.Traceback*의 속성), 1964
total_ordering() (*functools* 모듈), 443
total_seconds() (*datetime.timedelta* 메서드), 222
touch() (*pathlib.Path* 메서드), 480
touchline() (*curses.window* 메서드), 970
touchwin() (*curses.window* 메서드), 970
tounicode() (*array.array* 메서드), 303
ToUnicode() (*encodings.idna* 모듈), 216
towards() (*turtle* 모듈), 1690
toxml() (*xml.dom.minidom.Node* 메서드), 1438
tparm() (*curses* 모듈), 963
trace
    module, 1952
--trace
    trace 명령줄 옵션, 1952
Trace (*trace* 클래스), 1953
Trace (*tracemalloc* 클래스), 1963
trace 명령줄 옵션
    -C, 1953
    -c, 1952
    --count, 1952
    --coverdir, 1953
    -f, 1953
    --file, 1953
    -g, 1953
    --help, 1952
    --ignore-dir, 1953
    --ignore-module, 1953
    -l, 1952
    --listfuncs, 1952
    -m, 1953
    --missing, 1953
    --no-report, 1953
    -R, 1953
    -r, 1952
    --report, 1952
    -s, 1953
    --summary, 1953
    -T, 1952
    -t, 1952
    --timing, 1953
    --trace, 1952
    --trackcalls, 1952
    --version, 1952
trace() (*inspect* 모듈), 2096
trace_dispatch() (*bdb.Bdb* 메서드), 1921
traceback
    module, 2065
Traceback (*inspect* 클래스), 2095
Traceback (*tracemalloc* 클래스), 1963
traceback (*tracemalloc.StatisticDiff*의 속성), 1963
traceback (*tracemalloc.Statistic*의 속성), 1963
traceback (*tracemalloc.Trace*의 속성), 1963
traceback_limit (*tracemalloc.Snapshot*의 속성), 1962
traceback_limit (*wsgiref.handlers.BaseHandler*의 속성), 1475
TracebackException (*traceback* 클래스), 2068
tracebacklimit (*sys* 모듈), 2006
TracebackType (*types* 클래스), 315
tracemalloc
    module, 1954
tracer() (*turtle* 모듈), 1704
traces (*tracemalloc.Snapshot*의 속성), 1962
--trackcalls
    trace 명령줄 옵션, 1952
train_dict() (*compression.zstd* 모듈), 587
transfercmd() (*ftplib.FTP* 메서드), 1520
transient_internet() (*test.support.socket_helper* 모듈), 1906
translate() (*bytearray* 메서드), 74
translate() (*bytes* 메서드), 74
translate() (*fnmatch* 모듈), 510
translate() (*glob* 모듈), 507
translate() (*str* 메서드), 63
translation() (*gettext* 모듈), 1611
Transport (*asyncio* 클래스), 1185
transport (*asyncio.StreamWriter*의 속성), 1136
Transport Layer Security, 1241
traps (*decimal.Context*의 속성), 378
Traversable (*importlib.abc* 클래스), 2134
Traversable (*importlib.resources.abc* 클래스), 2149
TraversableResources (*importlib.abc* 클래스), 2134
TraversableResources (*importlib.resources.abc* 클래스), 2150
TreeBuilder (*xml.etree.ElementTree* 클래스), 1423
Treeview (*tkinter.ttk* 클래스), 1658
triangular() (*random* 모듈), 400
tries (*doctest.DocTestRunner*의 속성), 1795
triple-quoted string (삼중 따옴표 된 문자열), **2317**
True, 37, 46
true, 37
True (내장 변수), 35
truediv() (*operator* 모듈), 453
trunc() (*in module math*), 39
trunc() (*math* 모듈), 353
truncate() (*io.IOBase* 메서드), 762
truncate() (*os* 모듈), 730
truth
    value, 37
truth() (*operator* 모듈), 452

try
    문장, 111
Try (*ast* 클래스), 2181
TryStar (*ast* 클래스), 2182
TSTRING_END (*token* 모듈), 2206
TSTRING_MIDDLE (*token* 모듈), 2206
TSTRING_START (*token* 모듈), 2206
ttk, 1648
tty
    I/O 제어, 2271
    module, 2272
ttyname() (*os* 모듈), 710
TUESDAY (*calendar* 모듈), 266
tuple
    object, 49, 51
Tuple (*ast* 클래스), 2168
Tuple (*typing* 모듈), 1767
tuple (내장 클래스), 51
turtle
    module, 1678
Turtle (*turtle* 클래스), 1710
turtledemo
    module, 1714
turtles() (*turtle* 모듈), 1708
TurtleScreen (*turtle* 클래스), 1710
turtlesize() (*turtle* 모듈), 1697
type
    Boolean, 9
    built-in function, 106
    object, 31
    operations on dictionary, 93
    operations on list, 49
    union, 103
--type
    calendar 명령줄 옵션, 268
type (*ctypes.CField*의 속성), 877
type (*optparse.Option*의 속성), 940
type (*socket.socket*의 속성), 1237
type (*tarfile.TarInfo*의 속성), 631
Type (*typing* 클래스), 1767
type (*urllib.request.Request*의 속성), 1484
type (내장 클래스), 31
type (형), **2317**
type alias (형 에일리어스), **2317**
type hint (형 힌트), **2317**
TYPE_ALIAS (*symtable.SymbolTableType* 의 속성), 2201
type_check_only() (*typing* 모듈), 1763
TYPE_CHECKER (*optparse.Option*의 속성), 951
TYPE_CHECKING (*typing* 모듈), 1766
type_comment (*ast.arg*의 속성), 2192
type_comment (*ast.Assign*의 속성), 2176
type_comment (*ast.For*의 속성), 2180
type_comment (*ast.FunctionDef*의 속성), 2191
type_comment (*ast.With*의 속성), 2183
TYPE_COMMENT (*token* 모듈), 2206
TYPE_IGNORE (*token* 모듈), 2206
TYPE_PARAMETERS (*symtable.SymbolTableType*의 속성), 2201
type_repr() (*annotationlib* 모듈), 2105
TYPE_VARIABLE (*symtable.SymbolTableType*의 속성), 2201
typeahead() (*curses* 모듈), 963
TypeAlias (*ast* 클래스), 2178
TypeAlias (*typing* 모듈), 1732
TypeAliasType (*typing* 클래스), 1748
typecode (*array.array*의 속성), 302
typecodes (*array* 모듈), 301
TYPED_ACTIONS (*optparse.Option*의 속성), 952
typed_subpart_iterator() (*email.iterators* 모듈), 1359
TypedDict (*typing* 클래스), 1753
TypeError, 117
TypeGuard (*typing* 모듈), 1740
TypeIgnore (*ast* 클래스), 2189
TypeIs (*typing* 모듈), 1739
types
    built-in, 37
    immutable sequence, 49
    module, 106, 311
    mutable sequence, 49
    operations on integer, 40
    operations on mapping, 93
    operations on numeric, 39
    operations on sequence, 47, 49
TYPES (*optparse.Option*의 속성), 951
types_map (*mimetypes* 모듈), 1391
types_map (*mimetypes.MimeTypes*의 속성), 1392
types_map_inv (*mimetypes.MimeTypes* 의 속성), 1392
TypeVar (*ast* 클래스), 2190
TypeVar (*typing* 클래스), 1742
TypeVarTuple (*ast* 클래스), 2191
TypeVarTuple (*typing* 클래스), 1745
typing
    module, 1717
TZ, 780, 781
tzinfo (*datetime* 클래스), 244
tzinfo (*datetime.datetime*의 속성), 233
tzinfo (*datetime.time*의 속성), 241
tzname (*time* 모듈), 783
tzname() (*datetime.datetime* 메서드), 235
tzname() (*datetime.time* 메서드), 243
tzname() (*datetime.timezone* 메서드), 251
tzname() (*datetime.tzinfo* 메서드), 245
TZPATH (*zoneinfo* 모듈), 260
tzset() (*time* 모듈), 780

U

-u
    timeit 명령줄 옵션, 1949
    uuid 명령줄 옵션, 1545
U (*re* 모듈), 150
UAdd (*ast* 클래스), 2171
ucd_3_2_0 (*unicodedata* 모듈), 183

udata (*select.kevent*의 속성), 1284
UDPServer (*socketserver* 클래스), 1547
UF_APPEND (*stat* 모듈), 497
UF_COMPRESSED (*stat* 모듈), 497
UF_DATAVAULT (*stat* 모듈), 497
UF_HIDDEN (*stat* 모듈), 497
UF_IMMUTABLE (*stat* 모듈), 497
UF_NODUMP (*stat* 모듈), 497
UF_NOUNLINK (*stat* 모듈), 497
UF_OPAQUE (*stat* 모듈), 497
UF_SETTABLE (*stat* 모듈), 497
UF_TRACKED (*stat* 모듈), 497
UID (*plistlib* 클래스), 671
uid (*tarfile.TarInfo*의 속성), 632
uid() (*imaplib.IMAP4* 메서드), 1534
uidl() (*poplib.POP3* 메서드), 1526
ulp() (*math* 모듈), 355
umask() (*os* 모듈), 698
unalias (*pdb command*), 1937
uname (*tarfile.TarInfo*의 속성), 632
uname() (*os* 모듈), 698
uname() (*platform* 모듈), 830
UNARY_INVERT (*opcode*), 2229
UNARY_NEGATIVE (*opcode*), 2229
UNARY_NOT (*opcode*), 2229
UnaryOp (*ast* 클래스), 2170
UnboundLocalError, 118
unbuffered I/O, 25
UNC paths
    and os.makedirs(), 718
uncancel() (*asyncio.Task* 메서드), 1132
UNCHECKED_HASH (*py_compile.PycInvalidationMode*의 속성), 2218
unconsumed_tail (*zlib.Decompress*의 속성), 597
unctrl() (*curses* 모듈), 964
unctrl() (*curses.ascii* 모듈), 987
Underflow (*decimal* 클래스), 385
undisplay (*pdb command*), 1936
undo() (*turtle* 모듈), 1689
undobufferentries() (*turtle* 모듈), 1701
undoc_header (*cmd.Cmd*의 속성), 990
unescape() (*html* 모듈), 1401
unescape() (*xml.sax.saxutils* 모듈), 1450
UnexpectedException, 1799
unexpectedSuccesses (*unittest.TestResult*의 속성), 1825
unfreeze() (*gc* 모듈), 2080
unget_wch() (*curses* 모듈), 964
ungetch() (*curses* 모듈), 964
ungetch() (*msvcrt* 모듈), 2248
ungetmouse() (*curses* 모듈), 964
ungetwch() (*msvcrt* 모듈), 2248
unhexlify() (*binascii* 모듈), 1399
UNICODE (*re* 모듈), 150
unicode_literals (*__future__* 모듈), 2076
unicodedata
    module, 180
UnicodeDecodeError, 118
UnicodeEncodeError, 118
UnicodeError, 118
UnicodeTranslateError, 118
UnicodeWarning, 120
unidata_version (*unicodedata* 모듈), 183
unified_diff() (*difflib* 모듈), 167
uniform() (*random* 모듈), 400
UnimplementedFileMode, 1511
Union
    object, 103
union
    type, 103
Union (*ctypes* 클래스), 874
Union (*typing* 클래스), 1733
union() (*frozenset* 메서드), 92
UnionType (*types* 클래스), 315
UNIQUE (*enum.EnumCheck*의 속성), 338
unique() (*enum* 모듈), 342
--unit
    timeit 명령줄 옵션, 1949
unittest
    module, 1800
unittest 명령줄 옵션
    -b, 1803
    --buffer, 1803
    -c, 1803
    --catch, 1803
    --durations, 1803
    -f, 1803
    --failfast, 1803
    -k, 1803
    --locals, 1803
unittest-discover 명령줄 옵션
    -p, 1804
    --pattern, 1804
    -s, 1804
    --start-directory, 1804
    -t, 1804
    --top-level-directory, 1804
    -v, 1803
    --verbose, 1803
unittest.mock
    module, 1831
universal newlines
    bytearray.splitlines method, 79
    bytes.splitlines method, 79
    open() built-in function, 24
    str.splitlines method, 62
universal newlines (유니버설 줄 넘김), **2317**
UNIX
    I/O 제어, 2275
    파일 제어, 2275
unix_dialect (*csv* 클래스), 644
unix_shell (*test.support* 모듈), 1896
UnixDatagramServer (*socketserver* 클래스), 1547
UnixStreamServer (*socketserver* 클래스), 1547
unknown (*uuid.SafeUUID*의 속성), 1541

unknown_decl() (*html.parser.HTMLParser* 메서드), 1404
unknown_open() (*urllib.request.BaseHandler* 메서드), 1487
unknown_open() (*urllib.request.UnknownHandler* 메서드), 1491
UnknownHandler (*urllib.request* 클래스), 1484
UnknownProtocol, 1511
UnknownTransferEncoding, 1511
unlink() (*multiprocessing.shared_memory.SharedMemory* 메서드), 1058
unlink() (*os* 모듈), 730
unlink() (*pathlib.Path* 메서드), 482
unlink() (*test.support.os_helper* 모듈), 1910
unlink() (*xml.dom.minidom.Node* 메서드), 1438
unload() (*test.support.import_helper* 모듈), 1911
unlock() (*mailbox.Babyl* 메서드), 1379
unlock() (*mailbox.Mailbox* 메서드), 1374
unlock() (*mailbox.Maildir* 메서드), 1376
unlock() (*mailbox.mbox* 메서드), 1377
unlock() (*mailbox.MH* 메서드), 1378
unlock() (*mailbox.MMDF* 메서드), 1380
unlockpt() (*os* 모듈), 710
UNNAMED_SECTION (*configparser* 모듈), 665
UnnamedSectionDisabledError, 667
Unpack (*typing* 모듈), 1741
unpack() (*struct* 모듈), 192
unpack() (*struct.Struct* 메서드), 198
unpack_archive() (*shutil* 모듈), 519
UNPACK_EX (*opcode*), 2233
unpack_from() (*struct* 모듈), 192
unpack_from() (*struct.Struct* 메서드), 198
UNPACK_SEQUENCE (*opcode*), 2233
unparse() (*ast* 모듈), 2196
unparsedEntityDecl() (*xml.sax.handler.DTDHandler* 메서드), 1448
UnparsedEntityDeclHandler() (*xml.parsers.expat.xmlparser* 메서드), 1459
Unpickler (*pickle* 클래스), 527
UnpicklingError, 526
unquote() (*email.utils* 모듈), 1357
unquote() (*urllib.parse* 모듈), 1503
unquote_plus() (*urllib.parse* 모듈), 1503
unquote_to_bytes() (*urllib.parse* 모듈), 1503
unraisablehook() (*sys* 모듈), 2007
unregister() (*atexit* 모듈), 2064
unregister() (*codecs* 모듈), 200
unregister() (*faulthandler* 모듈), 1927
unregister() (*select.devpoll* 메서드), 1280
unregister() (*select.epoll* 메서드), 1281
unregister() (*selectors.BaseSelector* 메서드), 1286
unregister() (*select.poll* 메서드), 1281
unregister_archive_format() (*shutil* 모듈), 519
unregister_dialect() (*csv* 모듈), 642
unregister_unpack_format() (*shutil* 모듈), 520
unsafe (*uuid.SafeUUID*의 속성), 1541
unselect() (*imaplib.IMAP4* 메서드), 1534
unset() (*test.support.os_helper.EnvironmentVarGuard* 메서드), 1909
unsetenv() (*os* 모듈), 698
unshare() (*os* 모듈), 698
UnstructuredHeader (*email.headerregistry* 클래스), 1327
unsubscribe() (*imaplib.IMAP4* 메서드), 1534
UnsupportedOperation, 461, 759
until (*pdb command*), 1934
untokenize() (*tokenize* 모듈), 2211
untouchwin() (*curses.window* 메서드), 970
unused_data (*bz2.BZ2Decompressor*의 속성), 604
unused_data (*compression.zstd.ZstdDecompressor* 의 속성), 587
unused_data (*lzma.LZMADecompressor*의 속성), 609
unused_data (*zlib.Decompress*의 속성), 597
unverifiable (*urllib.request.Request*의 속성), 1484
unwrap() (*inspect* 모듈), 2093
unwrap() (*ssl.SSLSocket* 메서드), 1257
unwrap() (*urllib.parse* 모듈), 1500
up (*pdb command*), 1933
up() (*turtle* 모듈), 1692
update() (*collections.Counter* 메서드), 273
update() (*dict* 메서드), 96
update() (*frozenset* 메서드), 93
update() (*hashlib.hash* 메서드), 675
update() (*hmac.HMAC* 메서드), 685
update() (*http.cookies.Morsel* 메서드), 1566
update() (*mailbox.Mailbox* 메서드), 1373
update() (*mailbox.Maildir* 메서드), 1376
update() (*trace.CoverageResults* 메서드), 1954
update() (*turtle* 모듈), 1704
update_abstractmethods() (*abc* 모듈), 2063
update_authenticated() (*urllib.request.HTTPPasswordMgrWithPriorAuth* 메서드), 1489
update_lines_cols() (*curses* 모듈), 964
update_panels() (*curses.panel* 모듈), 987
update_visible() (*mailbox.BabylMessage* 메서드), 1386
update_wrapper() (*functools* 모듈), 450
--upgrade
venv 명령줄 옵션, 1968
upgrade_dependencies() (*venv.EnvBuilder* 메서드), 1972
--upgrade-deps
venv 명령줄 옵션, 1968
upper() (*bytearray* 메서드), 81
upper() (*bytes* 메서드), 81
upper() (*str* 메서드), 63
urandom() (*os* 모듈), 756
URL, 1496, 1505, 1556
구문 분석, 1496
상대적, 1496
url (*http.client.HTTPResponse*의 속성), 1515
url (*urllib.error.HTTPError*의 속성), 1505
url (*urllib.response.addinfourl*의 속성), 1495

url (*xmlrpc.client.ProtocolError*의 속성), 1581
url2pathname() (*urllib.request* 모듈), 1480
urlcleanup() (*urllib.request* 모듈), 1494
urldefrag() (*urllib.parse* 모듈), 1500
urlencode() (*urllib.parse* 모듈), 1503
URLError, 1504
urljoin() (*urllib.parse* 모듈), 1500
urllib
    module, 1478
urllib.error
    module, 1504
urllib.parse
    module, 1496
urllib.request
    module, 1478
    모듈, 1510
urllib.response
    module, 1495
urllib.robotparser
    module, 1505
urlopen() (*urllib.request* 모듈), 1479
urlparse() (*urllib.parse* 모듈), 1496
urlretrieve() (*urllib.request* 모듈), 1494
urlsafe_b64decode() (*base64* 모듈), 1395
urlsafe_b64encode() (*base64* 모듈), 1395
urlsplit() (*urllib.parse* 모듈), 1499
urlunparse() (*urllib.parse* 모듈), 1499
urlunsplit() (*urllib.parse* 모듈), 1500
urn (*uuid.UUID*의 속성), 1543
US (*curses.ascii* 모듈), 985
use_default_colors() (*curses* 모듈), 964
use_env() (*curses* 모듈), 964
use_rawinput (*cmd.Cmd*의 속성), 990
use_tool_id() (*sys.monitoring* 모듈), 2009
UseForeignDTD() (*xml.parsers.expat.xmlparser* 메서드), 1456
USER, 954
user
    effective id, 693
    id, 695
    id, setting, 697
--user
    ensurepip 명령줄 옵션, 1966
user() (*poplib.POP3* 메서드), 1525
USER_BASE (*site* 모듈), 2110
user_call() (*bdb.Bdb* 메서드), 1922
user_exception() (*bdb.Bdb* 메서드), 1922
user_line() (*bdb.Bdb* 메서드), 1922
user_return() (*bdb.Bdb* 메서드), 1922
USER_SITE (*site* 모듈), 2110
--user-base
    site 명령줄 옵션, 2111
usercustomize
    module, 2110
UserDict (*collections* 클래스), 285
UserList (*collections* 클래스), 286
USERNAME, 488, 694, 954
username (*email.headerregistry.Address*의 속성), 1331
USERPROFILE, 488
userptr() (*curses.panel.Panel* 메서드), 988
--user-site
    site 명령줄 옵션, 2111
UserString (*collections* 클래스), 286
UserWarning, 120
USTAR_FORMAT (*tarfile* 모듈), 626
USub (*ast* 클래스), 2171
UTC, 772
UTC (*datetime* 모듈), 218
utc (*datetime.timezone*의 속성), 251
utcfromtimestamp() (*datetime.datetime*의 클래스 메서드), 230
utcnow() (*datetime.datetime*의 클래스 메서드), 230
utcoffset() (*datetime.datetime* 메서드), 235
utcoffset() (*datetime.time* 메서드), 243
utcoffset() (*datetime.timezone* 메서드), 251
utcoffset() (*datetime.tzinfo* 메서드), 244
utctimetuple() (*datetime.datetime* 메서드), 235
utf8 (*email.policy.EmailPolicy*의 속성), 1322
utf8() (*poplib.POP3* 메서드), 1526
utf8_enabled (*imaplib.IMAP4*의 속성), 1534
utf8_mode (*sys.flags*의 속성), 1988
utime() (*os* 모듈), 730
uu
    module, 2297
uuid
    module, 1541
--uuid
    uuid 명령줄 옵션, 1545
UUID (*uuid* 클래스), 1541
uuid 명령줄 옵션
    -C, 1546
    --count, 1546
    -h, 1545
    --help, 1545
    -N, 1545
    -n, 1545
    --name, 1545
    --namespace, 1545
    -u, 1545
    --uuid, 1545
uuid1() (*uuid* 모듈), 1544
uuid3() (*uuid* 모듈), 1544
uuid4() (*uuid* 모듈), 1544
uuid5() (*uuid* 모듈), 1544
uuid6() (*uuid* 모듈), 1544
uuid7() (*uuid* 모듈), 1544
uuid8() (*uuid* 모듈), 1544

V

-v
    doctest 명령줄 옵션, 1780
    python--m-sqlite3-[-h]-[-v]-[filename]-[sql] 명령줄 옵션, 574
    tarfile 명령줄 옵션, 637
    timeit 명령줄 옵션, 1950
    unittest-discover 명령줄 옵션, 1803

v4_int_to_packed() (*ipaddress* 모듈), 1602
v6_int_to_packed() (*ipaddress* 모듈), 1602
valid_signals() (*signal* 모듈), 1292
validator() (*wsgiref.validate* 모듈), 1472
value
    truth, 37
VALUE (*annotationlib.Format*의 속성), 2102
value (*ctypes._SimpleCData*의 속성), 872
value (*enum.Enum*의 속성), 331
value (*http.cookiejar.Cookie*의 속성), 1574
value (*http.cookies.Morsel*의 속성), 1565
value (*StopIteration*의 속성), 116
value (*string.templatelib.Interpolation*의 속성), 140
value (*xml.dom.Attr*의 속성), 1433
Value() (*multiprocessing* 모듈), 1032
Value() (*multiprocessing.managers.SyncManager* 메서드), 1037
Value() (*multiprocessing.sharedctypes* 모듈), 1033
value_decode() (*http.cookies.BaseCookie* 메서드), 1564
value_encode() (*http.cookies.BaseCookie* 메서드), 1564
VALUE_WITH_FAKE_GLOBALS (*annotationlib.Format*의 속성), 2102
ValueError, 118
valuerefs() (*weakref.WeakValueDictionary* 메서드), 306
values
    Boolean, 46
Values (*optparse* 클래스), 939
values (*string.templatelib.Template*의 속성), 138
values() (*contextvars.Context* 메서드), 1104
values() (*dict* 메서드), 96
values() (*email.message.EmailMessage* 메서드), 1306
values() (*email.message.Message* 메서드), 1344
values() (*mailbox.Mailbox* 메서드), 1372
values() (*types.MappingProxyType* 메서드), 316
ValuesView (*collections.abc* 클래스), 291
ValuesView (*typing* 클래스), 1770
var (*contextvars.Token*의 속성), 1102
variable annotation (변수 어노테이션), **2318**
variance (*statistics.NormalDist*의 속성), 418
variance() (*statistics* 모듈), 414
variant (*uuid.UUID*의 속성), 1543
vars()
    built-in function, 32
vbar (*tkinter.scrolledtext.ScrolledText*의 속성), 1647
VBAR (*token* 모듈), 2207
VBAREQUAL (*token* 모듈), 2209
VC_ASSEMBLY_PUBLICKEYTOKEN (*msvcrt* 모듈), 2249
Vec2D (*turtle* 클래스), 1710
venv
    module, 1967
venv 명령줄 옵션
    --clear, 1968
    --copies, 1968
    ENV_DIR, 1968
    --prompt, 1968
    --symlinks, 1968
    --system-site-packages, 1968
    --upgrade, 1968
    --upgrade-deps, 1968
    --without-pip, 1968
    --without-scm-ignore-files, 1968
--verbose
    doctest 명령줄 옵션, 1780
    tarfile 명령줄 옵션, 637
    timeit 명령줄 옵션, 1950
    unittest-discover 명령줄 옵션, 1803
VERBOSE (*re* 모듈), 150
verbose (*sys.flags*의 속성), 1988
verbose (*tabnanny* 모듈), 2214
verbose (*test.support* 모듈), 1896
verify() (*enum* 모듈), 342
verify() (*smtplib.SMTP* 메서드), 1537
VERIFY_ALLOW_PROXY_CERTS (*ssl* 모듈), 1248
verify_client_post_handshake() (*ssl.SSLSocket* 메서드), 1257
verify_code (*ssl.SSLCertVerificationError*의 속성), 1245
VERIFY_CRL_CHECK_CHAIN (*ssl* 모듈), 1248
VERIFY_CRL_CHECK_LEAF (*ssl* 모듈), 1248
VERIFY_DEFAULT (*ssl* 모듈), 1248
verify_flags (*ssl.SSLContext*의 속성), 1266
verify_generated_headers (*email.policy.Policy*의 속성), 1320
verify_message (*ssl.SSLCertVerificationError*의 속성), 1245
verify_mode (*ssl.SSLContext*의 속성), 1266
verify_request() (*socketserver.BaseServer* 메서드), 1551
VERIFY_X509_PARTIAL_CHAIN (*ssl* 모듈), 1248
VERIFY_X509_STRICT (*ssl* 모듈), 1248
VERIFY_X509_TRUSTED_FIRST (*ssl* 모듈), 1248
VerifyFlags (*ssl* 클래스), 1248
VerifyMode (*ssl* 클래스), 1248
--version
    python--m-sqlite3-[-h]-[-v]-[filename]-[sql] 명령줄 옵션, 574
    trace 명령줄 옵션, 1952
version (*curses* 모듈), 971
version (*email.headerregistry.MIMEVersionHeader*의 속성), 1329
version (*http.client.HTTPResponse*의 속성), 1515
version (*http.cookiejar.Cookie*의 속성), 1574
version (*http.cookies.Morsel*의 속성), 1565
version (*ipaddress.IPv4Address*의 속성), 1590
version (*ipaddress.IPv4Network*의 속성), 1596
version (*ipaddress.IPv6Address*의 속성), 1593
version (*ipaddress.IPv6Network*의 속성), 1599
version (*marshal* 모듈), 545
version (*sys* 모듈), 2007
version (*sys.thread_info*의 속성), 2006
version (*uuid.UUID*의 속성), 1543
version() (*ensurepip* 모듈), 1966
version() (*importlib.metadata* 모듈), 2154

version() (*platform* 모듈), 830
version() (*ssl.SSLSocket* 메서드), 1258
version_info (*sys* 모듈), 2007
version_string() (*http.server.BaseHTTPRequestHandler* 메서드), 1560
vformat() (*string.Formatter* 메서드), 126
virtual environment (가상 환경), **2318**
virtual machine (가상 기계), **2318**
visit() (*ast.NodeVisitor* 메서드), 2197
visit_Constant() (*ast.NodeVisitor* 메서드), 2197
vline() (*curses.window* 메서드), 970
voidcmd() (*ftplib.FTP* 메서드), 1520
volume (*zipfile.ZipInfo*의 속성), 621
vonmisesvariate() (*random* 모듈), 401
VT (*curses.ascii* 모듈), 984

W

-w
  calendar 명령줄 옵션, 268
W_OK (*os* 모듈), 713
wait() (*asyncio* 모듈), 1124
wait() (*asyncio.Barrier* 메서드), 1145
wait() (*asyncio.Condition* 메서드), 1143
wait() (*asyncio.Event* 메서드), 1142
wait() (*asyncio.subprocess.Process* 메서드), 1148
wait() (*concurrent.futures* 모듈), 1071
wait() (*multiprocessing.connection* 모듈), 1045
wait() (*multiprocessing.pool.AsyncResult* 메서드), 1043
wait() (*os* 모듈), 748
wait() (*subprocess.Popen* 메서드), 1086
wait() (*threading.Barrier* 메서드), 1011
wait() (*threading.Condition* 메서드), 1007
wait() (*threading.Event* 메서드), 1010
wait3() (*os* 모듈), 749
wait4() (*os* 모듈), 749
wait_closed() (*asyncio.Server* 메서드), 1177
wait_closed() (*asyncio.StreamWriter* 메서드), 1137
wait_for() (*asyncio* 모듈), 1124
wait_for() (*asyncio.Condition* 메서드), 1143
wait_for() (*threading.Condition* 메서드), 1007
wait_process() (*test.support* 모듈), 1901
wait_threads_exit() (*test.support.threading_helper* 모듈), 1908
wait_until_any_call_with() (*unittest.mock.ThreadingMock* 메서드), 1848
wait_until_called() (*unittest.mock.ThreadingMock* 메서드), 1847
waitid() (*os* 모듈), 748
waitpid() (*os* 모듈), 749
waitstatus_to_exitcode() (*os* 모듈), 751
walk() (*ast* 모듈), 2197
walk() (*email.message.EmailMessage* 메서드), 1309
walk() (*email.message.Message* 메서드), 1348
walk() (*os* 모듈), 730
walk() (*pathlib.Path* 메서드), 478
walk_packages() (*pkgutil* 모듈), 2120
walk_stack() (*traceback* 모듈), 2067
walk_tb() (*traceback* 모듈), 2067
walrus operator, **2318**
want (*doctest.Example*의 속성), 1792
warn() (*warnings* 모듈), 2030
warn_default_encoding (*sys.flags*의 속성), 1988
warn_explicit() (*warnings* 모듈), 2030
Warning, 120, 572
WARNING (*logging* 모듈), 790
WARNING (*tkinter.messagebox* 모듈), 1646
warning() (*logging* 모듈), 798
warning() (*logging.Logger* 메서드), 788
warning() (*xml.sax.handler.ErrorHandler* 메서드), 1449
warnings
  module, 2025
WarningsRecorder (*test.support.warnings_helper* 클래스), 1913
warnoptions (*sys* 모듈), 2008
wasSuccessful() (*unittest.TestResult* 메서드), 1825
WatchedFileHandler (*logging.handlers* 클래스), 816
wave
  module, 1605
Wave_read (*wave* 클래스), 1606
Wave_write (*wave* 클래스), 1607
WCONTINUED (*os* 모듈), 750
WCOREDUMP() (*os* 모듈), 751
WeakKeyDictionary (*weakref* 클래스), 305
WeakMethod (*weakref* 클래스), 306
weakref
  module, 304
WeakSet (*weakref* 클래스), 306
WeakValueDictionary (*weakref* 클래스), 306
webbrowser
  module, 1465
webbrowser 명령줄 옵션
  -n, 1465
  --new-tab, 1466
  --new-window, 1465
  -t, 1466
WEDNESDAY (*calendar* 모듈), 266
weekday (*calendar.IllegalWeekdayError*의 속성), 267
weekday() (*calendar* 모듈), 265
weekday() (*datetime.date* 메서드), 226
weekday() (*datetime.datetime* 메서드), 236
weekheader() (*calendar* 모듈), 265
weibullvariate() (*random* 모듈), 401
WEXITED (*os* 모듈), 750
WEXITSTATUS() (*os* 모듈), 752
wfile (*http.server.BaseHTTPRequestHandler*의 속성), 1558
wfile (*socketserver.DatagramRequestHandler*의 속성), 1552
whatis (*pdb command*), 1935
when() (*asyncio.Timeout* 메서드), 1123
when() (*asyncio.TimerHandle* 메서드), 1176

whence (*concurrent.interpreters.Interpreter* 의 속성), 1075
where (*pdb command*), 1933
which() (*shutil* 모듈), 516
whichdb() (*dbm* 모듈), 545
while
    statement, 37
While (*ast* 클래스), 2180
whitespace (*shlex.shlex*의 속성), 2266
whitespace (*string* 모듈), 126
whitespace_split (*shlex.shlex*의 속성), 2266
Widget (*tkinter.ttk* 클래스), 1651
--width
    calendar 명령줄 옵션, 268
width (*sys.hash_info*의 속성), 1994
width (*textwrap.TextWrapper*의 속성), 179
width() (*turtle* 모듈), 1692
WIFCONTINUED() (*os* 모듈), 751
WIFEXITED() (*os* 모듈), 752
WIFSIGNALED() (*os* 모듈), 751
WIFSTOPPED() (*os* 모듈), 751
win32_code_page_search_function() (*encodings* 모듈), 215
win32_edition() (*platform* 모듈), 831
win32_is_iot() (*platform* 모듈), 831
win32_ver() (*platform* 모듈), 831
WinDLL (*ctypes* 클래스), 862
window (*curses* 클래스), 964
window() (*curses.panel.Panel* 메서드), 988
window_height() (*turtle* 모듈), 1708
window_log (*compression.zstd.CompressionParameter* 의 속성), 589
window_log_max (*compression.zstd.DecompressionParameter* 의 속성), 592
window_width() (*turtle* 모듈), 1708
WindowsError, 118
WindowsPath (*pathlib* 클래스), 471
WindowsProactorEventLoopPolicy (*asyncio* 클래스), 1199
WindowsRegistryFinder (*importlib.machinery* 클래스), 2135
WindowsSelectorEventLoopPolicy (*asyncio* 클래스), 1199
winerror (*OSError*의 속성), 115
WinError() (*ctypes* 모듈), 870
WINFUNCTYPE() (*ctypes* 모듈), 865
winreg
    module, 2250
WinSock, 1279
winsound
    module, 2258
winver (*sys* 모듈), 2008
With (*ast* 클래스), 2183
WITH_EXCEPT_START (*opcode*), 2233
with_hostmask (*ipaddress.IPv4Interface* 의 속성), 1601
with_hostmask (*ipaddress.IPv4Network* 의 속성), 1597
with_hostmask (*ipaddress.IPv6Interface* 의 속성), 1602
with_hostmask (*ipaddress.IPv6Network* 의 속성), 1599
with_name() (*pathlib.PurePath* 메서드), 469
with_netmask (*ipaddress.IPv4Interface*의 속성), 1601
with_netmask (*ipaddress.IPv4Network*의 속성), 1597
with_netmask (*ipaddress.IPv6Interface*의 속성), 1602
with_netmask (*ipaddress.IPv6Network*의 속성), 1599
with_prefixlen (*ipaddress.IPv4Interface* 의 속성), 1601
with_prefixlen (*ipaddress.IPv4Network* 의 속성), 1596
with_prefixlen (*ipaddress.IPv6Interface* 의 속성), 1602
with_prefixlen (*ipaddress.IPv6Network* 의 속성), 1599
with_pymalloc() (*test.support* 모듈), 1899
with_segments() (*pathlib.PurePath* 메서드), 470
with_statement (*__future__* 모듈), 2076
with_stem() (*pathlib.PurePath* 메서드), 470
with_suffix() (*pathlib.PurePath* 메서드), 470
with_traceback() (*BaseException* 메서드), 112
withitem (*ast* 클래스), 2183
--without-pip
    venv 명령줄 옵션, 1968
--without-scm-ignore-files
    venv 명령줄 옵션, 1968
WNOHANG (*os* 모듈), 750
WNOWAIT (*os* 모듈), 750
wordchars (*shlex.shlex*의 속성), 2266
wrap() (*textwrap* 모듈), 177
wrap() (*textwrap.TextWrapper* 메서드), 180
wrap_bio() (*ssl.SSLContext* 메서드), 1264
wrap_future() (*asyncio* 모듈), 1181
wrap_socket() (*ssl.SSLContext* 메서드), 1263
wrapper() (*curses* 모듈), 964
WrapperDescriptorType (*types* 모듈), 314
wraps() (*functools* 모듈), 450
WRITABLE (*_tkinter* 모듈), 1640
WRITABLE (*inspect.BufferFlags*의 속성), 2099
writable() (*bz2.BZ2File* 메서드), 603
writable() (*io.IOBase* 메서드), 762
WRITE (*inspect.BufferFlags*의 속성), 2100
write() (*asyncio.StreamWriter* 메서드), 1136
write() (*asyncio.WriteTransport* 메서드), 1187
write() (*codecs.StreamWriter* 메서드), 206
write() (*code.InteractiveInterpreter* 메서드), 2114
write() (*configparser.ConfigParser* 메서드), 664
write() (*email.generator.BytesGenerator* 메서드), 1316
write() (*email.generator.Generator* 메서드), 1317
write() (*io.BufferedIOBase* 메서드), 764
write() (*io.BufferedWriter* 메서드), 766
write() (*io.RawIOBase* 메서드), 762
write() (*io.TextIOBase* 메서드), 767

write() (*io.Writer* 메서드), 770
write() (*mmap.mmap* 메서드), 1301
write() (*os* 모듈), 710
write() (*sqlite3.Blob* 메서드), 571
write() (*ssl.MemoryBIO* 메서드), 1274
write() (*ssl.SSLSocket* 메서드), 1255
write() (*turtle* 모듈), 1696
write() (*xml.etree.ElementTree.ElementTree* 메서드), 1422
write() (*zipfile.ZipFile* 메서드), 616
write_byte() (*mmap.mmap* 메서드), 1301
write_bytes() (*pathlib.Path* 메서드), 477
write_docstringdict() (*turtle* 모듈), 1712
write_eof() (*asyncio.StreamWriter* 메서드), 1136
write_eof() (*asyncio.WriteTransport* 메서드), 1188
write_eof() (*ssl.MemoryBIO* 메서드), 1274
write_history_file() (*readline* 모듈), 185
write_results() (*trace.CoverageResults* 메서드), 1954
write_text() (*pathlib.Path* 메서드), 477
write_through (*io.TextIOWrapper* 의 속성), 768
writeframes() (*wave.Wave_write* 메서드), 1607
writeframesraw() (*wave.Wave_write* 메서드), 1607
writeheader() (*csv.DictWriter* 메서드), 647
writelines() (*asyncio.StreamWriter* 메서드), 1136
writelines() (*asyncio.WriteTransport* 메서드), 1187
writelines() (*codecs.StreamWriter* 메서드), 206
writelines() (*io.IOBase* 메서드), 762
writepy() (*zipfile.PyZipFile* 메서드), 619
Writer (*io* 클래스), 770
writer() (*csv* 모듈), 642
writerow() (*csv.csvwriter* 메서드), 647
writerows() (*csv.csvwriter* 메서드), 647
writestr() (*zipfile.ZipFile* 메서드), 617
WriteTransport (*asyncio* 클래스), 1185
writev() (*os* 모듈), 711
writexml() (*xml.dom.minidom.Node* 메서드), 1438
WrongDocumentErr, 1435
wsgi_file_wrapper (*wsgiref.handlers.BaseHandler* 의 속성), 1476
wsgi_multiprocess (*wsgiref.handlers.BaseHandler* 의 속성), 1474
wsgi_multithread (*wsgiref.handlers.BaseHandler* 의 속성), 1474
wsgi_run_once (*wsgiref.handlers.BaseHandler* 의 속성), 1474
WSGIApplication (*wsgiref.types* 모듈), 1476
WSGIEnvironment (*wsgiref.types* 모듈), 1476
wsgiref
    module, 1468
wsgiref.handlers
    module, 1473
wsgiref.headers
    module, 1470
wsgiref.simple_server
    module, 1471
wsgiref.types
    module, 1476
wsgiref.util
    module, 1468
wsgiref.validate
    module, 1472
WSGIRequestHandler (*wsgiref.simple_server* 클래스), 1472
WSGIServer (*wsgiref.simple_server* 클래스), 1471
wShowWindow (*subprocess.STARTUPINFO* 의 속성), 1088
WSTOPPED (*os* 모듈), 750
WSTOPSIG() (*os* 모듈), 752
wstring_at() (*ctypes* 모듈), 870
WTERMSIG() (*os* 모듈), 752
WUNTRACED (*os* 모듈), 750
WWW, 1465, 1496, 1505
    서버, 1556

X

-x
    compileall 명령줄 옵션, 2219
X (*re* 모듈), 150
X509 certificate, 1267
X_OK (*os* 모듈), 713
xatom() (*imaplib.IMAP4* 메서드), 1534
XATTR_CREATE (*os* 모듈), 738
XATTR_REPLACE (*os* 모듈), 738
XATTR_SIZE_MAX (*os* 모듈), 738
내장 함수
    eval, 320, 321
    컴파일, 313
xcor() (*turtle* 모듈), 1690
대용량 파일, 2269
데이터
    패킹 바이너리, 191
    표 형식, 641
데이터베이스, 550
    유니코드, 180
xdrlib
    module, 2297
도움말
    온라인, 1773
디렉터리
    site-packages, 2109
    삭제하기, 514
디버거, 998, 1669, 1993, 2002
    구성 파일, 1933
디버깅, 1928
XHTML, 1402
XHTML_NAMESPACE (*xml.dom* 모듈), 1428
마샬링
    객체, 523
메시지 다이제스트, MD5, 673
모듈
    __main__, 2123, 2124
    _locale, 1617
    base64, 1398
    bdb, 1928
    cmd, 1928

dbm.gnu, 541
dbm.ndbm, 541
errno, 115
glob, 509
pwd, 488
pyexpat, 1454
re, 509
urllib.request, 1510
검색 경로, 510, 1999, 2108
복사, 540
소켓, 1465
수학, 364
쉘프, 543
시그널, 1107
운영 체제, 2268
피클, 318, 540, 543
모듈 브라우저, 1666
문서화
생성, 1773
온라인, 1773
문자, 180
문장
assert, 113
except, 111
raise, 111
try, 111
임포트, 2109
바이너리
데이터, 패킹, 191
바이너리 모드, 25
바이트 코드
파일, 2217
배열, 301
벤치마킹, 775, 780, 1947
보안
http.server, 1563
복사
모듈, 540
프로토콜, 531
xml
module, 1407
XML() (*xml.etree.ElementTree* 모듈), 1417
XML_ERROR_ABORTED (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_AMPLIFICATION_LIMIT_BREACH (*xml.parsers.expat.errors* 모듈), 1464
XML_ERROR_ASYNC_ENTITY (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_BAD_CHAR_REF (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_BINARY_ENTITY_REF (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_DUPLICATE_ATTRIBUTE (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_ENTITY_DECLARED_IN_PE (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_EXTERNAL_ENTITY_HANDLING (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_FEATURE_REQUIRES_XML_DTD (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_FINISHED (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_INCOMPLETE_PE (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_INCORRECT_ENCODING (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_INVALID_ARGUMENT (*xml.parsers.expat.errors* 모듈), 1464
XML_ERROR_INVALID_TOKEN (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_JUNK_AFTER_DOC_ELEMENT (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_MISPLACED_XML_PI (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_NO_BUFFER (*xml.parsers.expat.errors* 모듈), 1464
XML_ERROR_NO_ELEMENTS (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_NO_MEMORY (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_NOT_STANDALONE (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_NOT_STARTED (*xml.parsers.expat.errors* 모듈), 1464
XML_ERROR_NOT_SUSPENDED (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_PARAM_ENTITY_REF (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_PARTIAL_CHAR (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_PUBLICID (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_RECURSIVE_ENTITY_REF (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_RESERVED_NAMESPACE_URI (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_RESERVED_PREFIX_XML (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_RESERVED_PREFIX_XMLNS (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_SUSPEND_PE (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_SUSPENDED (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_SYNTAX (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_TAG_MISMATCH (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_TEXT_DECL (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_UNBOUND_PREFIX (*xml.parsers.expat.errors* 모듈), 1463

XML_ERROR_UNCLOSED_CDATA_SECTION (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_UNCLOSED_TOKEN (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_UNDECLARING_PREFIX (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_UNDEFINED_ENTITY (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_UNEXPECTED_STATE (*xml.parsers.expat.errors* 모듈), 1463
XML_ERROR_UNKNOWN_ENCODING (*xml.parsers.expat.errors* 모듈), 1462
XML_ERROR_XML_DECL (*xml.parsers.expat.errors* 모듈), 1463
XML_NAMESPACE (*xml.dom* 모듈), 1428
xmlcharrefreplace
    에러 처리기의 이름, 202
xmlcharrefreplace_errors() (*codecs* 모듈), 203
XmlDeclHandler() (*xml.parsers.expat.xmlparser* 메서드), 1458
xml.dom
    module, 1426
xml.dom.minidom
    module, 1436
xml.dom.pulldom
    module, 1440
xml.etree.ElementInclude
    module, 1418
xml.etree.ElementTree
    module, 1408
XMLFilterBase (*xml.sax.saxutils* 클래스), 1450
XMLGenerator (*xml.sax.saxutils* 클래스), 1450
XMLID() (*xml.etree.ElementTree* 모듈), 1417
XMLNS_NAMESPACE (*xml.dom* 모듈), 1428
XMLParser (*xml.etree.ElementTree* 클래스), 1424
xml.parsers.expat
    module, 1454
xml.parsers.expat.errors
    module, 1461
xml.parsers.expat.model
    module, 1461
XMLParserType (*xml.parsers.expat* 모듈), 1455
XMLPullParser (*xml.etree.ElementTree* 클래스), 1425
XMLReader (*xml.sax.xmlreader* 클래스), 1450
xmlrpc
    module, 1575
xmlrpc.client
    module, 1576
xmlrpc.server
    module, 1583
xml.sax
    module, 1442
xml.sax.handler
    module, 1444
xml.sax.saxutils
    module, 1449
xml.sax.xmlreader
    module, 1450
xor() (*operator* 모듈), 453
xview() (*tkinter.ttk.Treeview* 메서드), 1661

Y

통합 개발 환경, 1666
ycor() (*turtle* 모듈), 1690
트레이스백
    객체, 1986, 2065
파이썬 편집기, 1666
파일
    .ini, 649
    mime.types, 1391
    .pdbrc, 1933
    plist, 670
    경로 구성, 2109
    구성, 649
    대용량 파일, 2269
    디버거 구성, 1933
    바이트 코드, 2217
    복사하기, 511
    임시, 501
파일 객체
    io 모듈, 757
파일 복사하기, 511
파일 제어
    UNIX, 2275
파일명
    경로명 확장, 506
    와일드카드 확장, 509
    임시, 501
패키지, 2109
패킹
    바이너리 데이터, 191
패킹 (위젯), 1635
평탄화
    객체, 523
표 형식
    데이터, 641
프로세서 시간, 775, 780
프로세스, 경량, 1105
프로토콜
    FTP, 1495, 1517
    HTTP, 1495, 1506, 1510, 1556
    IMAP4, 1527
    IMAP4_SSL, 1527
    IMAP4_stream, 1527
    POP3, 1524
    SMTP, 1534
    복사, 531
프로파일 함수, 998, 1993, 2001
프로파일러, 1993, 2001
프로파일링, 결정론적, 1939
프로퍼티 리스트, 670
프롬프트, 인터프리터, 2001
피클
    모듈, 318, 540, 543
피클링
    객체, 523

year
    calendar 명령줄 옵션, 268
year (*datetime.datetime*의 속성), 232
year (*datetime.date*의 속성), 225
yeardatescalendar() (*calendar.Calendar* 메서드), 262
yeardays2calendar() (*calendar.Calendar* 메서드), 262
yeardayscalendar() (*calendar.Calendar* 메서드), 262
헤더
    MIME, 1390
YES (*tkinter.messagebox* 모듈), 1646
YESEXPR (*locale* 모듈), 1620
YESNO (*tkinter.messagebox* 모듈), 1646
YESNOCANCEL (*tkinter.messagebox* 모듈), 1646
협정 세계시, 772
환경
    가상, 1967
환경 변수
    BROWSER, 1465, 1466
    COLUMNS, 964
    COMSPEC, 747, 1082
    DISPLAY, 1629
    HOME, 488, 1629
    HOMEDRIVE, 488
    HOMEPATH, 488
    IDLESTARTUP, 1673, 1674
    LANG, 1609, 1610, 1617, 1621
    LANGUAGE, 1609, 1610
    LC_ALL, 1609, 1610
    LC_MESSAGES, 1609, 1610
    LINES, 959, 964
    LOGNAME, 694, 954
    MANPAGER, 1773
    no_proxy, 1482
    PAGER, 1773
    PATH, 484, 516, 739, 745, 746, 755, 1081, 1465, 1969, 1970, 2109
    PATHEXT, 516
    PYTHON_CONTEXT_AWARE_WARNINGS, 1988, 2032
    PYTHON_CPU_COUNT, 754, 1026
    PYTHON_DOM, 1427
    PYTHON_GIL, 1988, 2308
    PYTHON_JIT, 1997
    PYTHON_THREAD_INHERIT_CONTEXT, 1988
    PYTHONASYNCIODEBUG, 1173, 1209, 1775
    PYTHONBREAKPOINT, 10, 1984
    PYTHONCASEOK, 34
    PYTHONCOERCECLOCALE, 691
    PYTHONDEVMODE, 1774
    PYTHONDONTWRITEBYTECODE, 1985
    PYTHONFAULTHANDLER, 1774, 1925
    PYTHONHOME, 1906, 2158, 2159
    PYTHONINTMAXSTRDIGITS, 109, 1996
    PYTHONIOENCODING, 690, 2005
    PYTHONLEGACYWINDOWSFSENCODING, 2005
    PYTHONLEGACYWINDOWSSTDIO, 2005
    PYTHONMALLOC, 1774
    PYTHONNOUSERSITE, 2110
    PYTHONPATH, 1906, 1999, 2158
    PYTHONPLATLIBDIR, 2158
    PYTHONPYCACHEPREFIX, 1985
    PYTHONSAFEPATH, 1999, 2299
    PYTHONSTARTUP, 187, 1110, 1673, 1674, 1996, 2110
    PYTHONTRACEMALLOC, 1954, 1960
    PYTHONTZPATH, 257, 261
    PYTHONUNBUFFERED, 2005
    PYTHONUSERBASE, 2111
    PYTHONUSERSITE, 1906
    PYTHONUTF8, 691, 2005
    PYTHONWARNDEFAULTENCODING, 758
    PYTHONWARNINGS, 1774, 2026, 2027
    SOURCE_DATE_EPOCH, 2217, 2220
    SSLKEYLOGFILE, 1243, 1244
    SystemRoot, 1084
    TEMP, 504
    TERM, 963
    TMP, 504
    TMPDIR, 504
    TZ, 780, 781
    USER, 954
    USERNAME, 488, 694, 954
    USERPROFILE, 488
Yield (*ast* 클래스), 2193
YIELD_VALUE (*opcode*), 2232
YieldFrom (*ast* 클래스), 2193
yiq_to_rgb() (*colorsys* 모듈), 1608
yview() (*tkinter.ttk.Treeview* 메서드), 1661

Z

z
    문자열 포매팅에서, 129
z85decode() (*base64* 모듈), 1397
z85encode() (*base64* 모듈), 1397
Zen of Python (파이썬 젠), **2318**
ZeroDivisionError, 118
zfill() (*bytearray* 메서드), 81
zfill() (*bytes* 메서드), 81
zfill() (*str* 메서드), 64
zip()
    built-in function, 32
ZIP_BZIP2 (*zipfile* 모듈), 612
ZIP_DEFLATED (*zipfile* 모듈), 612
zip_longest() (*itertools* 모듈), 434
ZIP_LZMA (*zipfile* 모듈), 612
ZIP_STORED (*zipfile* 모듈), 612
ZIP_ZSTANDARD (*zipfile* 모듈), 613
zipapp
    module, 1976
zipapp 명령줄 옵션
    -c, 1977
    --compress, 1977
    -h, 1977

--help, 1977
--info, 1977
-m, 1977
--main, 1977
-o, 1977
--output, 1977
-p, 1977
--python, 1977
zipfile
module, 611
ZipFile (*zipfile* 클래스), 613
zipfile 명령줄 옵션
-c, 622
--create, 622
-e, 622
--extract, 622
-l, 622
--list, 622
--metadata-encoding, 622
-t, 622
--test, 622
zipimport
module, 2117
zipimporter (*zipimport* 클래스), 2118
ZipImportError, 2118
ZipInfo (*zipfile* 클래스), 612
zlib
module, 594
ZLIB_RUNTIME_VERSION (*zlib* 모듈), 597
ZLIB_VERSION (*zlib* 모듈), 597
ZLIBNG_VERSION (*zlib* 모듈), 597
zoneinfo
module, 256
ZoneInfo (*zoneinfo* 클래스), 258
ZoneInfoNotFoundError, 261
zscore() (*statistics.NormalDist* 메서드), 418
zstd_version_info (*compression.zstd* 모듈), 593
ZstdCompressor (*compression.zstd* 클래스), 585
ZstdDecompressor (*compression.zstd* 클래스), 586
ZstdDict (*compression.zstd* 클래스), 588
ZstdError, 584
ZstdFile (*compression.zstd* 클래스), 584